CS234 Notes - Lecture 1
Introduction to Reinforcement Learning

Michael Painter, Emma Brunskill

March 20, 2018

1 Introduction

In **Reinforcement Learning** we consider the problem of learning how to act, through experience and without an explicit teacher. A reinforcement learning agent must interact with its world and from that learn how to maximize some cumulative reward over time.

Reinforcement learning has become increasingly more popular over recent years, likely due to large advances in the subject, such as Deep Q-Networks [1]. Moreover, other areas of Artificial Intelligence are seeing plenty of success stories by borrowing and utilizing concepts from Reinforcement Learning. For example, in game playing AlphaGo used Reinforcement Learning methods to reach superhuman performance in Go [3], and Reinforcement Learning concepts are also often borrowed in the training of Generative Adversarial Networks [2].

Many people often wonder how Reinforcement Learning differs from other types of learning. In **Supervised Learning** we are given a dataset, which consists of examples and labels. In the Supervised Learning setting, we are given a training set where for each example, we are provided the correct label (classification problems) / correct output (regression problems). In contrast, when no labels are provided in a dataset, **Unsupervised Learning** refers to methods that find underlying, latent structure in some data, when there is not labels for each example. However, in a Reinforcement Learning setting, we are dealing with making decisions and comparing actions that could be taken, rather than making predictions. A Reinforcement Learning agent may interact with the world, and receive some immediate, partial feedback signal — commonly called a **Reward** — for each interaction. However, the agent is given little indication if the action it took was actually the 'best' it could have chosen, and the agent must somehow learn to pick actions that will maximize a long term cumulative reward. Therefore, because of the weak/incomplete feedback provided by the reward signal we could consider Reinforcement Learning to lie somewhere between Supervised Learning, which gives strong feedback with labeled data, and Unsupervised Learning, with no feedback or labels.

The Reinforcement Learning setting introduces a number of challenges that we need to overcome, and potentially make trade-offs between. The agent must be able to *optimize* its actions to maximize the reward signal it receives. However, as the agent needs to learn by interacting with its environment, *exploration* is required. This leads to a natural trade-off between exploration and exploitation, where the agent needs to decide between potentially finding new, better strategies at the risk of a receiving a lower reward, or, if it should exploit what it already knows. Another question we face is, can the agent *generalize* its experience? That is, can it learn whether some actions are good/bad in previously unseen states? And finally, we also need to consider *delayed consequences* of the agents actions, that is, if it receive a high reward, was it because of an action it just took, or because of an action taken much earlier?

2 Overview of reinforcement learning

2.1 Sequential Decision Making

Commonly we will consider the problem of making a sequence of good decisions. To formalize this, in a discrete setting, an agent will make sequence of **actions** $\{a_t\}$, observe a sequence of **observations** $\{o_t\}$ and receive a sequence of **rewards** $\{r_t\}$. We define the **history** at time t to be $h_t = (a_1, o_1, r_1, ..., a_t, o_t, r_t)$. The agent's choice of what action to take next can be viewed as a function of the history, that is, $a_{t+1} = f(h_t)$, and the problem of sequential decision making can be thought of as defining and computing the function f appropriately.

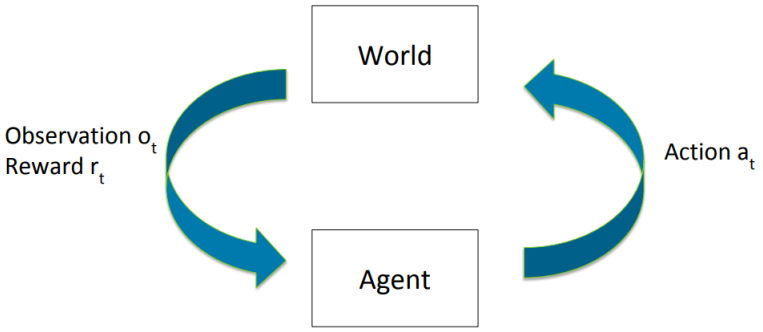


Figure 1: Overview of how an agent interacts with its world.

When the model of the sequential decision making process is known, in some deterministic and finite settings AI techniques like *A* search* and *minimax* can be used to find an optimal sequence of actions.

However, when we have a very large number of possible states (potentially infinite), or introduce stochasticity into our model of the world, brute-force search becomes infeasible. In such a setting, it becomes *necessary* to incorporate generalization to make the task feasible, as in the Atari example seen in class. Moreover, without the ability to exhaustively search, the agent must decide on how to strategically make actions to balance short and long term rewards.

2.2 Modeling the world

Let S be the set of possible states that our world may be in, let $\{s_t\}$ be a sequence of states observed, indexed by time and let A be the set of possible actions. Often we will want to consider the **transition dynamics** of the world $P(s_{t+1}|s_t, a_t, ..., s_1, a_1)$, a probability distribution over S, and a function of previous states and actions. In reinforcement learning, we often assume the **Markov Property** that

$$P(s_{t+1}|s_t, a_t, ..., s_1, a_1) = P(s_{t+1}|s_t, a_t), \tag{1}$$

which in practice is flexible enough for our needs. A useful trick to make sure that the Markov property holds is to use the history h_t *as* our state.

Usually, we consider the reward r_t to be received on the transition between states, $s_t \xrightarrow{a_t} s_{t+1}$. A **reward function** is used to predict rewards, $R(s, a, s') = \mathbb{E}[r_t|s_t = s, a_t = a, s_{t+1} = s']$. We will often consider the reward function to be of the form $R(s) = \mathbb{E}[r_t|s_t = s]$ or $R(s, a) = \mathbb{E}[r_t|s_t = s, a_t = a]$. It will also often be the case that $r_t|s_t = s$ is degenerate, that is, r_t has a fixed value, given $s_t = s$.

A **model** consists of the above transition dynamics and reward function.

2.3 Components of an reinforcement learning agent

First, let the **agent state** be a function of the history, $s_t^a = g(h_t)$. A reinforcement learning agent typically has an explicit representation of one or more of the following three things: a policy, a value function and optionally a model. A **policy** π is a mapping from the agent state to an action, $\pi(s_t^a) \in A$, or, sometimes it is a stochastic distribution over actions $\pi(a_t|s_t^a)$. When the agent wants to take an action and π is stochastic, it picks action $a \in A$ with probability $P(a_t = a) = \pi(a|s_t^a)$. Given a policy π and discount factor $\gamma \in [0, 1]$, a **value function** V^π is an expected sum of discounted rewards,

$$V^\pi(s) = \mathbb{E}_\pi[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... \ |s_t = s]. \tag{2}$$

Where $\mathbb{E}_\pi$ denotes that the expectation is taken over states encountered by following policy π, and, the **discount factor** γ is used to weigh immediate rewards versus delayed rewards. Lastly, a reinforcement learning agent may have a model, which is defined in section 2.2. If the agent has a model, we would call it a **model-based** agent, and if it doesn't incorporate a model, we would call it a **model-free** agent.

So far, we have consider definitions in a very general setting, and we have not made any assumptions about the relationship between o_t and s_t. We call the case where $o_t \neq s_t$ **partially observable**, and it is common in partially observable settings for RL algorithms to maintain a probability distribution over the true world state to define s_t^a, which is known as a **belief state**.

However, for the majority of the class, we will consider the **fully observable** case, where $o_t = s_t$, and will assume that $s_t^a = s_t$.

2.4 Taxonomy of reinforcement learning agents

We can classify our agents in a number of ways, as can be seen in table 1, and each type of agent isn't necessarily unique. For example, an actor critic agent could also be a model free agent. An overview of ways that we can classify agents can also be seen in figure 2.

Agent type	Policy	Value Function	Model
Value Based	Implicit	✓	?
Policy Based	✓	X	?
Actor Critic	✓	✓	?
Model Based	?	?	✓
Model Free	?	?	X

Table 1: An overview of properties of different types of reinforcement learning agent. Where a checkmark indicates that the agent has the component, a cross indicates that it must not have the component, and a question mark indicates that the agent may have that component, but isn't required to have it.

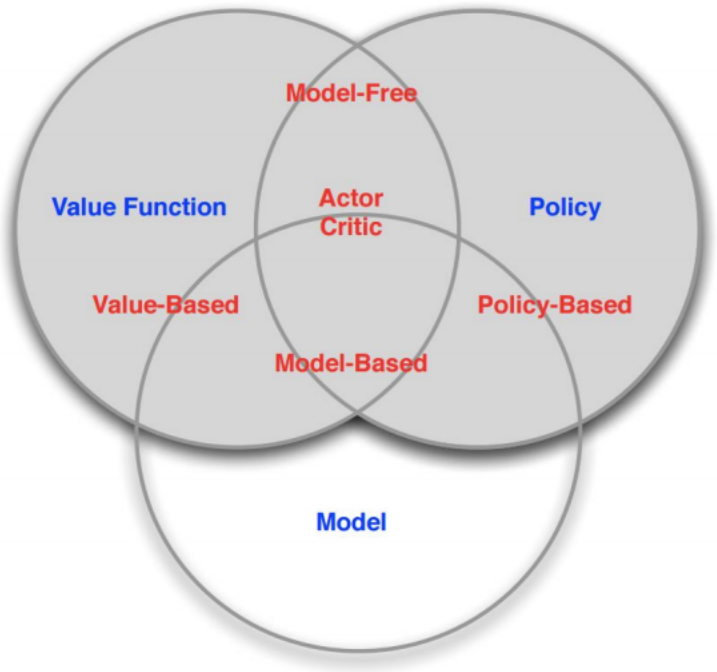


Figure 2: Classification of different reinforcement learning agents. Figure from David Silver. [4]

2.5 Continuous domains

For simplicity, we have focused our discussion on only discrete state and action spaces, with discrete time-steps. However, there are many applications — especially within Robotics and Control — that is most appropriately modeled with continuous state and action spaces, with continuous time. The discussion above can be generalized to incorporate these continuous settings.

References

[1] Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature 518.7540 (2015): 529-533.

[2] Pfau, David, and Oriol Vinyals. "Connecting generative adversarial networks and actor-critic methods." arXiv preprint arXiv:1610.01945 (2016).

[3] Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." Nature 529.7587 (2016): 484-489.

[4] Silver, David. "Reinforcement Learning." 15 Jan. 2016. Reinforcement Learning, UCL.

CS234 Notes - Lecture 2
Making Good Decisions Given a Model of the World

Rahul Sarkar, Emma Brunskill

March 20, 2018

3 Acting in a Markov decision process

We begin this lecture by recalling the definitions of a **model**, **policy** and **value function** for an agent. Let the agent's state and action spaces be denoted by S and A respectively. We then have the following definitions:

- **Model** : A model is the mathematical description of the dynamics and rewards of the agent's environment, which includes the transition probabilities $P(s'|s,a)$ of being in a successor state $s' \in S$ when starting from a state $s \in S$ and taking an action $a \in A$, and the rewards $R(s,a)$ (either deterministic or stochastic) obtained by taking an action $a \in A$ when in a state $s \in S$.
- **Policy** : A policy is a function $\pi : S \to A$ that maps the agent's states to actions. Policies can be stochastic or deterministic.
- **Value function** : The value function V^π corresponding to a particular policy π and for a state $s \in S$, is the cumulative sum of future (discounted) rewards obtained by the agent, by starting from the state s and following the policy.

We also recall the notion of **Markov property** from the last lecture. Consider a stochastic process $(s_0, s_1, s_2, \dots)$ evolving according to some transition dynamics. We say that the stochastic process has the Markov property if and only if $P(s_i|s_0, \dots, s_{i-1}) = P(s_i|s_{i-1})$, $\forall\ i = 1, 2, \dots$, i.e. the transition probability of the next state conditioned on the history including the current state is equal to the transition probability of the next state conditioned only on the current state. In such a scenario, the current state is a sufficient statistic of history of the stochastic process, and we say that *"the future is independent of the past given present."*

In this lecture, we will build on these definitions and proceed in order by first defining a **Markov process (MP)**, followed by the definition of a **Markov reward process (MRP)** and finally build on both of them to define a **Markov decision process (MDP)**. We will finish this lecture by discussing some algorithms which enable us to make good decisions when a MDP is completely known.

3.1 Markov process

In its most generality, a Markov process is a stochastic process that satisfies the Markov property, because of which we say that a Markov process is *"memoryless"*. For the purpose of this lecture, we will make two additional assumptions that are very common in the reinforcement learning setting:

- *Finite state space* : The state space of the Markov process is finite. This means that for the Markov process $(s_0, s_1, s_2, \dots)$, there is a state space S with $|S| < \infty$, such that for all realizations of the Markov process, we have $s_i \in S$ for all $i = 1, 2, \dots$.

- *Stationary transition probabilities* : The transition probabilities are time independent. Mathematically, this means the following:

$$P(s_i = s' | s_{i-1} = s) = P(s_j = s' | s_{j-1} = s) \ , \ \forall \, s, s' \in S \ , \ \forall \, i, j = 1, 2, \dots \ . \tag{1}$$

Unless otherwise specified, we will always assume that these two properties hold for any Markov process that we will encounter in this lecture, including for any Markov reward process and any Markov decision process to be defined later by adding progressively extra structure to the Markov process. Note that a Markov process satisfying these assumptions is also sometimes called a *"Markov chain"*, although the precise definition of a Markov chain varies.

For the Markov process, these assumptions lead to a nice characterization of the transition dynamics in terms of a *transition probability matrix* $\mathbf{P}$ of size $|S| \times |S|$, whose (i, j) entry is given by $P_{ij} = P(j|i)$, with i, j referring to the states of S ordered arbitrarily. It should be noted that the matrix $\mathbf{P}$ is a non-negative row-stochastic matrix, i.e. the sum of each row equals 1.

Henceforth, we will thus define a Markov process by the tuple $(S, \mathbf{P})$, which consists of the following:

- S : A finite state space.
- $\mathbf{P}$: A transition probability model that specifies $P(s'|s)$.

Exercise 3.1. (a) Prove that $\mathbf{P}$ is a row-stochastic matrix. (b) Show that 1 is an eigenvalue of any row-stochastic matrix, and find a corresponding eigenvector. (c) Show that any eigenvalue of a row-stochastic matrix has maximum absolute value 1.

Exercise 3.2. The *max-norm* or *infinity-norm* of a vector $x \in \mathbb{R}^n$ is denoted by $||x||_\infty$, and defined as $||x||_\infty = \max_i |x_i|$, i.e. it is the component of x with the maximum absolute value. For any matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, define the following quantity

$$||\mathbf{A}||_\infty = \sup_{\substack{x \in \mathbb{R}^n \\ x \neq 0}} \frac{||\mathbf{A}x||_\infty}{||x||_\infty} \ . \tag{2}$$

(a) Prove that $||\mathbf{A}||_\infty$ satisfies all the properties of a norm. The quantity so defined is called the *"induced infinity norm"* of a matrix.

(b) Prove that

$$||\mathbf{A}||_\infty = \max_{i=1,\dots,m} \left(\sum_{j=1}^{n} |A_{ij}| \right) \ . \tag{3}$$

(c) Conclude that if $\mathbf{A}$ is row-stochastic, then $||\mathbf{A}||_\infty = 1$.

(d) Prove that for every $x \in \mathbb{R}^n$, $||\mathbf{A}x||_\infty \leq ||\mathbf{A}||_\infty ||x||_\infty$.

3.1.1 Example of a Markov process : Mars Rover

To practice our understanding, consider the Markov process shown in Figure 1. Our agent is a Mars rover whose state space is given by $S = \{S1, \ S2, \ S3, \ S4, \ S5, \ S6, \ S7\}$. The transition probabilities of the states are indicated in the figure with arrows. So for example if the rover is in the state $S4$ at

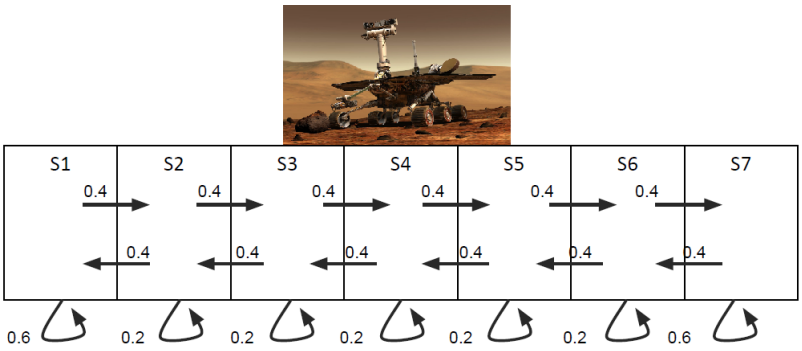


Figure 1: Mars Rover Markov process.

the current time step, in the next time step it can go to the states $S3,\ S4,\ S5$ with probabilities given by $0.4,\ 0.2,\ 0.4$ respectively.

Assuming that the rover starts out in state $S4$, some possible episodes of the Markov process could look as follows:

- $S4,\ S5,\ S6,\ S7,\ S7,\ S7,\ \ldots$
- $S4,\ S4,\ S5,\ S4,\ S5,\ S6,\ \ldots$
- $S4,\ S3,\ S2,\ S1,\ \ldots$

Exercise 3.3. Consider the example of a Markov process given in Figure 1. (a) Write down the transition probability matrix for the Markov process.

3.2 Markov reward process

A Markov reward process is a Markov process, together with the specification of a reward function and a discount factor. It is formally represented using the tuple $(S, \mathbf{P}, R, \gamma)$ which are listed below:

- S : A finite state space.
- $\mathbf{P}$: A transition probability model that specifies $P(s'|s)$.
- R : A reward function that maps states to rewards (real numbers), i.e $R : S \to \mathbb{R}$.
- γ: Discount factor between 0 and 1.

We have already explained the roles played by S and $\mathbf{P}$ in the context of a Markov process. We will next explain the concept of the reward function R and the discount factor γ, which are specific to the Markov reward process. Additionally, we will also define and explain a few quantities which are important in this context, such as the horizon, return and state value function of a Markov reward process.

3.2.1 Reward function

In a Markov reward process, whenever a transition happens from a current state s to a successor state s', a reward is obtained depending on the current state s. Thus for the Markov process $(s_0, s_1, s_2, \ldots)$, each transition $s_i \to s_{i+1}$ is accompanied by a reward r_i for all $i = 0, 1, \ldots$, and so a particular episode

of the Markov reward process is represented as $(s_0, r_0, s_1, r_1, s_2, r_2, \dots)$. We should note that these rewards can be either deterministic or stochastic. For a state $s \in S$, we define the expected reward $R(s)$ by:

$$R(s) = \mathbb{E}[r_0 | s_0 = s], \tag{4}$$

that is $R(s)$ is the expected reward obtained during the first transition, when the Markov process starts in state s. Just like the assumption of stationary transition probabilities, going forward we will also assume the following:

- *Stationary rewards* : The rewards in a Markov reward process are stationary which means that they are time independent. In the deterministic case, mathematically this means that for all realizations of the process we must have that:

$$r_i = r_j \text{ , whenever } s_i = s_j \ \ \forall\, i, j = 0, 1, \dots \text{ ,} \tag{5}$$

  while in the case of stochastic rewards we require that the cumulative distribution functions (cdf) of the rewards conditioned on the current state be time independent. This is written mathematically as:

$$F(r_i | s_i = s) = F(r_j | s_j = s) \ \ , \ \ \forall\, s \in S \ \ , \ \ \forall\, i, j = 0, 1, \dots \text{ ,} \tag{6}$$

  where $F(r_i | s_i = s)$ denotes the cdf of r_i conditioned on the state $s_i = s$. Notice that as a consequence of (5) and (6), we furthermore have the following result about the expected rewards:

$$R(s) = \mathbb{E}[r_i | s_i = s] \ \ , \ \ \forall\, i = 0, 1, \dots \text{ .} \tag{7}$$

We will see that as long as the "stationary rewards" assumption is true about a Markov reward process, only the expected reward R matters in the things that we will be interested in, and we can depose of the quantities r_i entirely. Hence going forward, the word "reward" will be used interchangeably to mean both R and r_i, and should be easily understood from context. Finally notice that R can be represented as a vector of dimension $|S|$, in the case of a finite state space S.

Exercise 3.4. (a) Under the assumptions of stationary transition probabilities and rewards, prove equation (7).

3.2.2 Horizon, Return and Value function

We next define the notions of the horizon, return and value function for a Markov reward process.

- **Horizon** : The horizon H of a Markov reward process is defined as the number of time steps in each episode (realization) of the process. The horizon can be finite or infinite. If the horizon is finite, then the process is also called a *finite Markov reward process*.
- **Return** : The return G_t of a Markov reward process is defined as the discounted sum of rewards starting at time t up to the horizon H, and is given by the following mathematical formula:

$$G_t = \sum_{i=t}^{H-1} \gamma^{i-t} r_i \ \ , \ \ \forall\, 0 \leq t \leq H-1. \tag{8}$$

- **State value function** : The state value function $V_t(s)$ for a Markov reward process and a state $s \in S$ is defined as the expected return starting from state s at time t, and is given by the following expression:

$$V_t(s) = \mathbb{E}[G_t | s_t = s]. \tag{9}$$

  Notice that when the horizon H is infinite, this definition (9) together with the stationary assumptions of the rewards and transition probabilities imply that $V_i(s) = V_j(s)$ for all $i, j = 0, 1, \dots$, and thus in this case we will define:

$$V(s) = V_0(s) \text{ .} \tag{10}$$

Exercise 3.5. (a) If the assumptions of stationary transition probabilities and stationary rewards hold, and if the horizon H is infinite, then using the definitions in (8) and (9) prove that $V_i(s) = V_j(s)$ for all $i, j = 0, 1, \ldots$.

3.2.3 Discount factor

Notice that in the definition of return G_t in (8), if the horizon is infinite and $\gamma = 1$, then the return can become infinite even if the rewards are all bounded. If this happens, then the value function $V(s)$ can also become infinite. Such problems cannot then be solved using a computer. To avoid such mathematical difficulties and make the problems computationally tractable we set $\gamma < 1$, which exponentially weighs down the contribution of rewards at future times, in the calculation of the return in (8). This quantity γ is called the *discount factor*. Other than for purely computational reasons, it should be noted that humans behave in much the same way - we tend to put more importance in immediate rewards over rewards obtained at a later time. The interpretation of γ is that when $\gamma = 0$, we only care about the immediate reward, while when $\gamma = 1$, we put as much importance on future rewards as compared the present. Finally, notice that if the horizon of the Markov reward process is finite, i.e. $H < \infty$, then we can set $\gamma = 1$, as the returns and value functions are always finite.

Exercise 3.6. Consider a finite horizon Markov reward process, with bounded rewards. Specifically assume that $\exists\ M \in (0, \infty)$ such that $|r_i| \leq M\ \ \forall\ i$ and across all episodes (realizations). (a) Show that the return for any episode G_t as defined in (8) is bounded. (b) Can you suggest a bound? Specifically can you find $C(M, \gamma, t, H)$ such that $|G_t| \leq C$ for any episode?

Exercise 3.7. Consider an infinite horizon Markov reward process, with bounded rewards and $\gamma < 1$. (a) Prove that the return for any episode G_t as defined in (8) converges to a finite limit. *Hint: Consider the partial sums* $S_N = \sum_{i=t}^{N} \gamma^{i-t} r_i$ *for* $N \geq t$. *Show that* $\{S_N\}_{N \geq t}$ *is a Cauchy sequence.*

3.2.4 Example of a Markov reward process : Mars Rover

As an example, consider the Markov reward process in Figure 2. The states and the transition probabilities of this process are exactly the same as in the Mars rover Markov process example of Exercise 3.3. The rewards obtained by executing an action from any of the states $\{S2,\ S3,\ S4,\ S5,\ S6\}$ is 0, while any moves from states $S1,\ S7$ yield rewards 1, 10 respectively. The rewards are stationary and deterministic. Assume $\gamma = 0.5$ in this example.

For illustration, let us again assume that the rover is initially in state $S4$. Consider the case when the horizon is finite : $H = 4$. A few possible episodes in this case with the return G_0 in each case are given below:

- $S4,\ S5,\ S6,\ S7,\ S7\ : G_0 = 0 + 0.5 * 0 + 0.5^2 * 0 + 0.5^3 * 10 = 1.25$
- $S4,\ S4,\ S5,\ S4,\ S5\ : G_0 = 0 + 0.5 * 0 + 0.5^2 * 0 + 0.5^3 * 0 = 0$
- $S4,\ S3,\ S2,\ S1,\ S2\ : G_0 = 0 + 0.5 * 0 + 0.5^2 * 0 + 0.5^3 * 1 = 0.125$

3.3 Computing the value function of a Markov reward process

In this section we give three different ways to compute the value function of a Markov reward process:

- Simulation
- Analytic solution
- Iterative solution

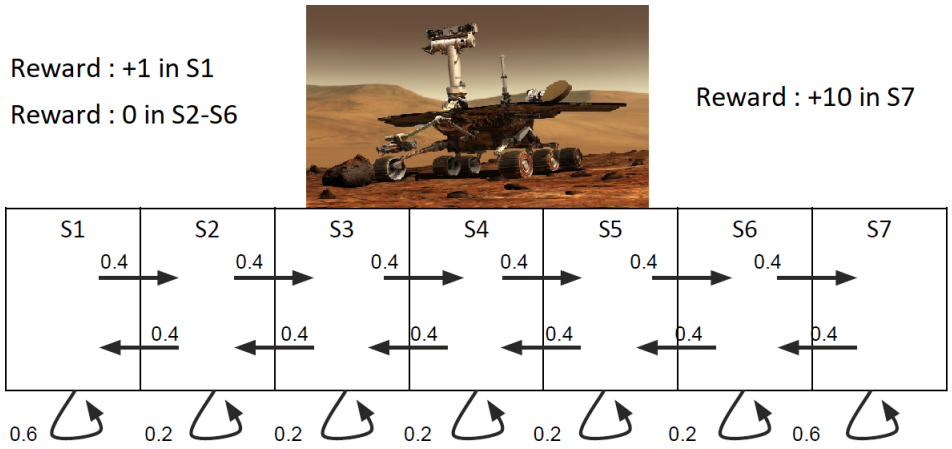


Figure 2: Mars Rover Markov reward process.

3.3.1 Monte Carlo simulation

The first method involves generating a large number of episodes using the transition probability model and rewards of the Markov reward process. For each episode, the returns can be calculated which can then be averaged to give the average returns. Concentration inequalities bound how quickly the averages concentrate to the mean value. For a Markov reward process $M = (S, \mathbf{P}, R, \gamma)$, state s, time t, and the number of simulation episodes N, the pseudo-code of the simulation algorithm is given in Algorithm 1.

Algorithm 1 Monte Carlo simulation to calculate MRP value function

procedure MONTE CARLO EVALUATION(M, s, t, N)
  $i \leftarrow 0$
  $G_t \leftarrow 0$
  while $i \neq N$ **do**
    Generate an episode, starting from state s and time t
    Using the generated episode, calculate return $g \leftarrow \sum_{i=t}^{H-1} \gamma^{i-t} r_i$
    $G_t \leftarrow G_t + g$
    $i \leftarrow i + 1$
  $V_t(s) \leftarrow G_t / N$
  return $V_t(s)$

3.3.2 Analytic solution

This method works only for an infinite horizon Markov reward processes with $\gamma < 1$. Using (9), the fact that the horizon is inifnite, and using the stationary Markov property we have for any state $s \in S$:

$$
\begin{aligned}
V(s) \overset{(a)}{=} V_0(s) = \mathbb{E}[G_0 | s_0 = s] = \mathbb{E}\left[\sum_{i=0}^{\infty} \gamma^i r_i \middle| s_0 = s\right] = \mathbb{E}[r_0 | s_0 = s] + \sum_{i=1}^{\infty} \gamma^i \mathbb{E}[r_i | s_0 = s] \\
\overset{(b)}{=} \mathbb{E}[r_0 | s_0 = s] + \sum_{i=1}^{\infty} \gamma^i \left(\sum_{s' \in S} P(s_1 = s' | s_0 = s) \mathbb{E}[r_i | s_0 = s, s_1 = s'] \right) \\
\overset{(c)}{=} \mathbb{E}[r_0 | s_0 = s] + \gamma \sum_{s' \in S} P(s'|s) \mathbb{E}\left[\sum_{i=0}^{\infty} \gamma^i r_i \middle| s_0 = s'\right] \overset{(d)}{=} R(s) + \gamma \sum_{s' \in S} P(s'|s) V(s') \ ,
\end{aligned}
\tag{11}
$$

where (a) follows from (8), (9), and (10), (b) follows by the law of total expectation, (c) follows from the Markov property and due to stationarity, and (d) follows from (4). There is a nice interpretation of the final result of (11), namely that the first term $R(s)$ is the immediate reward while the second term $\gamma \sum_{s' \in S} P(s'|s)V(s')$ is the discounted sum of future rewards. The value function $V(s)$ is the sum of these two quantities. As $|S| < \infty$, it is possible to write this equation in matrix form as:

$$V = R + \gamma \mathbf{P} V \ , \tag{12}$$

where $\mathbf{P}$ is the transition probability matrix introduced earlier, and R and V are column vectors of dimension $|S|$ formed by stacking all the values $R(s)$ and $V(s)$ respectively, for all $s \in S$. Equation (12) can be rearranged to give $(\mathbf{I} - \gamma \mathbf{P})V = R$, which has an analytical solution $V = (\mathbf{I} - \gamma \mathbf{P})^{-1} R$. Notice that as $\gamma < 1$ and $\mathbf{P}$ is row-stochastic, $(\mathbf{I} - \gamma \mathbf{P})$ is non-singular and hence can be inverted. Thus (12) always has a solution and the solution is unique. However, the computational cost of the analytical method is $O(|S|^3)$, as it involves a matrix inverse and hence it is completely unsuitable for cases where the state space is very large.

Exercise 3.8. Consider the matrix $(\mathbf{I} - \gamma \mathbf{P})$. (a) Show that $1 - \gamma$ is an eigenvalue of this matrix, and find a corresponding eigenvector. (b) For $0 < \gamma < 1$, use the result of Exercise 3.1 to conclude that $(\mathbf{I} - \gamma \mathbf{P})$ is non-singular, and thus invertible.

Exercise 3.9. Consider the Markov reward process introduced in the example in section 3.2.4. (a) If the horizon H is infinite, calculate the value function for all the states.

3.3.3 Iterative solution

We now give an iterative solution to evaluate the value function in the infinite horizon case (with $\gamma < 1$) and a dynamic programming based solution for the finite horizon case. The surprising thing is that both the algorithms look surprisingly similar, to the point that it is hard to tell the difference. We first consider the finite horizon case. It is easy to prove (by following almost exactly the same proof of (11)) that the analog of equation (11) in the finite horizon case is given by:

$$\begin{aligned} V_t(s) &= R(s) + \gamma \sum_{s' \in S} P(s'|s) V_{t+1}(s') \ \ , \ \ \forall \, t = 0, \ldots, H-1, \\ V_H(s) &= 0 \ \ . \end{aligned} \tag{13}$$

Exercise 3.10. Prove equations (13) for a finite horizon Markov reward process.

These equations immediately lend themselves to a dynamic programming solution whose pseudo-code is outlined in Algorithm 2. The algorithm takes as input a finite horizon Markov reward process $M = (S, \mathbf{P}, R, \gamma)$, and computes the value function for all states and at all times.

Algorithm 2 Dynamic programming algorithm to calculate finite MRP value function

procedure DYNAMIC PROGRAMMING VALUE FUNCTION EVALUATION(M)
    For all states $s \in S$, $V_H(s) \leftarrow 0$
    $t \leftarrow H - 1$
    while $t \geq 0$ **do**
        For all states $s \in S$, $V_t(s) = R(s) + \gamma \sum_{s' \in S} P(s'|s) V_{t+1}(s')$
        $t \leftarrow t - 1$
    return $V_t(s)$ for all $s \in S$ and $t = 0, \ldots, H$

Let us now look at the iterative algorithm for the infinite horizon case with $\gamma < 1$. The pseudo-code for this algorithm is presented in Algorithm 3. The algorithm takes as input a Markov reward process $M = (S, \mathbf{P}, R, \gamma)$, and a tolerance ϵ, and computes the value function for all states.

Algorithm 3 Iterative algorithm to calculate MRP value function

procedure ITERATIVE VALUE FUNCTION EVALUATION(M, ϵ)
  For all states $s \in S$, $V'(s) \leftarrow 0$, $V(s) \leftarrow \infty$
  while $||V - V'||_\infty > \epsilon$ **do**
    $V \leftarrow V'$
    For all states $s \in S$, $V'(s) = R(s) + \gamma \sum_{s' \in S} P(s'|s)V(s')$
  return $V'(s)$ for all $s \in S$

For both these algorithms 2 and 3, the computational cost of each loop is $O(|S|^2)$. This is an improvement over the $O(|S|^3)$ cost of the analytical method in the inifinite horizon case, however one may need quite a few iterations to converge depending on the tolerance level ϵ.

While the proof of correctness of algorithm 2 in the finite horizon case is obvious, for the infinite horizon case it is not so clear if algorithm 3 always converges, and if it does whether it converges to the correct solution $(\mathbf{I} - \gamma \mathbf{P})^{-1} R$. The answers to both these questions are affirmative as is shown by the following theorem.

Theorem 3.1. *Algorithm 3 always terminates. Moreover, if the output of the algorithm is V' and we denote the true solution as $V = (\mathbf{I} - \gamma \mathbf{P})^{-1} R$, then we have the error estimate $||V' - V||_\infty \leq \frac{\epsilon\gamma}{1-\gamma}$.*

Proof. We consider the vector space $\mathbb{R}^{|S|}$ equipped with the $||\cdot||_\infty$ norm (see Exercise 3.2), and recall that $\mathbb{R}^{|S|}$ so constructed is a Banach space (see Section A for a discussion on normed vector spaces). We start by noticing that both V and all the iterates of algorithm 3 are elements of $\mathbb{R}^{|S|}$.

Define the operator $B : \mathbb{R}^{|S|} \to \mathbb{R}^{|S|}$ *(also known as the "Bellman backup" operator)* that acts on an element $U \in \mathbb{R}^{|S|}$ as follows

$$(BU)(s) = R(s) + \gamma \sum_{s' \in S} P(s'|s)U(s') \quad , \quad \forall\, s \in S, \tag{14}$$

which can be written in compact matrix-vector notation as

$$BU = R + \gamma \mathbf{P} U \ . \tag{15}$$

We first prove that the operator B is a strict contraction (defined in Definition A.3). For every $U_1, U_2 \in \mathbb{R}^{|S|}$, using (15) we have

$$\begin{aligned} ||BU_1 - BU_2||_\infty &= \gamma ||\mathbf{P}U_1 - \mathbf{P}U_2||_\infty = \gamma ||\mathbf{P}(U_1 - U_2)||_\infty \\ &\leq \gamma ||\mathbf{P}||_\infty ||U_1 - U_2||_\infty = \gamma ||U_1 - U_2||_\infty \ , \end{aligned} \tag{16}$$

where the second step follows by Exercise 3.2, and thus as $0 < \gamma < 1$, we conclude that B is a strict contraction on $\mathbb{R}^{|S|}$. Thus by the contraction mapping theorem (Theorem A.5), we conclude that B has a unique fixed point. From (15) and (12) it also follows that $BV = R + \gamma \mathbf{P} V = V$, and hence V is a fixed point of B, and hence by uniqueness it must also be the only fixed point.

We next consider the iterates produced by algorithm 3 (if it is not allowed to terminate) and denote them by $\{V_k\}_{k \geq 1}$. Notice that these iterates satisfy the following relations

$$V_k = \begin{cases} 0 & \text{if } \ k = 1, \\ BV_{k-1} & \text{if } \ k > 1 \end{cases} \ . \tag{17}$$

By Theorem A.5, we further conclude that $\{V_k\}_{k \geq 1}$ is a Cauchy sequence, and hence by Definition A.1 we conclude that $\exists\ N \geq 1$, such that $||V_m - V_n||_\infty < \epsilon$ for all $m, n > N$. This completes the proof that algorithm 3 terminates. Notice that the contraction mapping theorem (Theorem A.5) also implies that $V_k \to V$ (see Definition A.2 for exact notion of convergence).

To prove the error bound when the algorithm terminates, let the algorithm terminate after k iterations, and so the last iterate is V_{k+1}. We then have $||V_{k+1} - V_k||_\infty \leq \epsilon$. Then using the triangle inequality and the fact that $V_{k+1} = BV_k$ we get,

$$\begin{aligned} ||V_k - V||_\infty &\leq ||V_k - V_{k+1}||_\infty + ||V_{k+1} - V||_\infty = ||V_k - V_{k+1}||_\infty + ||BV_k - BV||_\infty \\ &\leq ||V_k - V_{k+1}||_\infty + \gamma||V_k - V||_\infty = \epsilon + \gamma||V_k - V||_\infty \ , \end{aligned} \tag{18}$$

and so $||V_k - V||_\infty \leq \frac{\epsilon}{1-\gamma}$. This finally allows us to conclude that

$$||V_{k+1} - V||_\infty = ||BV_k - BV||_\infty \leq \gamma||V_k - V||_\infty \leq \frac{\epsilon\gamma}{1-\gamma} \ . \tag{19}$$

□

Exercise 3.11. Suppose that in algorithm 3, the initialization step is changed so V' is set randomly (all entries finite), instead of $V' \leftarrow 0$. (a) Will the algorithm still converge? (b) Does the algorithm still retain the same error estimate of Theorem 3.1 ?

Exercise 3.12. Suppose the assumptions of Theorem 3.1 hold. Using the same notations as in the theorem prove the following:
(a) For all $k \geq 1$, $||V_k - V||_\infty \leq \gamma^{k-1}||V||_\infty$.
(b) $||V_2||_\infty \leq (1+\gamma)||V||_\infty$.
(c) For all $m, n \geq 1$, $||V_m - V_n||_\infty \leq (\gamma^{m-1} + \gamma^{n-1})||V||_\infty$.

3.4 Markov decision process

We are now in a position to define a Markov decision process (MDP). A MDP inherits the basic structure of a Markov reward process with some important key differences, together with the specification of a set of actions that an agent can take from each state. It is formally represented using the tuple (S, A, P, R, γ) which are listed below:

- S : A finite state space.
- A : A finite set of actions which are available from each state s.
- P : A transition probability model that specifies $P(s'|s, a)$.
- R : A reward function that maps a state-action pair to rewards (real numbers), i.e. $R : S \times A \to \mathbb{R}$.
- γ: Discount factor $\gamma \in [0, 1]$.

Some of these quantities have been explained in the context of a Markov reward process. However in the context of a MDP, there are important differences that we need to mention. The basic model of the dynamics is that there is a state space S, and an action space A, both of which we will consider to be finite. The agent starts from a state s_i at time i, chooses an action a_i from the action space, obtains a reward r_i and then reaches a successor state s_{i+1}. An episode of a MDP is thus represented as $(s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, \dots)$.

Unlike in the case of a Markov process or a Markov reward process where the transition probability was only a function of the successor state and the current state, in the case of a MDP the transition probabilities at time i are a function of the successor state s_{i+1} along with both the current state s_i and the action a_i, written as $P(s_{i+1}|s_i, a_i)$. We still assume the principle of stationary transition probabilities which in the context of a MDP is written mathematically as

$$P(s_i = s'|s_{i-1} = s, a_{i-1} = a) = P(s_j = s'|s_{j-1} = s, a_{j-1} = a), \tag{20}$$

for all $s, s' \in S$, for all $a \in A$, and for all $i, j = 1, 2, \ldots$.

The reward r_i at time i depends on both s_i and a_i in the case of a MDP, in contrast to a Markov reward process where it depended only on the current state. These rewards can be stochastic or deterministic, but just like in the case of a Markov reward process, we will assume that the rewards are stationary and the only relevant quantity will be the expected reward which we will denote by $R(s, a)$ for a fixed state s and action a, and defined below:

$$R(s, a) = \mathbb{E}[r_i | s_i = s, a_i = a] \ , \ \ \forall \ i = 0, 1, \ldots \ . \tag{21}$$

The notions of the **discount factor** γ, **horizon** H and **return** G_t for a MDP are exactly equivalent to those in the case of a Markov reward process. However the notion of a **state value function** is slightly modified for a MDP as explained next.

3.4.1 MDP policies and policy evaluation

Given a MDP, a policy for the MDP specifies what action to take in each state. A policy can either be deterministic or stochastic. To cover both these cases, we will consider a policy to be a probability distribution over actions given the current state. It is important to note that the policy may be varying with time, which is especially true in the case of finite horizon MDPs. We will denote a generic policy by the boldface symbol $\boldsymbol{\pi}$, defined as the infinite dimensional tuple $\boldsymbol{\pi} = (\pi_0, \pi_1, \ldots)$, where π_t refers to the policy at time t. We will call policies that do not vary with time *"stationary policies"*, and indicate them as π, i.e. in this case $\boldsymbol{\pi} = (\pi, \pi, \ldots)$. For a stationary policy π, if at time t the agent is in state s, it will choose an action a with probability given by $\pi(a|s)$ and this probability does not depend on t, while for a non-stationary policy the probability will depend on time t and we will be denoted by $\pi_t(a|s)$.

Given a policy $\boldsymbol{\pi}$ one can define two quantities : *the state value function* and *the state-action value function* for the MDP corresponding to the policy $\boldsymbol{\pi}$, as shown below:

- **State value function** : The state value function $V_t^{\boldsymbol{\pi}}(s)$ for a state $s \in S$ is defined as the expected return starting from the state $s_t = s$ at time t and following policy $\boldsymbol{\pi}$, and is given by the expression $V_t^{\boldsymbol{\pi}}(s) = \mathbb{E}_{\boldsymbol{\pi}}[G_t | s_t = s]$, where $\mathbb{E}_{\boldsymbol{\pi}}$ denotes that the expectation is taken with respect to the policy $\boldsymbol{\pi}$. Frequently we will drop the subscript $\boldsymbol{\pi}$ in the expectation to simplify notation going forward. Thus $\mathbb{E}$ will mean expectation with respect to the policy unless specified otherwise, and so we can write

  $$V_t^{\boldsymbol{\pi}}(s) = \mathbb{E}[G_t | s_t = s] \ . \tag{22}$$

  Notice that when the horizon H is infinite, this definition (22) together with the stationary assumptions of the rewards, transition probabilities and policy imply that for all $s \in S$, $V_i^{\boldsymbol{\pi}}(s) = V_j^{\boldsymbol{\pi}}(s)$ for all $i, j = 0, 1, \ldots$, and thus in this case we will define in a manner analogous to the case of a Markov reward process:

  $$V^{\pi}(s) = V_0^{\boldsymbol{\pi}}(s) \ . \tag{23}$$

- **State-action value function** : The state-action value function $Q_t^{\boldsymbol{\pi}}(s, a)$ for a state s and action a is defined as the expected return starting from the state $s_t = s$ at time t and taking the action $a_t = a$, and then subsequently following the policy $\boldsymbol{\pi}$. It is written mathematically as

  $$Q_t^{\boldsymbol{\pi}}(s, a) = \mathbb{E}[G_t | s_t = s, a_t = a] \ . \tag{24}$$

  In the infinite horizon case, similar to the state value function, the stationary assumptions about the rewards, transition probabilities and policy imply that for all $s \in S$ and $a \in A$, $Q_i^{\boldsymbol{\pi}}(s, a) = Q_j^{\boldsymbol{\pi}}(s, a)$ for all $i, j = 0, 1, \ldots$, which motivates the following definition

  $$Q^{\pi}(s, a) = Q_0^{\boldsymbol{\pi}}(s, a) \ . \tag{25}$$

Exercise 3.13. Consider a stationary policy $\boldsymbol{\pi} = (\pi, \pi, \dots)$. If the assumptions of stationary transition probabilities and stationary rewards hold, and if the horizon H is infinite, then using the definitions in (22) and (24) prove that for all $s \in S$ and $a \in A$, (a) $V_i^{\boldsymbol{\pi}}(s) = V_j^{\boldsymbol{\pi}}(s)$, and (b) $Q_i^{\boldsymbol{\pi}}(s,a) = Q_j^{\boldsymbol{\pi}}(s,a)$ for all $i, j = 0, 1, \dots$.

In the infinite horizon case, the assumptions about stationary transition probabilities and rewards lead to the following important identity connecting the state value function and the state-action value function for a stationary policy π :

$$\begin{aligned} Q^\pi(s,a) &\overset{(a)}{=} Q_0^\pi(s,a) = \mathbb{E}[G_0 | s_0 = s, a_0 = a] = \mathbb{E}\left[\sum_{i=0}^{\infty} \gamma^i r_i \middle| s_0 = s, a_0 = a\right] \\ &= \mathbb{E}[r_0 | s_0 = s, a_0 = a] + \sum_{i=1}^{\infty} \gamma^i \mathbb{E}[r_i | s_0 = s, a_0 = a] \\ &\overset{(b)}{=} R(s,a) + \sum_{i=1}^{\infty} \gamma^i \left(\sum_{s' \in S} P(s_1 = s' | s_0 = s, a_0 = a) \mathbb{E}[r_i | s_0 = s, a_0 = a, s_1 = s'] \right) \\ &\overset{(c)}{=} R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) \left(\sum_{i=1}^{\infty} \gamma^{i-1} \mathbb{E}[r_i | s_1 = s'] \right) \\ &\overset{(d)}{=} R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^\pi(s') \ , \end{aligned} \tag{26}$$

for all $s \in S$, $a \in A$, where (a) follows from (24) and (25), (b) is due to the law of total expectation, (c) follows from the Markov property, and (d) follows from Exercise 3.13 and linearity of expectation.

Exercise 3.14. Consider a policy $\boldsymbol{\pi}$, not necessarily stationary. (a) Prove that in this case the analog of equation (26) is given by $Q_t^{\boldsymbol{\pi}}(s,a) = R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V_{t+1}^{\boldsymbol{\pi}}(s')$, for all $s \in S$, $a \in A$ and for all $t = 0, 1, \dots$.

An interesting aspect of specifying a stationary policy π on a MDP is that evaluating the value function for the policy is equivalent to evaluating the value function on an equivalent Markov reward process. Specifically we define the Markov reward process $M'(S, \mathbf{P}^\pi, R^\pi, \gamma)$, where $\mathbf{P}^\pi$ and R^π are given by:

$$\begin{aligned} R^\pi(s) &= \sum_{a \in A} \pi(a|s) R(s,a) \ , \\ P^\pi(s'|s) &= \sum_{a \in A} \pi(a|s) P(s'|s,a) \ . \end{aligned} \tag{27}$$

Exercise 3.15. Consider a stationary policy π for a MDP. (a) Prove that the value function of the policy V^π satisfies the identity $V^\pi(s) = R^\pi(s) + \gamma \sum_{s' \in S} P^\pi(s'|s) V^\pi(s')$ for all states $s \in S$, with R^π and $\mathbf{P}^\pi$ defined by (27).

The evaluation of the value function corresponding to the policy can then be carried out using the techniques introduced in the context of Markov reward processes. For example, in the infinite horizon case with $\gamma < 1$, the iterative algorithm to calculate the value function corresponding to a stationary policy π is given in algorithm 4. The algorithm takes as input a Markov decision process $M = (S, A, P, R, \gamma)$, a stationary policy π, and a tolerance ϵ, and computes the value function for all the states.

Exercise 3.16. (a) Prove that when $\gamma < 1$, algorithm 4 always converges. *Hint: Use Theorem 3.1.* (b) Consider a positive sequence of real numbers $\{\epsilon_i\}_{i \geq 1}$ such that $\epsilon_i \to 0$. Suppose algorithm 4 is run to termination for each ϵ_i, and denote each corresponding output of the algorithm as V_i^π. Prove that the sequence $V_i^\pi \to V^\pi$, where V^π is the value of the policy.

Algorithm 4 Iterative algorithm to calculate MDP value function for a stationary policy π

procedure POLICY EVALUATION(M, π, ϵ)
  For all states $s \in S$, define $R^\pi(s) = \sum_{a \in A} \pi(a|s) R(s,a)$
  For all states $s, s' \in S$, define $P^\pi(s'|s) = \sum_{a \in A} \pi(a|s) P(s'|s,a)$
  For all states $s \in S$, $V'(s) \leftarrow 0$, $V(s) \leftarrow \infty$
  while $||V - V'||_\infty > \epsilon$ **do**
    $V \leftarrow V'$
    For all states $s \in S$, $V'(s) = R^\pi(s) + \gamma \sum_{s' \in S} P^\pi(s'|s) V(s')$
  return $V'(s)$ for all $s \in S$

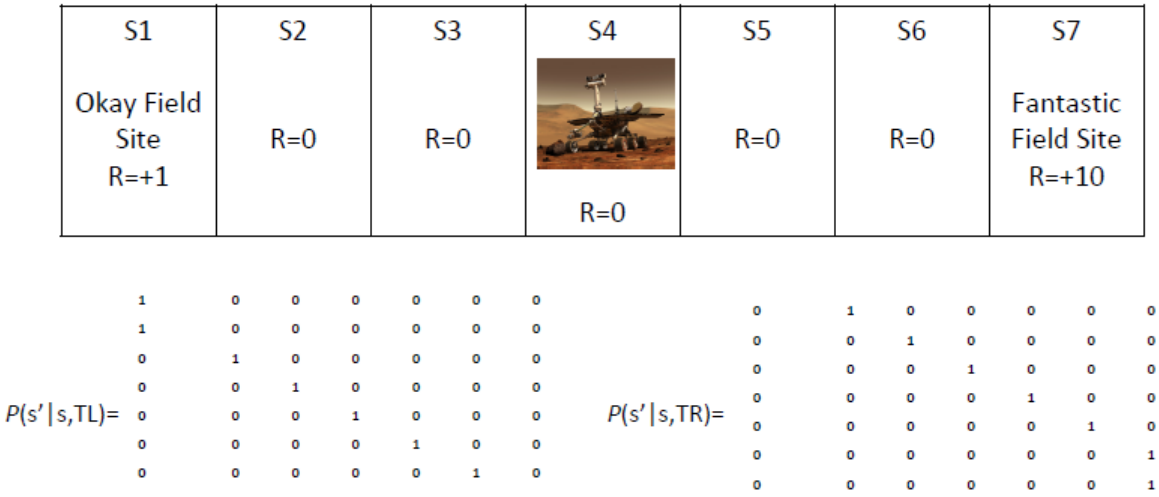


Figure 3: Mars Rover Markov decision process.

3.4.2 Example of a Markov decision process : Mars Rover

As an example of a MDP, consider the example given in Figure 3. The agent is again a Mars rover whose state space is given by $S = \{S1,\ S2,\ S3,\ S4,\ S5,\ S6,\ S7\}$. The agent has two actions in each state called *"try left"* and *"try right"*, and so the action space is given by $A = \{TL, TR\}$. Taking an action always succeeds, unless we hit an edge in which case we stay in the same state. This leads to the two transition probability matrices for each of the two actions as shown in Figure 3. The rewards from each state are the same for all actions, and is 0 in the states $\{S2,\ S3,\ S4,\ S5,\ S6\}$, while for the states $S1,\ S7$ the rewards are 1, 10 respectively. The discount factor for this MDP is some $\gamma \in [0, 1]$.

Exercise 3.17. Consider the MDP discussed above in Figure 3. Let $\gamma = 0$, and consider a stationary policy π which always involves taking the action TL from any state. (a) Calculate the value function of the policy for all states if the horizon is finite. (b) Calculate the value function of the policy when the horizon is infinite. *Hint: Use Theorem A.3.*

3.5 Bellman backup operators

In this section, we introduce the concept of the Bellman backup operators and prove some of their properties which will turn out to be extremely useful in the next section when we discuss MDP control. We have already encountered one Bellman backup operator in (14), (15) in the proof of Theorem 3.1. We will now define two other closely related (but not same!) Bellman backup operators : *the Bellman*

expectation backup operator and *the Bellman optimality backup operator.*

3.5.1 Bellman expectation backup operator

Suppose we are given a MDP $M = (S, A, P, R, \gamma)$, and a stationary policy π which can be deterministic or stochastic. We have already seen in section 3.4.1 that this is equivalent to a MRP $M' = (S, P^\pi, R^\pi, \gamma)$, where P^π and R^π are defined in (27). The value function of policy π evaluated on M, and denoted by V^π, is the same as the value function evaluated on M', where we have used the corresponding definitions of the value function for a MDP and MRP respectively. Note that V^π lives in the finite dimensional Banach space $\mathbb{R}^{|S|}$, which we will equip with the infinity norm $||\cdot||_\infty$ introduced in Exercise 3.2.

Then for element $U \in \mathbb{R}^{|S|}$ the Bellman expectation backup operator B^π for the policy π is defined as

$$(B^\pi U)(s) = R^\pi(s) + \gamma \sum_{s' \in S} P^\pi(s'|s) U(s') \quad , \quad \forall\, s \in S \,. \tag{28}$$

We should note that we have already seen this operator appear once before in algorithm 4. We now prove some properties of this operator.

Theorem 3.2. *The operator B^π defined in (28) is a contraction map. If $\gamma < 1$ then it is a strict contraction and has a unique fixed point.*

Proof. Consider $U_1, U_2 \in \mathbb{R}^{|S|}$. Then for a state $s \in S$, we have from (28) and triangle inequality

$$\begin{aligned} |(B^\pi U_1)(s) - (B^\pi U_2)(s)| &= \gamma \left| \sum_{s' \in S} P^\pi(s'|s)(U_1(s') - U_2(s')) \right| \leq \gamma \sum_{s' \in S} P^\pi(s'|s) |U_1(s') - U_2(s')| \\ &\leq \gamma \sum_{s' \in S} P^\pi(s'|s) \max_{s'' \in S} |U_1(s'') - U_2(s'')| = \gamma \sum_{s' \in S} P^\pi(s'|s)\, ||U_1 - U_2||_\infty \\ &= \gamma\, ||U_1 - U_2||_\infty \quad . \end{aligned} \tag{29}$$

As (29) is true for every $s \in S$ we conclude that $||B^\pi U_1 - B^\pi U_2||_\infty \leq \gamma\, ||U_1 - U_2||_\infty$, and hence B^π is a contraction map as $\gamma \in [0, 1]$.

Considering $\gamma < 1$ in (29), we conclude that in this case B^π is a strict contraction, and hence by applying Theorem A.5 it has a unique fixed point. □

Corollary 3.2.1. *Let $\gamma < 1$. Then for any $U \in \mathbb{R}^{|S|}$ the sequence $\{(B^\pi)^k U\}_{k \geq 0}$ is a Cauchy sequence and converges to the fixed point of B^π.*

Proof. The proof follows directly by applying Theorem 3.2, followed by Theorem A.4 and the contraction mapping theorem (TheoremA.5). □

This also implies that for a stationary policy π, the value function of the policy V^π is a fixed point of B^π, as shown by the following corollary.

Corollary 3.2.2. *Let π be a policy for an infinite horizon MDP with $\gamma < 1$. Then the value function of the policy V^π is a fixed point of B^π.*

Proof. The fact that $(B^\pi V^\pi)(s) = V^\pi(s)$ for all states $s \in S$, follows from the definition (28) of B^π and Exercise 3.15. □

The next theorem proves the *"monotonicity"* property of the Bellman expectation backup operator.

Theorem 3.3. *Suppose we have $U_1, U_2 \in \mathbb{R}^{|S|}$ such that for all $s \in S$, $U_1(s) \geq U_2(s)$. Then for every stationary policy π, we have $(B^\pi U_1)(s) \geq (B^\pi U_2)(s)$ for all $s \in S$. If instead the inequality is strict, i.e. $U_1(s) > U_2(s)$ for all $s \in S$, then we have $(B^\pi U_1)(s) > (B^\pi U_2)(s)$ for all $s \in S$.*

Proof. When $U_1(s) \geq U_2(s)$ for all $s \in S$, using definition (28) of B^π we obtain,

$$(B^\pi U_1)(s) - (B^\pi U_2)(s) = \sum_{s' \in S} P^\pi(s'|s)(U_1(s') - U_2(s')) \geq 0 \ , \tag{30}$$

ans when $U_1(s) > U_2(s)$ for all $s \in S$, the same steps give $(B^\pi U_1)(s) - (B^\pi U_2)(s) > 0$, for all states $s \in S$. □

3.5.2 Bellman optimality backup operator

Suppose we are now given a MDP $M = (S, A, P, R, \gamma)$. We again consider the finite dimensional Banach space $\mathbb{R}^{|S|}$ equipped with the infinity norm $||\cdot||_\infty$. Then for every element $U \in \mathbb{R}^{|S|}$ the Bellman optimality backup operator B^* is defined as

$$(B^* U)(s) = \max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) U(s') \right] \ , \ \forall \ s \in S \ . \tag{31}$$

We next prove analogous properties for this operator which are similar to the ones for the Bellman expectation backup operator.

Theorem 3.4. *For every $U_1, U_2 \in \mathbb{R}^{|S|}$, and for all states $s \in S$ the following inequalities are true:*

(a)

$$\begin{aligned} (B^* U_1)(s) - (B^* U_2)(s) &\leq \gamma \max_{a \in A} \left[\sum_{s' \in S} P(s'|s,a) \ (U_1(s') - U_2(s')) \right] \\ &\leq \gamma \max_{a \in A} \left[\sum_{s' \in S} P(s'|s,a) \ |U_1(s') - U_2(s')| \right] \ , \end{aligned} \tag{32}$$

(b)

$$|(B^* U_1)(s) - (B^* U_2)(s)| \leq \gamma \max_{a \in A} \left[\sum_{s' \in S} P(s'|s,a) \ |U_1(s') - U_2(s')| \right] \leq \gamma \ ||U_1 - U_2||_\infty \ . \tag{33}$$

Proof. We first prove part *(a)*. Fix a state $s \in S$. Using (31) and as the action space A is finite, we conclude that there exists $a_1, a_2 \in A$, not necessarily different, such that the following holds:

$$\begin{aligned} (B^* U_1)(s) &= R(s,a_1) + \gamma \sum_{s' \in S} P(s'|s,a_1) U_1(s') \ , \\ (B^* U_2)(s) &= R(s,a_2) + \gamma \sum_{s' \in S} P(s'|s,a_2) U_2(s') \ . \end{aligned} \tag{34}$$

Then by the definition of maximum in (31), we also have for the action a_1 that

$$(B^* U_2)(s) \geq R(s,a_1) + \gamma \sum_{s' \in S} P(s'|s,a_1) U_2(s') \ . \tag{35}$$

Thus from (34) and (35) we deduce the following

$$\begin{aligned}(B^*U_1)(s) - (B^*U_2)(s) &\leq \gamma \sum_{s' \in S} P(s'|s, a_1)\ (U_1(s') - U_2(s')) \\ &\leq \gamma \max_{a \in A} \left[\sum_{s' \in S} P(s'|s, a)\ (U_1(s') - U_2(s')) \right] \quad ,\end{aligned} \tag{36}$$

which proves the first inequality of *(a)*. For the second inequality notice that we have for all states $s' \in S$, $U_1(s') - U_2(s') \leq |U_1(s') - U_2(s')|$, and so multiplying each of these inequalities by positive numbers $P(s'|s, a)$ for some $a \in A$, and summing over all s' gives

$$\sum_{s' \in S} P(s'|s, a)\ (U_1(s') - U_2(s')) \leq \sum_{s' \in S} P(s'|s, a)\ |(U_1(s') - U_2(s'))| \quad . \tag{37}$$

The result is proved by taking the max over all $a \in A$, by using monotonicity of the max function.

To prove part *(b)*, notice that by interchanging the roles of U_1, U_2, we have from part *(a)*

$$(B^*U_2)(s) - (B^*U_1)(s) \leq \gamma \max_{a \in A} \left[\sum_{s' \in S} P(s'|s, a)\ |U_1(s') - U_2(s')| \right] \quad , \tag{38}$$

and thus combining (38) and (32) we obtain

$$\begin{aligned}|(B^*U_1)(s) - (B^*U_2)(s)| &\leq \gamma \max_{a \in A} \left[\sum_{s' \in S} P(s'|s, a)\ |U_1(s') - U_2(s')| \right] \\ &\leq \gamma \max_{a \in A} \left[\sum_{s' \in S} P(s'|s, a) \max_{s'' \in S} |U_1(s'') - U_2(s'')| \right] \\ &= \gamma \max_{a \in A} \left[\sum_{s' \in S} P(s'|s, a)\ ||U_1 - U_2||_\infty \right] \\ &= \gamma \max_{a \in A} ||U_1 - U_2||_\infty = \gamma\ ||U_1 - U_2||_\infty \quad ,\end{aligned} \tag{39}$$

which proves *(b)*. □

Theorem 3.5. *The operator B^* defined in (31) is a contraction map. If $\gamma < 1$ then it is a strict contraction and has a unique fixed point.*

Proof. The fact that B^* is a contraction follows from Theorem 3.4 by observing that (32) is true for all $s \in S$, and so must be true in particular for $\arg\max_{s \in S} |(B^*U_1)(s) - (B^*U_2)(s)|$, for every $U_1, U_2 \in \mathbb{R}^{|S|}$. Thus $||B^*U_1 - B^*U_2||_\infty \leq \gamma\ ||U_1 - U_2||_\infty$, proving that B^* is a contraction map as $\gamma \in [0, 1]$. Setting $\gamma < 1$ in this inequality proves that B^* is a strict contraction for $\gamma \in [0, 1)$ and thus has a unique fixed point by Theorem A.5. □

Corollary 3.5.1. *Let $\gamma < 1$. Then for any $U \in \mathbb{R}^{|S|}$ the sequence $\{(B^*)^k U\}_{k \geq 0}$ is a Cauchy sequence and converges to the fixed point of B^*.*

Proof. The proof follows directly by applying Theorem 3.5, followed by Theorem A.4 and the contraction mapping theorem (TheoremA.5). □

The next theorem compares the result of the application of B^π versus B^* to some $U \in \mathbb{R}^{|S|}$.

Theorem 3.6. *For every stationary policy π, for every $U \in \mathbb{R}^{|S|}$ and for all $s \in S$, $(B^*U)(s) \geq (B^\pi U)(s)$.*

Proof. Fix a stationary policy π, and let B^π be the corresponding Bellman expectation backup operator. Fix some $U \in \mathbb{R}^{|S|}$. Let us also fix some $s \in S$. Then from definition (31) of B^* we have

$$(B^*U)(s) = \max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a)U(s') \right] \geq R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a)U(s') \ , \ \forall \ a \in A \ . \quad (40)$$

Multiplying (40) by $\pi(a|s)$ and summing over all $a \in A$ gives

$$\begin{aligned} (B^*U)(s) &= \sum_{a \in A} \pi(a|s)(B^*U)(s) \geq \sum_{a \in A} \pi(a|s) \left[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a)U(s') \right] \\ &= \sum_{a \in A} \pi(a|s)R(s,a) + \gamma \sum_{s' \in S} \left(\sum_{a \in A} \pi(a|s)P(s'|s,a) \right) U(s') \\ &= R^\pi(s) + \gamma \sum_{s' \in S} P^\pi(s'|s)U(s') = (B^\pi U)(s) \ , \end{aligned} \quad (41)$$

where the last equality follows from definitions (27) and (28) of R^π, P^π and B^π, thus proving the theorem. □

3.6 MDP control in the infinite horizon setting

We now have all the background necessary to discuss the problem of *"MDP control"*, where we seek to find the best policy (often a policy), that achieves the greatest value function among the set of all possible policies. In the context of reinforcement learning, this is precisely the objective of the agent. We are going to first discuss the infinite horizon case in this section, and the finite horizon case will be mentioned in the next section. We do it this way because the infinite horizon case is a much harder problem, that presents quite a few mathematical challenges which will need to be resolved.

To get started, we need to address the question *"what do we exactly mean by finding an optimal policy ?"*. Precisely we want to know whether a policy always exists, which we will denote by $\boldsymbol{\pi}^*$, whose value function is at least as good as the value function of any other policy. In other words, we need to ensure that the supremum of the value function is actually attained for some policy ! To appreciate the subtlety of this point, consider the example of maximizing the function $f : \mathbb{R} \to \mathbb{R}$ on $(0,1)$ defined as $f(x) = x$, and note that this problem does not have a solution. But $\sup f(x) = 1$, although $\nexists \ x \in (0,1)$ for which this is attained.

We first define precisely what it means for a policy, not necessarily stationary, to be an **optimal policy**.

Definition 3.1. A policy $\boldsymbol{\pi}^*$ is an *optimal policy* iff for every policy $\boldsymbol{\pi}$, for all $t = 0, 1, \ldots$, and for all states $s \in S$, $V_t^{\boldsymbol{\pi}^*}(s) \geq V_t^{\boldsymbol{\pi}}(s)$.

The next result that we leave for the reader to prove states that for an infinite horizon MDP, existence of an optimal policy also implies the existence of a stationary optimal policy. This result is intuitively obvious, and is a very important result as it significantly reduces the universe of policies to consider when searching for an optimal policy, if it exists. In particular, it states that we need only consider policies that are stationary.

Exercise 3.18. (a) Consider an infinite horizon MDP. Let $\boldsymbol{\pi}^*$ be an optimal policy for the MDP. Prove that there exists a stationary policy π, that is $\boldsymbol{\pi} = (\pi, \pi, \ldots)$, which is also optimal.

The next two theorems improve on the conclusion of Exercise 3.18 and show us that we may restrict the search to a finite set of deterministic stationary policies.

Theorem 3.7. *The number of deterministic stationary policies is finite, and equals* $|A|^{|S|}$.

Proof. Since the policies are stationary and deterministic, each policy can be represented as a function $\pi : S \to A$. The number of such distinct functions is given by $|A|^{|S|}$. This also proves that the set of deterministic stationary policies is finite. □

Theorem 3.8. *If* π *is a stationary policy for an infinite horizon MDP with* $\gamma < 1$*, then there exists a deterministic stationary policy* $\hat{\pi}$ *such that* $V^{\hat{\pi}}(s) \geq V^{\pi}(s)$ *for all states* $s \in S$*. One such policy is given by the stationary policy*

$$\hat{\pi}(s) = \arg\max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^{\pi}(s') \right] \ , \ \forall \ s \in S \ , \tag{42}$$

which satisfies the equality $(B^{\hat{\pi}} V^{\pi})(s) = (B^* V^{\pi})(s) \geq V^{\pi}(s)$ *for all* s*. Moreover* $V^{\hat{\pi}}(s) = V^{\pi}(s)$ *for all* s*, iff* $(B^* V^{\pi})(s) = V^{\pi}(s)$ *for all* s*.*

Proof. We first notice that the policy $\hat{\pi}$ defined in (42) is a stationary policy (by definition), and is also deterministic for every $s \in S$, by the definition of arg max with ties broken randomly.

As $\hat{\pi}$ is deterministic, we can conclude using (27) that $R^{\hat{\pi}}(s) = R(s, \hat{\pi}(s))$ and $P^{\hat{\pi}}(s'|s) = P(s'|s, \hat{\pi}(s))$ for all $s \in S$ and $a \in A$, and thus we have

$$(B^{\hat{\pi}} V^{\pi})(s) = R(s, \hat{\pi}(s)) + \gamma \sum_{s' \in S} P(s'|s, \hat{\pi}(s)) V^{\pi}(s) = (B^* V^{\pi})(s) \ , \tag{43}$$

for all states $s \in S$ using (42), and the definitions of the Bellman backup operators in (28) and (31). Next, by Corollary 3.2.2 we have $B^{\pi} V^{\pi} = V^{\pi}$, and by Theorem 3.6 we have $B^* V^{\pi} \geq B^{\pi} V^{\pi}$, and so combining these with (43) we obtain

$$(B^{\hat{\pi}} V^{\pi})(s) = (B^* V^{\pi})(s) \geq V^{\pi}(s) \ , \ \forall \ s \in S \ . \tag{44}$$

Next using Theorem 3.3, the monotonicity property of $B^{\hat{\pi}}$ allows us to conclude by repeatedly applying $B^{\hat{\pi}}$ to both sides of (44) that $((B^{\hat{\pi}})^k V^{\pi})(s) \geq V^{\pi}(s)$ for all $k \geq 1$, and for all states $s \in S$. Then using Corollary 3.2.1, and noticing that $V^{\hat{\pi}}$ is the unique fixed point of $B^{\hat{\pi}}$ we obtain by taking limits

$$V^{\hat{\pi}}(s) = (B^{\hat{\pi}} V^{\hat{\pi}})(s) = \lim_{k \to \infty} ((B^{\hat{\pi}})^k V^{\pi})(s) \geq V^{\pi}(s) \ , \ \forall \ s \in S \ . \tag{45}$$

To prove the second part of the theorem, first assume that $B^* V^{\pi} = V^{\pi}$. Then by (44) we have $B^{\hat{\pi}} V^{\pi} = B^* V^{\pi} = V^{\pi}$, and so by uniqueness of the fixed point of $B^{\hat{\pi}}$ we get $V^{\hat{\pi}} = B^{\hat{\pi}} V^{\hat{\pi}} = V^{\pi}$. Next assume that $V^{\hat{\pi}} = V^{\pi}$. Then again by (44) we have $V^{\pi} = V^{\hat{\pi}} = B^{\hat{\pi}} V^{\hat{\pi}} = B^{\hat{\pi}} V^{\pi} = B^* V^{\pi} \geq V^{\pi}$, implying that $B^* V^{\pi} = V^{\pi}$, thus completing the proof. □

Corollary 3.8.1. *In the notation of Theorem 3.8, if* $\exists \ s \in S$ *such that* $(B^* V^{\pi})(s) > V^{\pi}(s)$*, then* $V^{\hat{\pi}}(s) > V^{\pi}(s)$*. In this case, we say that* $\hat{\pi}$ *is "strictly better" than* π *as a policy.*

Proof. The proof follows immediately by noting that the inequality in (44) becomes a strict inequality, and then applying Theorem 3.3. □

The consequences of Theorems 3.7 and 3.8 is spectacular, because now the search for an optimal policy has been reduced to the set of only the deterministic stationary policies which is a finite set, if such a policy exists. The reader is to prove that this is actually the case in the following exercise.

Exercise 3.19. Consider an infinite horizon MDP with $\gamma < 1$. Denote Π to be the set of all deterministic stationary policies. (a) Prove that $\exists \ \pi^* \in \Pi$, such that for all $\pi \in \Pi$, and for all states $s \in S$, $V^{\pi^*}(s) \geq V^{\pi}(s)$. (b) Conclude that $\boldsymbol{\pi^*} = (\pi^*, \pi^*, \dots)$ is an optimal policy. *Hint : See Theorem 3.10.*

We have thus established the existence of an optimal policy and moreover concluded that a deterministic stationary policy suffices. This then allows us to make the following definition:

Definition 3.2. The *optimal value function* for an infinite horizon MDP is defined as

$$V^*(s) = \max_{\pi \in \Pi} V^\pi(s) \ , \tag{46}$$

and there exists a stationary deterministic policy $\pi^* \in \Pi$, which is an optimal policy, such that $V^*(s) = V^{\pi^*}(s)$ for all states $s \in S$, where Π is the set of all stationary deterministic policies.

We next look at a few algorithms to compute the optimal value function and an optimal policy.

3.6.1 Policy search

Definition 3.2 immediately renders itself to a brute force algorithm called **policy search** to find the optimal value function V^* and an optimal policy π^*, as described in pseudo-code in algorithm 5. The algorithm takes as input an infinite horizon MDP $M = (S, A, P, R, \gamma)$ and a tolerance ϵ for accuracy of policy evaluation, and returns the optimal value function and an optimal policy.

Algorithm 5 Policy search algorithm to calculate optimal value function and find an optimal policy

```
procedure POLICY SEARCH(M, ε)
    Π ← All stationary deterministic policies of M
    π* ← Randomly choose a policy π ∈ Π
    V* ← POLICY EVALUATION (M, π*, ε)
    for π ∈ Π do
        V^π ← POLICY EVALUATION (M, π, ε)
        if V^π(s) ≥ V*(s) for all s ∈ S, then
            V* ← V^π
            π* ← π
    return V*(s), π*(s) for all s ∈ S
```

It is clear that algorithm 5 always terminates as it checks all $|A|^{|S|}$ deterministic stationary policies. Thus the run-time complexity of this algorithm is $O(|A|^{|S|})$. It is possible to prove correctness of the algorithm when $\epsilon = 0$, i.e. when in each iteration the policy evaluation is done exactly. In practice ϵ is set to a small number such as 10^{-9} to 10^{-12}.

Theorem 3.9. *Algorithm 5 returns the optimal value function and an optimal policy when $\epsilon = 0$.*

Proof. Let π^* be an optimal policy, and thus $V^{\pi^*}(s) = V^*(s)$ for all states $s \in S$. Since the algorithm checks every policy in Π, it means that π^* must get selected at some iteration of the algorithm. Thus for the policies considered in future iterations the value function can no longer strictly increase. Future iterations may select a different policy with the same optimal value function, thus completing the proof. □

Exercise 3.20. Consider the MDP discussed in section 3.4.2, shown in Figure 3. Consider the horizon to be infinite. (a) How many deterministic stationary policies does the agent have ? (b) If $\gamma < 1$, is the optimal policy unique ? (c) If $\gamma = 1$, is the optimal policy unique ?

3.6.2 Policy iteration

We now discuss a more efficient algorithm than policy search called **policy iteration**. The algorithm is a straightforward application of Theorem 3.8, which states that given any stationary policy π, we can find a deterministic stationary policy that is no worse than the existing policy. In particular the

Algorithm 6 Policy improvement algorithm to improve an input policy

procedure POLICY IMPROVEMENT(M, V^π)
  $\hat{\pi}(s) \leftarrow \arg\max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^\pi(s') \right] \ , \ \forall \ s \in S$
  return $\hat{\pi}(s)$ for all $s \in S$

theorem also applies to deterministic policies. This simple step has a special name called **"policy improvement"**, whose pseudo-code is presented in algorithm 6.

The output of algorithm 6 is always guaranteed to be at least as good as the policy π corresponding to the input value function V^π, and represents a *"greedy"* attempt to improve the policy. When performed iteratively with the policy evaluation algorithm (algorithm 4), this gives rise to the policy iteration algorithm. The pseudo-code of policy iteration is outlined in algorithm 7.

Algorithm 7 Policy iteration algorithm to calculate optimal value function and find an optimal policy

procedure POLICY ITERATION(M, ϵ)
  $\pi \leftarrow$ Randomly choose a policy $\pi \in \Pi$
  while true **do**
    $V^\pi \leftarrow$ POLICY EVALUATION (M, π, ϵ)
    $\pi^* \leftarrow$ POLICY IMPROVEMENT (M, V^π)
    if $\pi^*(s) = \pi(s)$ **then**
      break
    else
      $\pi \leftarrow \pi^*$
  $V^* \leftarrow V^\pi$
  return $V^*(s)$, $\pi^*(s)$ for all $s \in S$

The proof of correctness of algorithm 7 is left to the reader as the next exercise. Note that the algorithm will always terminate as there are a finite number of stationary deterministic policies by Theorem 3.7.

Exercise 3.21. Consider an infinite horizon MDP with $\gamma < 1$. (a) Show that when algorithm 7 is run with $\epsilon = 0$, it finds the optimal value function and an optimal policy. *Hint : See Theorem 3.10.* (b) Prove that the termination criteria used in the algorithm makes sense: precisely show that if the policy does not change during a policy improvement step, then the policy cannot improve in future iterations. (c) Show that the value functions corresponding to the policies in each iteration of the algorithm form a non-decreasing sequence for every $s \in S$. (d) What is the worst case run-time complexity of this algorithm ?

3.6.3 Value iteration

We now discuss **value iteration** which is yet another technique that can be used to compute the optimal value function and an optimal policy, given a MDP. To motivate this method we will need the following theorem:

Theorem 3.10. *For a MDP with $\gamma < 1$, let the fixed point of the Bellman optimality backup operator B^* be denoted by $V^* \in \mathbb{R}^{|S|}$. Then the policy given by*

$$\pi^*(s) = \arg\max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^*(s') \right] \ , \ \forall \ s \in S \ , \tag{47}$$

is a stationary deterministic policy. The value function of this policy V^{π^*} satisfies the identity $V^{\pi^*} = V^*$, and thus V^* is also the fixed point of the operator B^{π^*}. In particular this implies that there exists a stationary deterministic policy π^* whose value function is the fixed point of B^*. Moreover, π^* is an optimal policy.

Proof. We start by noting that π^* as defined in (47) is a stationary deterministic policy, and so we can conclude using (27) that $R^{\pi^*}(s) = R(s, \pi^*(s))$ and $P^{\pi^*}(s'|s) = P(s'|s, \pi^*(s))$ for all $s \in S$ and $a \in A$.

As V^* is the fixed point of B^*, we have $B^*V^* = V^*$. So using definition (31) of B^*, and (47) we can write

$$\begin{aligned} V^*(s) &= \max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^*(s') \right] \\ &= R(s, \pi^*(s)) + \gamma \sum_{s' \in S} P(s'|s, \pi^*(s)) V^*(s') \\ &= R^{\pi^*}(s) + \gamma \sum_{s' \in S} P^{\pi^*}(s'|s) V^*(s') \\ &= V^{\pi^*}(s) \end{aligned} \tag{48}$$

for all $s \in S$, completing the proof of the first part of the theorem.

To prove that π^* is an optimal policy, we show that if an optimal policy exists then its value function must be a fixed point of the operator B^*. So assume that an optimal policy exists, which by Theorem 3.8 we can take to be a stationary deterministic policy, and let us denote it as μ and the corresponding optimal value function as V^μ. Now for the sake of contradiction, suppose V^μ is not a fixed point of B^*. Then there exists $s \in S$ such that $V^\mu(s) \neq (B^*V^\mu)(s)$, which upon combining with Theorem 3.8 implies that $V^\mu(s) > (B^*V^\mu)(s)$. Then application of Corollary 3.8.1 implies that there exists a policy $\hat{\pi}$ which is strictly better than μ, and so we have a contradiction. This proves that V^μ must be the unique fixed point of B^*. Combining this fact with the first part implies that V^* must be the optimal value function and π^* is an optimal policy. This completes the proof. □

Theorem 3.10 suggests a straightforward way to calculate the optimal value function V^* and an optimal policy π^*. The idea is to run fixed point iterations to find the fixed point of B^* using Corollary 3.5.1. Once we have V^*, an optimal policy π^* can be extracted using (47). The pseudo-code of this algorithm is given in algorithm 8, which takes as input an infinite horizon MDP $M = (S, A, P, R, \gamma)$ and a tolerance ϵ, and returns the optimal value function and an optimal policy.

Algorithm 8 Value iteration algorithm to calculate optimal value function and find an optimal policy

procedure VALUE ITERATION(M, ϵ)
  For all states $s \in S$, $V'(s) \leftarrow 0$, $V(s) \leftarrow \infty$
  while $||V - V'||_\infty > \epsilon$ **do**
    $V \leftarrow V'$
    For all states $s \in S$, $V'(s) = \max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V(s') \right]$
  $V^* \leftarrow V$ for all $s \in S$
  $\pi^* \leftarrow \arg\max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^*(s') \right] \quad , \forall\, s \in S$
  return $V^*(s)$, $\pi^*(s)$ for all $s \in S$

If algorithm 8 is run with $\epsilon = 0$, we can recover the optimal value function and an optimal policy exactly. However in practice, ϵ is set to be a small number such as 10^{-9}-10^{-12}.

3.7 MDP control for a finite horizon MDP

We now briefly discuss the MDP control problem for a finite horizon MDP. Having already discussed the control problem for infinite horizon MDPs, we simply state that in the finite horizon case, a deterministic policy can be obtained that is optimal. But the policy is no longer stationary, and so at each time t the policy is different. The proof is not too difficult and the reader is asked to derive these facts in the following exercise.

Exercise 3.22. Consider a MDP with finite horizon H and finite rewards. A typical episode of the MDP will look like $(s_0, a_0, s_1, a_1, \ldots, s_{H-1}, a_{H-1}, s_H)$. Let a policy for the MDP be denoted by $\boldsymbol{\pi} = (\pi_0, \pi_1, \ldots, \pi_{H-1})$. Then prove the following statements:

(a) Show that the number of deterministic policies for the MDP is given by $H|A|^{|S|}$.

(b) Assuming that an optimal policy $\boldsymbol{\pi^*}$ exists, derive a recurrence relation for the optimal value function $V^{\boldsymbol{\pi}^*} = (V_0^{\boldsymbol{\pi}^*}, \ldots, V_H^{\boldsymbol{\pi}^*})$, with $V_H^{\boldsymbol{\pi}^*}(s) = 0$ for all states $s \in S$. Precisely, derive a relationship between $V_t^{\boldsymbol{\pi}^*}$ and $V_{t+1}^{\boldsymbol{\pi}^*}$.

(c) Let $\boldsymbol{\Pi}$ be the set of all deterministic policies, i.e. for every $\boldsymbol{\pi} \in \boldsymbol{\Pi}$, π_t is a deterministic policy at time t and for all times $t = 0, \ldots, H-1$. Show that for every policy, deterministic or stochastic, there exists a $\boldsymbol{\pi} \in \boldsymbol{\Pi}$ which is no worse.

(b) Show that $\boldsymbol{\Pi}$ contains a policy that is optimal.

Because of the conclusion of Exercise 3.22, just like in the infinite horizon case we can restrict our search for an optimal policy to the set of deterministic policies. We present an algorithm, namely **value iteration** for this purpose, which is analogous to its counterpart in the infinite horizon case.

Algorithm 9 Value iteration algorithm for finite horizon MDPs

procedure FINITE VALUE ITERATION(M)
  For all states $s \in S$, $V_H^*(s) \leftarrow 0$
  $t \leftarrow H - 1$
  while $t \geq 0$ **do**
    For all states $s \in S$, $V_t^*(s) = \max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V_{t+1}^*(s')\right]$
    For all states $s \in S$, $\pi_t^*(s) = \arg\max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V_{t+1}^*(s')\right]$
    $t \leftarrow t - 1$
  return For all states $s \in S$, $V_t^*(s)$ for $t = 0, \ldots, H$, $\pi_t^*(s)$ for $t = 0, \ldots, H-1$

The proof of correctness of the algorithm is left to the reader as the next exercise.

Exercise 3.23. (a) Prove the correctness of algorithm 9. *Hint : Use results of Exercise 3.22 (b).*

The next exercise, which is also not too difficult to prove, establishes a correspondence between value iteration in the finite and infinite horizon cases.

Exercise 3.24. Consider a MDP $M = (S, A, P, R, \gamma)$ with infinite horizon and $\gamma < 1$. Let V^* be the optimal value function of M. Define a sequence of finite horizon MDPs M_k with horizon H_k, such that $M_k = M$ and $H_k = k$, for all $k = 1, 2, \ldots$. Let $\{(V_k)^*\}_{k \geq 1}$ be the sequence of optimal value functions returned by algorithm 9 when run with the input M_k, and corresponding to $t = 0$. (a) Prove that $(V_k)^* \to V^*$ as $k \to \infty$.

Appendices

A Contraction mapping theorem [1]

In this section, we introduce the notion of contraction maps in a Banach space setting, that we have heavily relied on in the previous section to prove many of our important theorems. The notation used in this section will be completely independent of what was introduced before, and so the reader should read this section in a self-contained fashion.

Let $(V, ||\cdot||)$ be a Banach space, where V is a vector space and $||\cdot||$ is the norm defined on the vector space. V may be finite or infinite dimensional. As it is a Banach space, we remind the reader that the space is complete, meaning that all Cauchy sequences (Definition A.1) converge (Definition A.2). We first give a few definitions:

Definition A.1. A sequence $\{v_k\}_{k\geq 1}$ of elements $v_k \in V$, $\forall\, k = 1, 2, \ldots$, is called a *Cauchy sequence* iff for every real number $\epsilon > 0$ there exists an integer $N \geq 1$, such that $||v_m - v_n|| < \epsilon$ for all $m, n > N$.

Definition A.2. Let $\{v_k\}_{k\geq 1}$ be a sequence of elements of V. We say that the sequence *converges* to an element $v \in V$, iff for every real number $\epsilon > 0$ there exists an integer $N \geq 1$, such that $||v_k - v|| < \epsilon$ for all $k \geq N$. We write this as $v_k \to v$.

Our first theorem of this section shows that any sequence that is eventually constant is Cauchy.

Theorem A.1. *A sequence $\{v_k\}_{k\geq 1}$ in a normed vector space that is eventually constant is Cauchy.*

Proof. As the sequence is eventually constant, there exists a positive integer r and $v \in V$ such that for all $k \geq r$, $v_k = v$. Then for any $\epsilon > 0$, one can choose $N = r$ in Definition A.1, giving $0 = ||v_m - v_n|| < \epsilon$ for all $m, n > N$, thus completing the proof. □

We can now prove that the limit of a Cauchy sequence is unique.

Theorem A.2. *A Cauchy sequence $\{v_k\}_{k\geq 1}$ in a Banach space converges to a unique limit.*

Proof. The fact that the Cauchy sequence converges to a limit is true by the definition of a Banach space. We need to show that this limit is unique. We prove it by contradiction.

Suppose $\exists v, w \in V$, $v \neq w$, such that $v_k \to v$ and $v_k \to w$. Let $\delta = ||v - w||$, and note that $\delta > 0$ as $v \neq w$. By Definition A.2, there exist positive integers M, N such that $||v_m - v|| < \delta/2 \;\;,\forall\; m \geq M$ and $||v_n - w|| < \delta/2 \;\;,\forall\; n \geq N$. Let $l = \max(M, N)$. Then by triangle inequality we have, $||v - w|| \leq ||v - v_l|| + ||v_l - w|| < \delta$, which is a contradiction. □

We next define the notion of a *"contraction map"* on a Banach space, and the notion of a *"fixed point"* of an operator that maps V to itself.

Definition A.3. A function $T : V \to V$ is called a *contraction* on V iff for every $v, w \in V$, $||Tv - Tw|| \leq ||v - w||$. The map is called a *strict contraction* iff there exists a real number $0 \leq \gamma < 1$, such that for every $v, w \in V$, $||Tv - Tw|| \leq \gamma||v - w||$. The constant γ is called the *contraction factor* of T.

Definition A.4. Consider a function $T : V \to V$. We say that $v \in V$ is a *fixed point* of T in V, iff $Tv = v$.

[1] Additional material that was not covered in class.

We should note that a map $T : V \to V$ may have many fixed points or none. For example, the contraction map $T : \mathbb{R} \to \mathbb{R}$ given by $T(x) = x + 1$ has no fixed points in $\mathbb{R}$. On the other hand the map $T : \mathbb{R} \to \mathbb{R}$ given by $T(x) = x$, which is also a contraction, has infinitely many fixed points in $\mathbb{R}$. Similarly, any linear map from V to itself has 0 as a fixed point, but may not be a contraction.

The $\gamma = 0$ case is special, as shown by the following theorem.

Theorem A.3. *Suppose T is a strict contraction on a normed vector space V (not necessarily Banach) with contraction factor $\gamma = 0$. Then T is a constant map.*

Proof. Consider an element $v \in V$, and let $c = Tv$. Now for every element $w \in V$, we have $||Tv - Tw|| \leq 0$, which implies $||Tv - Tw|| = 0$. By property of norms this implies that $Tw = Tv = c$. □

We next prove a theorem involving repeated application of a strict contraction map.

Theorem A.4. *Suppose T is a strict contraction on a normed vector space V (not necessarily Banach) with contraction factor γ. Then for every element $v \in V$, the sequence $\{v, Tv, T^2v, \dots\}$ is a Cauchy sequence.*

Proof. If $\gamma = 0$, Theorem A.3 implies that the sequence $\{v, Tv, T^2v, \dots\}$ is a constant sequence, except for the first term, and hence Cauchy by Theorem A.1.

So assume that $\gamma \neq 0$. Let $\alpha = ||Tv - v||$. By repeated application of the contraction map we have for all $n \geq 0$,

$$||T^{n+1}v - T^n v|| \leq \gamma ||T^n v - T^{n-1}v|| \leq \cdots \leq \gamma^n ||Tv - v|| = \gamma^n \alpha. \tag{49}$$

Then by the triangle inequality and (49) we additionally have for all m, n satisfying $0 \leq n \leq m$,

$$\begin{aligned} ||T^m v - T^n v|| &= \left\| \sum_{k=n}^{m-1} (T^{k+1}v - T^k v) \right\| \leq \sum_{k=n}^{m-1} ||T^{k+1}v - T^k v|| \\ &\leq \sum_{k=n}^{m-1} \gamma^k \alpha = \alpha \left(\frac{\gamma^n - \gamma^m}{1 - \gamma} \right) < \frac{\alpha \gamma^n}{1 - \gamma} \,. \end{aligned} \tag{50}$$

To prove the sequence is Cauchy, we fix an $\epsilon > 0$, and set $N = \max\left(1, \left\lceil \log\left(\frac{\epsilon(1-\gamma)}{\alpha}\right) \Big/ \log \gamma \right\rceil\right)$. Then for all m, n satisfying $m \geq n > N$, and as a consequence of (50), we have

$$||T^m v - T^n v|| \leq \frac{\alpha \gamma^n}{1 - \gamma} < \frac{\alpha \gamma^N}{1 - \gamma} \leq \epsilon \,, \tag{51}$$

which completes the proof. □

We can now prove the main result of this section : *"the contraction mapping theorem".*

Theorem A.5. *Suppose the function $T : V \to V$ is a strict contraction on a Banach space V. Then T has a unique fixed point in V. Moreover, for every element $v \in V$, the sequence $\{v, Tv, T^2v, \dots\}$ is Cauchy and converges to the fixed point.*

Proof. As T is a strict contraction, let $\gamma \in [0, 1)$ be the contraction factor of T.

We first prove the uniqueness part by contradiction. Let $v, w \in V$ be fixed points of T and $v \neq w$, so $||v - w|| > 0$. Then we have that $||Tv - Tw|| = ||v - w||$. By the contraction property we also have $||Tv - Tw|| \leq \gamma ||v - w|| < ||v - w||$. But then this implies $||v - w|| < ||v - w||$, a contradiction.

We now prove the existence part. Take any element $v \in V$ and consider the sequence $\{v_k\}_{k\geq 1}$ defined as follows:

$$v_k = \begin{cases} v & \text{if } k = 1, \\ Tv_{k-1} & \text{if } k > 1 \end{cases} . \tag{52}$$

Then by Theorem A.4, $\{v_k\}_{k\geq 1}$ is a Cauchy sequence, and hence as V is a Banach space, the sequence converges to a unique limit $v^* \in V$ by Theorem A.2. We claim that v^* is a fixed point of T. To prove this, choose any $\epsilon > 0$ and define $\delta = \epsilon/(1+\gamma)$. As $v_k \to v^*$, by Definition A.2, $\exists\ N \geq 1$ such that $||v_k - v^*|| < \delta\ ,\ \forall\ k \geq N$. Then by triangle inequality we have:

$$\begin{aligned} ||Tv^* - v^*|| &\leq ||Tv^* - v_{N+1}|| + ||v_{N+1} - v^*|| \\ &= ||Tv^* - Tv_N|| + ||v_{N+1} - v^*|| \\ &\leq \gamma||v^* - v_N|| + ||v_{N+1} - v^*|| \\ &< \gamma\delta + \delta = \epsilon\ . \end{aligned} \tag{53}$$

Thus we have proved that $||Tv^* - v^*|| < \epsilon$ for all $\epsilon > 0$, which implies that $||Tv^* - v^*|| = 0$. As V is a normed vector space, this finally implies that $Tv^* = v^*$, thus completing the existence proof and also proving the second part of the theorem. □

B Solutions to selected exercises

Exercise 3.3

Solution. The transition probability matrix is given by:

$$\mathbf{P} = \begin{array}{c} \\ S1 \\ S2 \\ S3 \\ S4 \\ S5 \\ S6 \\ S7 \end{array} \begin{array}{c} \begin{array}{ccccccc} S1 & S2 & S3 & S4 & S5 & S6 & S7 \end{array} \\ \begin{pmatrix} 0.6 & 0.4 & 0 & 0 & 0 & 0 & 0 \\ 0.4 & 0.2 & 0.4 & 0 & 0 & 0 & 0 \\ 0 & 0.4 & 0.2 & 0.4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.2 & 0.4 & 0 & 0 \\ 0 & 0 & 0 & 0.4 & 0.2 & 0.4 & 0 \\ 0 & 0 & 0 & 0 & 0.4 & 0.2 & 0.4 \\ 0 & 0 & 0 & 0 & 0 & 0.4 & 0.6 \end{pmatrix} \end{array}$$

Exercise 3.9

Solution. If the states are ordered as $\{S1,\ S2,\ S3,\ S4,\ S5,\ S6,\ S7\}$, the value function vector can be found by solving (12). The result is $V = [1.53,\ 0.37,\ 0.13,\ 0.22,\ 0.85,\ 3.59,\ 15.31]^T$.

Exercise 3.17

Solution. In both cases the value function of the policy is given by the vector $V^\pi = [1,\ 0,\ 0,\ 0,\ 0,\ 0,\ 10]^T$.

Exercise 3.20

Solution. The agent has 2^7 deterministic stationary policies available to it. When $\gamma < 1$, the optimal policy is unique and the action in each state is to *"try right"*. If $\gamma = 1$, the optimal policy is not unique. All policies lead to infinite reward and are hence optimal.

CS234 Notes - Lecture 3
Model Free Policy Evaluation: Policy Evaluation Without Knowing How the World Works

Youkow Homma, Emma Brunskill

March 20, 2018

4 Model-Free Policy Evaluation

In the previous lecture, we began by discussing three problem formulations of increasing complexity, which we recap below.

1. A Markov process (MP) is a stochastic process augmented with the Markov property.
2. A Markov reward process (MRP) is a Markov process with rewards at each time step and the accumulation of discounted rewards, called values.
3. A Markov decision process (MDP) is a Markov reward process augmented with choices, or actions, at each state.

In the second half of last lecture, we discussed two methods for evaluating a given policy in an MDP and three methods for finding the optimal policy of an MDP. The two methods for policy evaluation were directly solving via a linear system of equations and dynamic programming. The three methods for control were brute force policy search, policy iteration and value iteration.

Implicit in all of these methods was the assumption that we know both the rewards and probabilities for every transition. However, in many cases, such information is not readily available to us, which necessitates **model-free algorithms**. In this set of lectures notes, we will be discussing **model-free policy evaluation**. That is, we will be given a policy and will attempt to learn the value of that policy without leveraging knowledge of the rewards or transition probabilities. In particular, we will not be discussing how to improve our policies in the model-free case until next lecture.

4.1 Notation Recap

Before diving into some methods for model-free policy evaluation, we'll first recap some of the notation surrounding MDP's from last lecture that we'll need in this lecture.

Recall that we define the return of an MRP as the discounted sum of rewards starting from time step t and ending at horizon H, where H may be infinite. Mathematically, this takes the form

$$G_t = \sum_{i=t}^{H-1} \gamma^{i-t} r_i, \tag{1}$$

for $0 \leq t \leq H-1$, where $0 < \gamma \leq 1$ is the discount factor and r_i is the reward at time step i. For an MDP, the return G_t is defined identically, and the rewards r_i are generated by following policy $\pi(a|s)$.

The state value function $V^\pi(s)$ is the expected return from starting at state s under stationary policy π. Mathematically, we can express this as

$$V^\pi(s) = \mathbb{E}_\pi[G_t|s_t = s] \tag{2}$$

$$= \mathbb{E}_\pi\left[\sum_{i=t}^{H-1} \gamma^{i-t} r_i | s_t = s\right]. \tag{3}$$

The state-action value function $Q^\pi(s,a)$ is the expected return from starting in state s, taking action a, and then following stationary policy π for all transitions thereafter. Mathematically, we can express this as

$$Q^\pi(s,a) = \mathbb{E}_\pi[G_t|s_t = s, a_t = a] \tag{4}$$

$$= \mathbb{E}_\pi\left[\sum_{i=t}^{H-1} \gamma^{i-t} r_i | s_t = s, a_t = a\right]. \tag{5}$$

Throughout this lecture, we will assume an infinite horizon as well as stationary rewards, transition probabilities and policies. This allows us to have time-independent state and state-action value functions, as derived last lecture.

There is one new definition that we will use in this lecture, which is the definition of a history.

Definition 4.1. *The **history** is the ordered tuple of states, actions and rewards that an agent experiences. In episodic domains, we will use the word episode interchangeably with history. When considering many interactions, we will index the histories in the following manner: the jth history is*

$$h_j = (s_{j,1}, a_{j,1}, r_{j,1}, s_{j,2}, a_{j,2}, r_{j,2}, \ldots, s_{j,L_j}),$$

where L_j is the length of the interaction, and $s_{j,t}, a_{j,t}, r_{j,t}$ are the state, action and reward at time step t in history j, respectively.

4.2 Dynamic Programming

Recall also from last lecture the dynamic programming algorithm to calculate the value of an infinite horizon MDP with $\gamma < 1$ under policy π, which we rewrite here for convenience as Algorithm 1.

Algorithm 1 Iterative algorithm to calculate MDP value function for a stationary policy π

procedure POLICY EVALUATION(M, π, ϵ)
  For all states $s \in S$, define $R^\pi(s) = \sum_{a\in A} \pi(a|s) R(s,a)$
  For all states $s, s' \in S$, define $P^\pi(s'|s) = \sum_{a\in A} \pi(a|s) P(s'|s,a)$
  For all states $s \in S$, $V_0(s) \leftarrow 0$
  $k \leftarrow 0$
  while $k = 0$ or $||V_k - V_{k-1}||_\infty > \epsilon$ **do**
    $k \leftarrow k + 1$
    For all states $s \in S$, $V_k(s) = R^\pi(s) + \gamma \sum_{s'\in S} P^\pi(s'|s) V_{k-1}(s')$
  return V_k

Written in this form, we can think of V_k in a few ways. First, $V_k(s)$ is the exact value of following policy π for k additional transitions, starting at state s. Second, for large k, or when Algorithm 1 terminates, $V_k(s)$ is an estimate of the true, infinite horizon value $V^\pi(s)$ of state s.

We can additionally express the behavior of this algorithm via a **backup diagram**, which is shown in Figure 1. This backup diagram is read top-down with white circles representing states, black circles

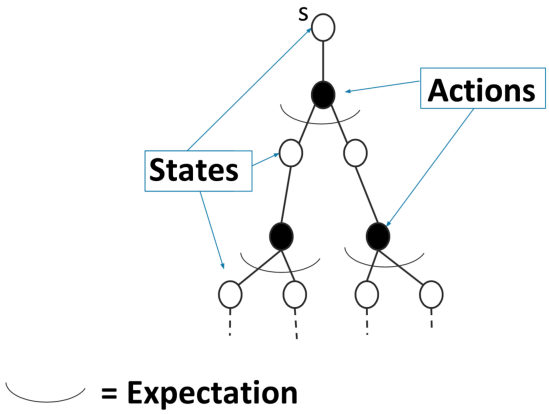


Figure 1: Backup diagram for the dynamic programming policy evaluation algorithm.

representing actions and arcs representing taking the expectation. The diagram shows the branching effect of starting at state s at the top and transitioning two time steps as we move down the diagram. It furthermore shows how after starting at state s and taking an action under policy π, we take the expectation over the value of the next state. In dynamic programming, we **bootstrap**, or estimate, the value of the next state using our current estimate, $V_{k-1}(s')$.

4.3 Monte Carlo On Policy Evaluation

We now describe our first model-free policy evaluation algorithm which uses a popular computational method called the Monte Carlo method. We first walk through an example of the Monte Carlo method outside the context of reinforcement learning, then discuss the method more generally, and finally apply Monte Carlo to reinforcement learning. We emphasize here that this method only works in episodic environments, and we'll see why this is as we examine the algorithm more carefully in this section.

Suppose we want to estimate how long the commute from your house to Stanford's campus will take today. Suppose we also have access to a commute simulator which models our uncertainty of how bad the traffic will be, the weather, construction delays, and other variables, as well as how these variables interact with each other. One way to estimate the expected commute time is to simulate our commute many times on the simulator and then take an average over the simulated commute times. This is called a Monte Carlo estimate of our commute time.

In general, we get the Monte Carlo estimate of some quantity by observing many iterations of how that quantity is generated either in real life or via simulation and then averaging over the observed quantities. By the law of large numbers, this average converges to the expectation of the quantity.

In the context of reinforcement learning, the quantity we want to estimate is $V^\pi(s)$, which is the average of returns G_t under policy π starting at state s. We can thus get a Monte Carlo estimate of $V^\pi(s)$ through three steps:

1. Execute a **rollout** of policy π until termination many times
2. Record the returns G_t that we observe when starting at state s
3. Take an average of the values we get for G_t to estimate $V^\pi(s)$.

The backup diagram for Monte Carlo policy evaluation can be seen in Figure 2. The new blue line indicates that we sample an entire episode until termination starting at state s.

There are two forms of Monte Carlo on policy evaluation, which are differentiated by whether we take an average over just the first time we visit a state in each rollout or every time we visit the state in each

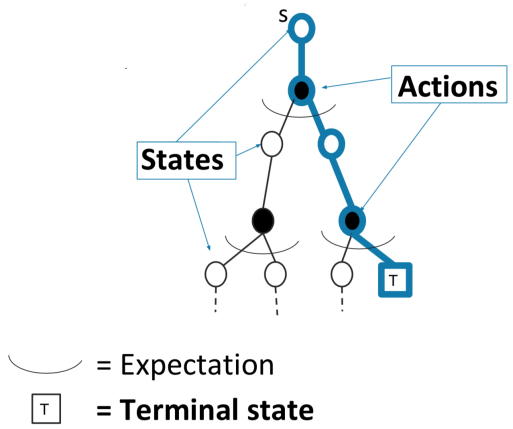


Figure 2: Backup diagram for the Monte Carlo policy evaluation algorithm.

rollout. These are called First-Visit Monte Carlo and Every-Visit Monte Carlo On Policy Evaluation, respectively.

More formally, we describe the First-Visit Monte Carlo in Algorithm 2 and the Every-Visit Monte Carlo in Algorithm 3.

Algorithm 2 First-Visit Monte Carlo Policy Evaluation

procedure FIRST-VISIT-MONTE-CARLO($h_1, \ldots, h_j$)
  For all states s, $N(s) \leftarrow 0$, $S(s) \leftarrow 0$, $V(s) \leftarrow 0$
  for each episode h_j **do**
    for $t = 1, \ldots, L_j$ **do**
      if $s_{j,t} \neq s_{j,u}$ for $u < t$ **then**
        $N(s_{j,t}) \leftarrow N(s_{j,t}) + 1$
        $S(s_{j,t}) \leftarrow S(s_{j,t}) + G_{j,t}$
        $V^{\pi}(s_{j,t}) \leftarrow S(s_{j,t})/N(s_{j,t})$
  return V^{π}

Algorithm 3 Every-Visit Monte Carlo Policy Evaluation

procedure EVERY-VISIT-MONTE-CARLO($h_1, \ldots, h_j$)
  For all states s, $N(s) \leftarrow 0$, $S(s) \leftarrow 0$, $V(s) \leftarrow 0$
  for each episode h_j **do**
    for $t = 1, \ldots, L_j$ **do**
      $N(s_{j,t}) \leftarrow N(s_{j,t}) + 1$
      $S(s_{j,t}) \leftarrow S(s_{j,t}) + G_{j,t}$
      $V^{\pi}(s_{j,t}) \leftarrow S(s_{j,t})/N(s_{j,t})$
  return V^{π}

Note that in the body of the for loop in Algorithms 2 and 3, we can remove vector S and replace the update for $V^{\pi}(s_{j,t})$ with

$$V^{\pi}(s_{j,t}) \leftarrow V^{\pi}(s_{j,t}) + \frac{1}{N(s_{j,t})}(G_{j,t} - V^{\pi}(s_{j,t})). \tag{6}$$

This is because the new average is the average of $N(s_{j,t}) - 1$ of the old values $V^{\pi}(s_{j,t})$ and the new return $G_{j,t}$, giving us

$$\frac{V^{\pi}(s_{j,t}) \times (N(s_{j,t}) - 1) + G_{j,t}}{N(s_{j,t})} = V^{\pi}(s_{j,t}) + \frac{1}{N(s_{j,t})}(G_{j,t} - V^{\pi}(s_{j,t})), \tag{7}$$

which is precisely the new form of the update.

Replacing $\frac{1}{N(s_{j,t})}$ with α in this new update gives us the more general **Incremental Monte Carlo On Policy Evaluation**. Algorithms 4 and 5 detail this procedure in the First-Visit and Every-Visit cases, respectively.

Algorithm 4 Incremental First-Visit Monte Carlo Policy Evaluation

procedure INCREMENTAL-FIRST-VISIT-MONTE-CARLO($\alpha, h_1, \ldots, h_j$)
  For all states s, $N(s) \leftarrow 0$, $V(s) \leftarrow 0$
  for each episode h_j **do**
    for $t = 1, \ldots, terminal$ **do**
      if $s_{j,t} \neq s_{j,u}$ for $u < t$ **then**
        $N(s_{j,t}) \leftarrow N(s_{j,t}) + 1$
        $V^\pi(s_{j,t}) \leftarrow V^\pi(s) + \alpha(G_{j,t} - V^\pi(s))$
  return V^π

Algorithm 5 Incremental Every-Visit Monte Carlo Policy Evaluation

procedure INCREMENTAL-EVERY-VISIT-MONTE-CARLO($\alpha, h_1, \ldots, h_j$)
  For all states s, $N(s) \leftarrow 0$, $V(s) \leftarrow 0$
  for each episode h_j **do**
    for $t = 1, \ldots, terminal$ **do**
      $N(s_{j,t}) \leftarrow N(s_{j,t}) + 1$
      $V^\pi(s_{j,t}) \leftarrow V^\pi(s) + \alpha(G_{j,t} - V^\pi(s))$
  return V^π

Setting $\alpha = \frac{1}{N(s_{j,t})}$ recovers the original Monte Carlo On Policy Evaluation algorithms given in Algorithms 2 and 3, while setting $\alpha > \frac{1}{N(s)}$ gives a higher weight to newer data, which can help learning in non-stationary domains. If we are in a truly Markovian-domain, Every-Visit Monte Carlo will be more data efficient because we update our average return for a state every time we visit the state.

Exercise 4.1. *Recall our Mars Rover MDP from last lecture, shown in Figure 3 below. Suppose that our estimate for the value of each state is currently 0. If we experience the history*

$$h = (S3, TL, +0, S2, TL, +0, S1, TL, +1, terminal),$$

then:

1. *What is the first-visit Monte Carlo estimate of V at each state?*
2. *What is the every-visit Monte Carlo estimate of each state?*
3. *What is the incremental first-visit Monte Carlo estimate of V with $\alpha = \frac{2}{3}$?*
4. *What is the incremental every-visit Monte Carlo estimate of V with $\alpha = \frac{2}{3}$?*

4.4 Monte Carlo Off Policy Evaluation

In the section above, we discussed the case where we are able to obtain many realizations of G_t under the policy π that we want to evaluate. However, in many costly or high stakes situations, we aren't able to obtain rollouts of G_t under the policy that we wish to evaluate. For example, we may have data associated with one medical policy, but want to determine the value of a different medical policy. In this section, we describe Monte Carlo off policy evaluation, which is a method for using data taken from one policy to evaluate a different policy.

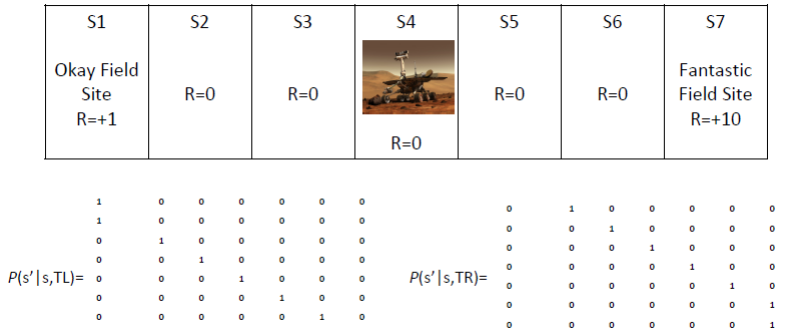


Figure 3: Mars Rover Markov Decision Process with actions Try Left (TL) and Try Right (TR)

4.4.1 Importance Sampling

The key ingredient of off policy evaluation is a method called importance sampling. The goal of importance sampling is to estimate the expected value of a function $f(x)$ when x is drawn from distribution q using only the data $f(x_1), \ldots, f(x_n)$, where x_i are drawn from a different distribution p. In summary, given $q(x_i), p(x_i), f(x_i)$ for $1 \leq x_i, \leq n$, we would like an estimate for $\mathbb{E}_{x\sim q}[f(x)]$. We can do this via the following approximation:

$$\mathbb{E}_{x\sim q}[f(x)] = \int_x q(x)f(x)dx \tag{8}$$

$$= \int_x p(x)\left[\frac{q(x)}{p(x)}f(x)\right]dx \tag{9}$$

$$= \mathbb{E}_{x\sim p}\left[\frac{q(x)}{p(x)}f(x)\right] \tag{10}$$

$$\approx \sum_{i=1}^{n}\left[\frac{q(x_i)}{p(x_i)}f(x_i)\right]. \tag{11}$$

The last equation gives us the **importance sampling estimate** of f under distribution q using samples of f under distribution p. Note that the first step only holds if $q(x)f(x) > 0$ implies $p(x) > 0$ for all x.

4.4.2 Importance Sampling for Off Policy Evaluation

We now apply the general result of importance sampling estimates to reinforcement learning. In this instance, we want to approximate the value of state s under policy π_1, given by $V^{\pi_1}(s) = \mathbb{E}[G_t|s_t = s]$, using n histories $h_1, \ldots, h_n$ generated under policy π_2. Using the importance sampling estimate result gives us that

$$V^{\pi_1}(s) \approx \frac{1}{n}\sum_{j=1}^{n}\frac{p(h_j|\pi_1, s)}{p(h_j|\pi_2, s)}G(h_j), \tag{12}$$

where $G(h_j) = \sum_{t=1}^{L_j-1}\gamma^{t-1}r_{j,t}$ is the total discounted sum of rewards for history h_j.

Now, for a general policy π, we have that the probability of experiencing history h_j under policy π is

$$p(h_j|\pi, s = s_{j,1}) = \prod_{t=1}^{L_j-1} p(a_{j,t}|s_{j,t})p(r_{j,t}|s_{j,t}, a_{j,t})p(s_{j,t+1}|s_{j,t}, a_{j,t}) \tag{13}$$

$$= \prod_{t=1}^{L_j-1} \pi(a_{j,t}|s_{j,t})p(r_{j,t}|s_{j,t}, a_{j,t})p(s_{j,t+1}|s_{j,t}, a_{j,t}), \tag{14}$$

where L_j is the length of the jth episode. The first line follows from looking at the three components of each transition. The components are:

1. $p(a_{j,t}|s_{j,t})$ - probability we take action $a_{j,t}$ at state $s_{j,t}$
2. $p(r_{j,t}|s_{j,t}, a_{j,t})$ - probability we experience reward $r_{j,t}$ after taking action $a_{j,t}$ in state $s_{j,t}$
3. $p(s_{j,t+1}|s_{j,t}, a_{j,t})$ - probability we transition to state $s_{j,t+1}$ after taking action $a_{j,t}$ in state $s_{j,t}$

Now, combining our importance sampling estimate for $V^{\pi_1}(s)$ with our decomposition of the history probabilities, $p(h_j|\pi, s = s_{j,1})$, we get that

$$V^{\pi_1}(s) \approx \frac{1}{n}\sum_{j=1}^{n} \frac{p(h_j|\pi_1, s)}{p(h_j|\pi_2, s)} G(h_j) \tag{15}$$

$$= \frac{1}{n}\sum_{j=1}^{n} \frac{\prod_{t=1}^{L_j-1} \pi_1(a_{j,t}|s_{j,t})p(r_{j,t}|s_{j,t}, a_{j,t})p(s_{j,t+1}|s_{j,t}, a_{j,t})}{\prod_{t=1}^{L_j-1} \pi_2(a_{j,t}|s_{j,t})p(r_{j,t}|s_{j,t}, a_{j,t})p(s_{j,t+1}|s_{j,t}, a_{j,t})} G(h_j) \tag{16}$$

$$= \frac{1}{n}\sum_{j=1}^{n} G(h_j) \prod_{t=1}^{L_j-1} \frac{\pi_1(a_{j,t}|s_{j,t})}{\pi_2(a_{j,t}|s_{j,t})}. \tag{17}$$

Notice we can now explicitly evaluate the expression without the transition probabilities or rewards since all of the terms involving model dynamics canceled out in the second step of the equation. In particular, we are given the histories h_j, so we can calculate $G(h_j) = \sum_{t=1}^{L_j-1} \gamma^{t-1} r_{j,t}$, and we know the two policies π_1 and π_2, so we can also evaluate the second term.

4.5 Temporal Difference (TD) Learning

So far, we have two methods for policy evaluation: dynamic programming and Monte Carlo. Dynamic programming leverages bootstrapping to help us get value estimates with only one backup. On the other hand, Monte Carlo samples many histories for many trajectories which frees us from using a model. Now, we introduce a new algorithm that combines bootstrapping with sampling to give us a second model-free policy evaluation algorithm.

To see how to combine sampling with bootstrapping, let's go back to our incremental Monte Carlo update:

$$V^{\pi}(s_t) \leftarrow V^{\pi}(s_t) + \alpha(G_t - V^{\pi}(s_t)). \tag{18}$$

Recall that G_t is the return after rolling out the policy from time step t to termination starting at state s_t. Let's now replace G_t with a Bellman backup like in dynamic programming. That is, let's replace G_t with $r_t + \gamma V^{\pi}(s_{t+1})$, where r_t is a sample of the reward at time step t and $V^{\pi}(s_{t+1})$ is our current estimate of the value at the next state. Making this substitution gives us the TD-learning update

$$V^{\pi}(s_t) \leftarrow V^{\pi}(s_t) + \alpha(r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)). \tag{19}$$

The difference

$$\delta_t = r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t) \tag{20}$$

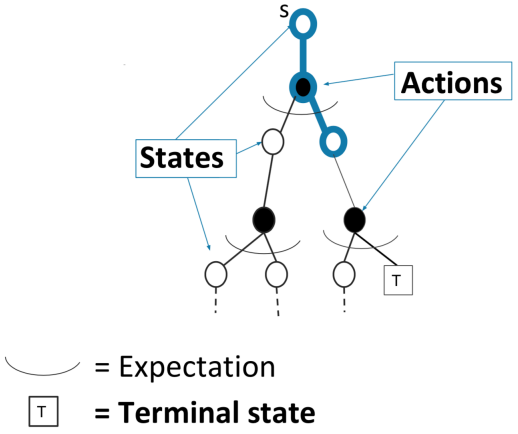


Figure 4: Backup diagram for the TD Learning policy evaluation algorithm.

is commonly referred to as the **TD error**, and the sampled reward combined with the bootstrap estimate of the next state value,

$$r_t + \gamma V^\pi(s_{t+1}), \tag{21}$$

is referred to as the **TD target**. The full TD learning algorithm is given in Algorithm 6. We can see that using this method, we update our value for $V^\pi(s_t)$ directly after witnessing the transition (s_t, a_t, r_t, s_{t+1}). In particular, we don't need to wait for the episode to terminate like in Monte Carlo.

Algorithm 6 TD Learning to evaluate policy π

procedure TDLEARNING(step size α, number of trajectories n)
  For all states s, $V^\pi(s) \leftarrow 0$
  while $n > 0$ **do**
    Begin episode E at state s
    while $n > 0$ and episode E has not terminated **do**
      $a \leftarrow$ action at state s under policy π
      Take action a in E and observe reward r, next state s'
      $V^\pi(s) \leftarrow V^\pi(s) + \alpha(R + \gamma V^\pi(s') - V^\pi(s))$
      $s \leftarrow s'$
  return V^π

We can again examine this algorithm via a backup diagram as shown in Figure 4. Here, we see via the blue line that we sample one transition starting at s, then we estimate the value of the next state via our current estimate of the next state to construct a full Bellman backup estimate.

There is actually an entire spectrum of ways we can blend Monte Carlo and dynamic programming using a method called TD(λ). When $\lambda = 0$, we get the TD-learning formulation above, hence giving us the alias TD(0). When $\lambda = 1$, we recover Monte Carlo policy evaluation, depending on the formulation used. When $0 < \lambda < 1$, we get a blend of these two methods. For a more thorough treatment of TD(λ), we refer the interested reader to Sections 7.1 and 12.1-12.5 of Sutton and Barto [1] which detail n-step TD learning and TD(λ)/eligibility traces, respectively.

Exercise 4.2. *Consider again the Mars Rover example in Figure 3. Suppose that our estimate for the value of each state is currently 0. If we experience the history*

$$h = (S3, TL, +0, S2, TL, +0, S2, TL, +0, S1, TL, +1, terminal),$$

then:

1. *What is the TD(0) estimate of V with $\alpha = 1$?*
2. *What is the TD(0) estimate of V with $\alpha = \frac{2}{3}$?*

4.6 Summary of Methods Discussed

In this lecture, we re-examined policy evaluation using dynamic programming from last lecture, and we introduced two new methods for policy evaluation, namely Monte Carlo evaluation and Temporal Difference (TD) learning.

First, we motivated the introduction of Monte Carlo and TD-Learning by noting that dynamic programming relied on a model of the world. That is, we needed to feed our dynamic programming policy evaluation algorithm with the rewards and transition probabilities of the domain. Monte Carlo and TD-Learning are both free from this constraint, making them model-free methods.

In Monte Carlo policy evaluation, we generate many histories and then average the returns over the states that we encounter. In order for us to generate these histories in a finite amount of time, we require the domain to be episodic - that is, we need to ensure that each history that we observe terminates. In both dynamic programming and temporal difference learning, we only backup over one transition (we only look one step ahead in the future), so termination of histories is not a concern, and we can apply these algorithms to non-episodic domains.

On the flip side, the reason we are able to backup over just one transition in dynamic programming and TD learning is because we leverage the Markovian assumption of the domain. Furthermore, Incremental Monte Carlo policy evaluation, described in Algorithms 4 and 5 can be utilized in non-Markovian domains.

In all three methods, we converge to the true value function. Last lecture, we proved this result for dynamic programming by using the fact that the Bellman backup operator is a contraction. We saw in today's lecture that Monte Carlo policy evaluation converges to the policy's value function due to the law of large numbers. TD(0) converges to the true value as well, which we will look at more closely in the next section on batch learning.

Because we are taking an average over the true distribution of returns in Monte Carlo, we obtain an unbiased estimator of the value at each state. On the other hand, in TD learning, we bootstrap the next state's value estimate to get the current state's value estimate, so the estimate is biased by the estimated value of the next state. Further discussions on this can be found in section 6.2 of Sutton and Barto [1].

The variance of Monte Carlo evaluation is relatively higher than TD learning because in Monte Carlo evaluation, we consider many transitions in each episode with each transition contributing variance to our estimate. On the other hand, TD learning only considers one transition per update, so we do not accumulate variance as quickly.

Finally, Monte Carlo is generally more data efficient than TD(0). In Monte Carlo, we update the value of a state based on the returns of the entire episode, so if there are highly positive or negative rewards many trajectories in the future, these rewards will still be immediately incorporated into our update of the value of the state. On the other hand in TD(0), we update the value of a state using only the reward in the current step and some previous estimate of the value at the next state. This means that if there are highly positive or negative rewards many trajectories in the future, we will only incorporate these into the current state's value update when that reward has been used to update the bootstrap estimate of the next state's value. This means that if a highly rewarding episode has length L, then we may need to experience that episode L times for the information of the highly rewarding episode to travel all the way back to the starting state.

In Table 1, we summarize the strengths and limitations of each method discussed here.

	Dynamic Programming	Monte Carlo	Temporal Difference
Model Free?	No	Yes	Yes
Non-episodic domains?	Yes	No	Yes
Non-Markovian domains?	No	Yes	No
Converges to true value	Yes	Yes	Yes
Unbiased Estimate	N/A	Yes	No
Variance	N/A	High	Low

Table 1: Summary of methods in this lecture.

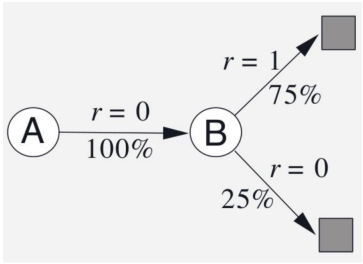


Figure 5: Example 6.4 from Sutton and Barto [1].

4.7 Batch Monte Carlo and Temporal Difference

We now look at the batch versions of the algorithms in today's lecture, where we have a set of histories that we use to make updates many times. Before looking at the batch cases in generality, let's first look at Example 6.4 from Sutton and Barto [1] to more closely examine the difference between Monte Carlo and TD(0). Suppose $\gamma = 1$ and we have eight histories generated by policy π, take action $act1$ in all states:

$$h_1 = (A, act1, +0, B, act1, +0, terminal)$$
$$h_j = (B, act1, +1, terminal) \text{ for } j = 2, \ldots, 7$$
$$h_8 = (B, act1, +0, terminal).$$

Then, using either batch Monte Carlo or TD(0) with $\alpha = 1$, we see that $V(B) = 0.75$. However, using Monte Carlo, we get that $V(A) = 0$ since only the first episode visits state A and has return 0. On the other hand, TD(0) gives us $V(A) = 0.75$ because we perform the update $V(A) \leftarrow r_{1,1} + \gamma V(B)$. Under a Markovian domain like the one shown in Figure 5, the estimate given by TD(0) makes more sense.

In this section, we consider the batch cases of Monte Carlo and TD(0). In the batch case, we are given a batch, or set of histories $h_1, \ldots, h_n$, which we then feed through Monte Carlo or TD(0) many times. The only difference from our formulations before is that we only update the value function after each time we process the entire batch. Thus in TD(0), the bootstrap estimate is updated only after each pass through the batch.

In the Monte Carlo batch setting, the value at each state converges to the value that minimizes the mean squared error with the observed returns. This follows directly from the fact that in Monte Carlo, we take an average over returns at each state, and in general, the MSE minimizer of samples is precisely the average of the samples. That is, given samples $y_1, \ldots, y_n$, the value $\sum_{i=1}^{n}(y_i - \hat{y})^2$ is minimized for $\hat{y} = \sum_{i=1}^{n} y_i$. We can also see this in the example at the beginning of the section. We get that $V(A) = 0$ for Monte Carlo because this is the only history visiting state A.

In the TD(0) batch setting, we do not converge to the same result as in Monte Carlo. In this case, we

converge to the value V^π that is the value of policy π on the maximum likelihood MDP model where

$$\hat{P}(s'|s,a) = \frac{1}{N(s,a)} \sum_{j=1}^{n} \sum_{t=1}^{L_j - 1} \mathbb{1}(s_{j,t} = s, a_{j,t}, s_{j,t+1} = s') \tag{22}$$

$$\hat{r}(s,a) = \frac{1}{N(s,a)} \sum_{j=1}^{n} \sum_{t=1}^{L_j - 1} \mathbb{1}(s_{j,t} = s, a_{j,t}) r_{j,t}. \tag{23}$$

In other words, the maximum likelihood MDP model is the most naive model we can create based on the batch - the transition probability $\hat{P}(s'|s,a)$ is the fraction of times that we see the transition (s,a,s') after we take action a at state s in the batch, and the reward $\hat{r}(s,a)$ is the average reward experienced after taking action a at state s in the batch.

We also see this result in the example from the beginning of the section. In this case, our maximum likelihood model is

$$\hat{P}(B|A, act1) = 1 \tag{24}$$

$$\hat{P}(terminal|B, act1) = 1 \tag{25}$$

$$\hat{r}(A, act1) = 0 \tag{26}$$

$$\hat{r}(B, act1) = 0.75. \tag{27}$$

This gives us $V^\pi(A) = 0.75$, like we stated before.

The value function derived from the maximum likelihood MDP model is known as the **certainty equivalence estimate**. Using this relationship, we have another method for evaluating the policy. We can first compute the maximum likelihood MDP model using the batch. Then we can compute V^π using this model and the model-based policy evaluation methods discussed in last lecture. This method is highly data efficient but is computationally expensive because it involves solving the MDP which takes time $O(|S|^3)$ analytically and $(|S|^2|A|)$ via dynamic programming.

Exercise 4.3. *Consider again the Mars Rover example in Figure 3. Suppose that our estimate for the value of each state is currently 0. If our batch consists of two histories*

$$h_1 = (S3, TL, +0, S2, TL, +0, S1, TL, +1, terminal)$$
$$h_2 = (S3, TL, +0, S2, TL, +0, S2, TL, +0, S1, TL, +1, terminal)$$

and our policy is TL, then what is the certainty equivalence estimate?

References

[1] Sutton, Richard S. and Andrew G. Barto. *Introduction to Reinforcement Learning*. 2nd ed., MIT Press, 2017. Draft. http://incompleteideas.net/book/the-book-2nd.html.

CS234 Notes - Lecture 4
Model Free Control

Michael Painter, Emma Brunskill

March 20, 2018

5 Model Free Control

Last class, we discussed how to evaluate a given policy while assuming we do not know how the world works and only interacting with the environment. This gave us our **model-free policy evaluation** methods: Monte Carlo policy evaluation and TD learning. This lecture, we will discuss **model-free control** where we learn good policies under the same constraints (only interactions, no knowledge of reward structure or transition probabilities). This framework is important in two types of domains:

1. When the MDP model is unknown, but we can sample trajectories from the MDP, or
2. When the MDP model is known but computing the value function via our model-based control methods is infeasible due to size of the domain.

In this lecture, we will still restrict ourselves to the **tabular** setting, where we can represent each state or state-action value as an element of a lookup table. Next lecture, we will be re-examining the algorithms from today and the previous lecture under the **value function approximation** setting, which involves trying to fit a function to the state or state-action value function.

5.1 Generalized Policy Iteration

Recall first our model-based policy iteration algorithm, which we re-write in algorithm 1, for convenience.

Algorithm 1 Model-based Policy Iteration Algorithm

1: **procedure** POLICY ITERATION(M, ϵ)
2: $\pi \leftarrow$ Randomly choose a policy $\pi \in \Pi$
3: **while** true **do**
4: $V^\pi \leftarrow$ POLICY EVALUATION (M, π, ϵ)
5: $\pi^*(s) \leftarrow \arg\max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^\pi(s')\right]$, $\forall\, s \in S$ (policy improvement)
6: **if** $\pi^*(s) = \pi(s)$ **then**
7: break
8: **else**
9: $\pi \leftarrow \pi^*$
10: $V^* \leftarrow V^\pi$
11: **return** $V^*(s)$, $\pi^*(s)$ for all $s \in S$

Last lecture, we introduced methods for model-free policy evaluation, so we can complete line 4 in a model-free way. However, in order to make this entire algorithm model-free, we must also find a way to do line 5, the policy improvement step, in a model-free way. By definition, we have $Q^\pi(s,a) =$

$R(s,a)+\gamma \sum_{s' \in S} P(s'|s,a)V^{\pi}(s')$. Thus, we can get a model-free policy iteration algorithm (Algorithm 2) by substituting this value into the model-based policy iteration algorithm and using state-action values throughout.

Algorithm 2 Model-free Generalized Policy Iteration Algorithm

1: **procedure** POLICY ITERATION(M, ϵ)
2: $\pi \leftarrow$ Randomly choose a policy $\pi \in \Pi$
3: **while** true **do**
4: $Q^{\pi} \leftarrow$ POLICY EVALUATION (M, π, ϵ)
5: $\pi^*(s) \leftarrow \arg\max_{a \in A} Q^{\pi}(s,a)$, $\forall\, s \in S$ (policy improvement)
6: **if** $\pi^*(s) = \pi(s)$ **then**
7: break
8: **else**
9: $\pi \leftarrow \pi^*$
10: $Q^* \leftarrow Q^{\pi}$
11: **return** $Q^*(s,a)$, $\pi^*(s)$ for all $s \in S$, $a \in A$

There are a few caveats to this algorithm due to the substitution we made in line 5:

1. If policy π is deterministic or doesn't take every action a with some positive probability, then we cannot actually compute the argmax in line 5.
2. The policy evaluation algorithm gives us an estimate of Q^{π}, so it is not clear whether line 5 will monotonically improve the policy like in the model-based case.

5.2 Importance of Exploration

5.2.1 Exploration

In the previous section, we saw that one caveat to the model-free policy iteration algorithm is that the policy π needs to take every action a with some positive probability, so the value of each state-action pair can be determined. In other words, the policy π needs to explore actions, even if they might be suboptimal with respect to our current Q-value estimates.

5.2.2 ϵ-greedy Policies

In order to explore actions that are suboptimal with respect to our current Q-value estimates, we'll need a systematic way to balance exploration of suboptimal actions with exploitation of the optimal, or greedy, action. One naive strategy is to take a random action with small probability and take the greedy action the rest of the time. This type of exploration strategy is called an **ϵ-greedy policy**. Mathematically, an ϵ-greedy policy with respect to the state-action value $Q^{\pi}(s,a)$ takes the following form:

$$\pi(a|s) = \begin{cases} a & \text{with probability } \frac{\epsilon}{|A|} \\ \arg\max_a Q^{\pi}(s,a) & \text{with probability } 1-\epsilon \end{cases}$$

5.2.3 Monotonic ϵ-greedy Policy Improvement

We saw in the second lecture via the policy improvement theorem that if we take the greedy action with respect to the current values and then follow a policy π thereafter, this policy is an improvement to the policy π. A natural question then is whether an ϵ-greedy policy with respect to ϵ-greedy policy π is an improvement on policy π. This would help us address our second caveat of the generalized

policy iteration algorithm, Algorithm 2. Fortunately, there is an analogue of the policy improvement theorem for the ϵ-greedy policy, which we state and derive below.

Theorem 5.1 (Monotonic ϵ-greedy Policy Improvement)**.** *Let π_i be an ϵ-greedy policy. Then, the ϵ-greedy policy with respect to Q^{π_i}, denoted π_{i+1}, is a monotonic improvement on policy π. In other words, $V^{\pi_{i+1}} \geq V^{\pi_i}$.*

Proof. We first show that $Q^{\pi_i}(s, \pi_{i+1}(s)) \geq V^{\pi_i}(s)$ for all states s.

$$\begin{aligned}
Q^{\pi_i}(s, \pi_{i+1}(s)) &= \sum_{a \in A} \pi_{i+1}(a|s) Q^{\pi_i}(s, a) \\
&= \frac{\epsilon}{|A|} \sum_{a \in A} Q^{\pi_i}(s, a) + (1 - \epsilon) \max_{a'} Q^{\pi_i}(s, a') \\
&= \frac{\epsilon}{|A|} \sum_{a \in A} Q^{\pi_i}(s, a) + (1 - \epsilon) \max_{a'} Q^{\pi_i}(s, a') \frac{1 - \epsilon}{1 - \epsilon} \\
&= \frac{\epsilon}{|A|} \sum_{a \in A} Q^{\pi_i}(s, a) + (1 - \epsilon) \max_{a'} Q^{\pi_i}(s, a') \sum_{a \in A} \frac{\pi_i(a|s) - \frac{\epsilon}{|A|}}{1 - \epsilon} \\
&= \frac{\epsilon}{|A|} \sum_{a \in A} Q^{\pi_i}(s, a) + (1 - \epsilon) \sum_{a \in A} \frac{\pi_i(a|s) - \frac{\epsilon}{|A|}}{1 - \epsilon} \max_{a'} Q^{\pi_i}(s, a') \\
&\geq \frac{\epsilon}{|A|} \sum_{a \in A} Q^{\pi_i}(s, a) + (1 - \epsilon) \sum_{a \in A} \frac{\pi_i(a|s) - \frac{\epsilon}{|A|}}{1 - \epsilon} Q^{\pi_i}(s, a) \\
&= \sum_{a \in A} \pi_i(a|s) Q^{\pi_i}(s, a) \\
&= V^{\pi_i}(s)
\end{aligned}$$

The first equality follows from the fact that the first action we take is with respect to policy π_{i+1}, then we follow policy π_i thereafter. The fourth equality follows because $1 - \epsilon = \sum_a \left[\pi_i(a|s) - \frac{\epsilon}{|A|}\right]$.

Now from the policy improvement theorem, we have that $Q^{\pi_i}(s, \pi_{i+1}(s)) \geq V^{\pi_i}(s)$ implies $V^{\pi_{i+1}}(s) \geq V^{\pi_i}(s)$ for all states s, as desired. □

Thus, the monotonic ϵ-greedy policy improvement shows us that our policy does in fact improve if we act ϵ-greedy on the current ϵ-greedy policy.

5.2.4 Greedy in the limit of exploration

We introduced ϵ-greedy strategies above as a naive way to balance exploration of new actions with exploitation of current knowledge; however, we can further refine this balance by introducing a new class of exploration strategies that allow us to make convergence guarantees about our algorithms. This class of strategies is called Greedy in the Limit of Infinite Exploration (GLIE).

Definition 5.1 (Greedy in the Limit of Infinite Exploration (GLIE))**.** A policy π is greedy in the limit of infinite exploration (GLIE) if it satisfies the following two properties:

1. All state-action pairs are visited an infinite number of times. I.e. for all $s \in S, a \in A$,

$$\lim_{i \to \infty} N_i(s, a) \to \infty,$$

where $N_i(s, a)$ is the number of times action a is taken at state s up to and including episode i.

2. The behavior policy converges to the policy that is greedy with respect to the learned Q-function. I.e. for all $s \in S, a \in A$,

$$\lim_{i \to \infty} \pi_i(a|s) = \arg\max_a q(s,a) \text{ with probability 1}$$

One example of a GLIE strategy is an ϵ-greedy policy where ϵ is decayed to zero with $\epsilon_i = \frac{1}{i}$, where i is the episode number. We can see that since $\epsilon_i > 0$ for all i, we will explore with some positive probability at every time step, hence satisfying the first GLIE condition. Since $\epsilon_i \to 0$ as $i \to \infty$, we also have that the policy is greedy in the limit, hence satisfying the second GLIE condition.

5.3 Monte Carlo Control

Now, we will incorporate the exploration strategies described above with our model-free policy evaluation algorithms from last lecture to give us some model-free control methods. The first algorithm that we discuss is the Monte Carlo online control algorithm. In Algorithm 3, we give the formulation for first-visit online Monte Carlo control. An equivalent formulation for every-visit control is given by not checking for the first visit in line 7.

Algorithm 3 Online Monte Carlo Control/On Policy Improvement

1: **procedure** Online Monte Carlo Control
2: Initialize $Q(s,a) = 0$, $Returns(s,a) = 0$ for all $s \in S, a \in A$
3: Set $\epsilon \leftarrow 1$, $k \leftarrow 1$
4: **loop**
5: Sample kth episode $(s_{k1}, a_{k1}, r_{k1}, s_{k2}, \ldots, s_T)$ under policy π
6: **for** $t = 1, \ldots, T$ **do**
7: **if** First visit to (s,a) in episode k **then**
8: Append $\sum_{j=t}^{T} r_{kj}$ to $Returns(s_t, a_t)$
9: $Q(s_t, a_t) \leftarrow average(Returns(s_t, a_t))$
10: $k \leftarrow k + 1$, $\epsilon = \frac{1}{k}$
11: $\pi_k = \epsilon$-greedy with respect to Q (policy improvement)
12: **Return** Q, π

Now, as stated before, GLIE strategies can help us arrive at convergence guarantees for our model-free control methods. In particular, we have the following result:

Theorem 5.2. *GLIE Monte Carlo control converges to the optimal state-action value function. That is* $Q(s,a) \to q(s,a)$.

In other words, if the ϵ-greedy strategy used in Algorithm 3 is GLIE, then the Q value derived from the algorithm will converge to the optimal Q function.

5.4 Temporal Difference Methods for Control

Last lecture, we also introduced TD(0) as a method for model-free policy evaluation. Now, we can build on TD(0) with our ideas of exploration to get TD-style model-free control. There are two methods of doing this: on-policy and off-policy. We first introduce the on-policy method in Algorithm 4, called SARSA.

We can see the policy evaluation update takes place in line 10, while the policy improvement is at line 11. SARSA gets its name from the parts of the trajectory used in the update equation. We can see that to update the Q-value at state-action pair (s,a), we need the reward, next state and next action,

Algorithm 4 SARSA

1: **procedure** SARSA(ϵ, α_t)
2: Initialize $Q(s,a)$ for all $s \in S, a \in A$ arbitrarily except $Q(terminal, \cdot) = 0$
3: $\pi \leftarrow \epsilon$-greedy policy with respect to Q
4: **for** each episode **do**
5: Set s_1 as the starting state
6: Choose action a_1 from policy $\pi(s_1)$
7: **loop** until episode terminates
8: Take action a_t and observe reward r_t and next state s_{t+1}
9: Choose action a_{t+1} from policy $\pi(s_{t+1})$
10: $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$
11: $\pi \leftarrow \epsilon$-greedy with respect to Q (policy improvement)
12: $t \leftarrow t + 1$
13: **Return** Q, π

thereby using the values (s, a, r, s', a'). SARSA is an on-policy method because the actions a and a' used in the update equation are both derived from the policy that is being followed at the time of the update.

Just like in Monte Carlo, we can arrive at the convergence of SARSA given one extra condition on the step-sizes as stated below:

Theorem 5.3. *SARSA for finite-state and finite-action MDP's converges to the optimal action-value, i.e.* $Q(s,a) \to q(s,a)$*, if the following two conditions hold:*

1. *The sequence of policies* π *from is GLIE*
2. *The step-sizes* α_t *satisfy the* ***Robbins-Munro*** *sequence such that*

$$\sum_{t=1}^{\infty} \alpha_t = \infty$$
$$\sum_{t=1}^{\infty} \alpha_t^2 < \infty$$

Exercise 5.1. What is the benefit to performing the policy improvement step after each update in line 11 of Algorithm 4? What would be the benefit to performing the policy improvement step less frequently?

5.5 Importance Sampling for Off-Policy TD

Before diving into off-policy TD-style control, let's first take a step back and look at one way to do off-policy TD policy evaluation. Recall that our TD update took the form

$$V(s) \to V(s) + \alpha(r + \gamma V(s') - V(s)).$$

Now, let's suppose that like in the off-policy Monte Carlo policy evaluation case, we have data from a policy π_b, and we want to estimate the value of policy π_e. Then much like in the Monte Carlo policy evaluation case, we can weight the target by the ratio of the probability of seeing the sample in the behavior policy and the evaluated policy via importance sampling. This new update then takes the form

$$V^{\pi_e}(s) \to V^{\pi_e}(s) + \alpha \left[\frac{\pi_e(a|s)}{\pi_b(a|s)} (r + \gamma V^{\pi_e}(s') - V^{\pi_e}(s)) \right].$$

Notice that because we only use one trajectory sample instead of sampling the entire trajectory like in Monte Carlo, we only incorporate the likelihood ratio from one step. For the same reason, this method also has significantly lower variance than Monte Carlo.

Additionally, π_b doesn't need to be the same at each step, but we do need to know the probability for every step. As in Monte Carlo, we need the two policies to have the same support. That is, if $\pi_e(a|s) \times V^{\pi_e}(s') > 0$, then $\pi_b(a|s) > 0$.

5.6 Q-learning

Now, we return to finding an off-policy method for TD-style control. In the above formulation, we again leveraged importance sampling, but in the control case, we do not need to rely on this. Instead, we can maintain state-action Q estimates and bootstrap the value of the best future action. Our SARSA update took the form

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)],$$

but we can instead bootstrap the Q value at the next state to get the following update:

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

This gives rise to **Q-learning**, which is detailed in Algorithm 5. Now that we take a maximum over the actions at the next state, this action is not necessarily the same as the one we would derive from the current policy. Therefore, Q-learning is considered an off-policy algorithm.

Algorithm 5 Q-Learning withϵ-greedy exploration

procedure Q-Learning(ϵ, α, γ)
  Initialize $Q(s, a)$ for all $s \in S, a \in A$ arbitrarily except $Q(terminal, \cdot) = 0$
  $\pi \leftarrow \epsilon$-greedy policy with respect to Q
  for each episode **do**
    Set s_1 as the starting state
    $t \leftarrow 1$
    loop until episode terminates
      Sample action a_t from policy $\pi(s_t)$
      Take action a_t and observe reward r_t and next state s_{t+1}
      $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t))$
      $\pi \leftarrow \epsilon$-greedy policy with respect to Q (policy improvement)
      $t \leftarrow t + 1$
  return Q, π

6 Maximization Bias

Finally, we are going to discuss a phenomenon known as **maximization bias**. We'll first examine maximization bias in a small example.

6.1 Example: Coins

Suppose there are two identical fair coins, but we don't know that they are fair or identical. If a coin lands on heads, we get one dollar and if a coin lands on tails, we lose a dollar. We want to answer the following two questions:

1. Which coin will yield more money for future flips?
2. How much can I expect to win/lose per flip using the coin from question 1?

In an effort to answer this question, we flip each coin once. We then pick the coin that yields more money as the answer to question 1. We answer question 2 with however much that coin gave us. For example, if coin 1 landed on heads and coin 2 landed on tails, we would answer question 1 with coin 1, and question 1 with one dollar.

Let's examine the possible scenarios for the outcome of this procedure. If at least one of the coins is heads, then our answer to question 2 is one dollar. If both coins are tails, then our answer is negative one dollar. Thus, the expected value of our answer to question 2 is $\frac{3}{4} \times (1) + \frac{1}{4} \times (-1) = 0.5$. This gives us a higher estimate of the expected value of flipping the better coin than the true expected value of flipping that coin. In other words, we're systematically going to think the coins are better than they actually are.

This problem comes from the fact that we are using our estimate to both choose the better coin and estimate its value. We can alleviate this by separating these two steps. One method for doing this would be to change the procedure as follows: After choosing the better coin, flip the better coin again and use this value as your answer for question 2. The expected value of this answer is now 0, which is the same as the true expected value of flipping either coin.

6.2 Maximization Bias in Reinforcement Learning

Let's now formalize what we saw above in the context of a one-state MDP with two actions. Suppose we are in state s and have two actions a_1 and a_2, both with mean reward 0. Then, we have that the true values of the state as well as the state-action pairs are zero. That is, $Q(s, a_1) = Q(s, a_2) = V(s) = 0$. Suppose that we've sampled some rewards for taking each action so that we have some estimates for the state-action values, $\hat{Q}(s, a_1), \hat{Q}(s, a_2)$. Suppose further that these samples are unbiased because they were generated using a Monte Carlo method. In other words, $\hat{Q}(s, a_i) = \frac{1}{n(s,a_i)} \sum_{j=1}^{n(s,a_i)} r_j(s, a_i)$, where $n(s, a_i)$ is the number of samples for the state-action pair (s, a_i). Let $\hat{\pi} = \arg\max_a \hat{Q}(s, a)$. be the greedy policy with respect to our Q value estimates. Then, we have that

$$
\begin{aligned}
\hat{V}(s) &= E[\max(\hat{Q}(s, a_1), \hat{Q}(s, a_2)] \\
&\geq \max(E[\hat{Q}(s, a_1)], E[\hat{Q}(s, a_2)]) && \text{by Jensen's inequality} \\
&= \max(0, 0) && \text{since each estimate is unbiased and } Q(s, \cdot) = 0 \\
&= 0 = V^*(s)
\end{aligned}
$$

Thus, our state value estimate is at least as large as the true value of state s, so we are systematically overestimating the value of the state in the presence of finite samples.

6.3 Double Q-Learning

As we saw in the previous subsection, the state values can suffer from maximization bias as well when we have finitely many samples. As we discussed in the coin example, decoupling taking the max and estimating the value of the max can get rid of this bias. In Q-learning, we can maintain two independent unbiased estimates, Q_1 and Q_2 and when we use one to select the maximum, we can use the other to get an estimate of the value of this maximum. This gives rise to **double Q-learning** which is detailed in algorithm 6. When we refer to the ϵ-greedy policy with respect to $Q_1 + Q_2$ we mean the ϵ-greedy policy where the optimal action at state s is equal to $\arg\max_a Q_1(s, a) + Q_2(s, a)$.

Algorithm 6 Double Q-Learning

procedure DOUBLE Q-LEARNING(ϵ, α, γ)
  Initialize $Q_1(s,a), Q_2(s,a)$ for all $s \in S, a \in A$, set $t \leftarrow 0$
  $\pi \leftarrow \epsilon$-greedy policy with respect to $Q_1 + Q_2$
  loop
    Sample action a_t from policy π at state s_t
    Take action a_t and observe reward r_t and next state s_{t+1}
    if (with 0.5 probability) **then**
      $Q_1(s_t, a_t) \leftarrow Q_1(s_t, a_t) + \alpha(r_t + \gamma Q_2(s_{t+1}, \arg\max_{a'} Q_1(s_{t+1}, a')) - Q_1(s_t, a_t))$
    else
      $Q_2(s_t, a_t) \leftarrow Q_2(s_t, a_t) + \alpha(r_t + \gamma Q_1(s_{t+1}, \arg\max_{a'} Q_2(s_{t+1}, a')) - Q_2(s_t, a_t))$
    $\pi \leftarrow \epsilon$-greedy policy with respect to $Q_1 + Q_2$ (policy improvement)
    $t \leftarrow t + 1$
  return $\pi, Q_1 + Q_2$

Double Q-learning can significantly speed up training time by eliminating suboptimal actions more quickly than normal Q-learning. Sutton and Barto [1] have a nice example of this in a toy MDP in section 6.7.

References

[1] Sutton, Richard S. and Andrew G. Barto. *Introduction to Reinforcement Learning*. 2nd ed., MIT Press, 2017. Draft. http://incompleteideas.net/book/the-book-2nd.html.

CS234 Notes - Lecture 5
Value Function Approximation

Alex Jin, Emma Brunskill

March 20, 2018

7 Introduction

So far we have represented value function by a *lookup* table where each state has a corresponding entry, $V(s)$, or each state-action pair has an entry, $Q(s, a)$. However, this approach might not generalize well to problems with very large state and/or action spaces, or in other cases we might prefer quickly learning approximations over converged values of each state. A popular approach to address this problem is via function approximation:

$$v_\pi(s) \approx \hat{v}(s, \mathbf{w}) \qquad \text{or} \qquad q_\pi(s, a) \approx \hat{q}(s, a, \mathbf{w})$$

Here $\mathbf{w}$ is usually referred to as the parameter or weights of our function approximator. We list a few popular choices for function approximators:

- Linear combinations of features
- Neural networks
- Decision trees
- Nearest neighbors
- Fourier / wavelet bases

We will further explore two popular classes of **differentiable** function approximators: Linear feature representations and Neural networks. The reason for demanding a differentiable function is covered in the following section.

8 Linear Feature Representations

In linear function representations, we use a feature vector to represent a state:

$$x(s) = [x_1(s)\ x_2(s)\ ...\ x_n(s)]$$

We then approximate our value functions using a linear combination of features:

$$\hat{v}(s, \mathbf{w}) = x(s)^T \mathbf{w} = \sum_{j=1}^{n} x_j(s) \mathbf{w}_j$$

We define the (quadratic) objective (also known as the loss) function to be:

$$J(\mathbf{w}) = \mathbb{E}_\pi \Big[(v_\pi(s) - \hat{v}(s, \mathbf{w}))^2 \Big]$$

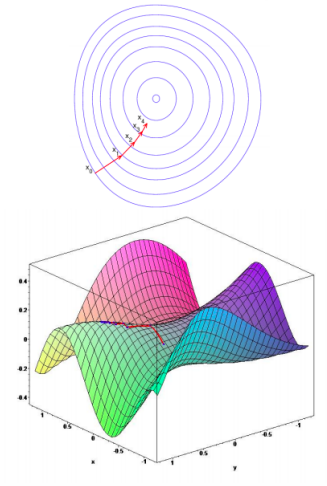


Figure 1: Visualization of Gradient Descent. We wish to reach the center point where our objective function is minimize. We do so by following the red arrows, which points in the negative direction of our evaluated gradient.

8.1 Gradient descent

A common technique to minimize the above objective function is called *Gradient Descent*. Figure 1 provides a visual illustration: we start at some particular spot x_0, corresponding to some initial value of our parameter $\mathbf{w}$; we then evaluate the gradient at x_0, which tells us the direction of the steepest increase in the objective function; to minimize our objective function, we take a step along the negative direction of the gradient vector and arrive at x_1; this process is repeated until we reach some convergence criteria.

Mathematically, this can be summarized as:

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \left[\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_1} \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_2} \dots \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_n} \right] \qquad \text{compute the gradient}$$

$$\Delta \mathbf{w} = -\frac{1}{2} \alpha \nabla_{\mathbf{w}} J(\mathbf{w}) \qquad \text{compute an update step using gradient descent}$$

$$\mathbf{w} \leftarrow \mathbf{w} + \Delta \mathbf{w} \qquad \text{take a step towards the local minimum}$$

8.2 Stochastic gradient descent

In practice, Gradient Descent is not considered a sample efficient optimizer and stochastic gradient descent (**SGD**) is used more often. Although the original SGD algorithm referred to updating the weights using a single sample, due to the convenience of vectorization, people often refer to gradient descent on a minibatch of samples as SGD as well. In (minibatch) SGD, we sample a minibatch of past experiences, compute our objective function on that minibatch, and update our parameters using gradient descent on the minibatch. Let us now go back to several algorithms we covered in previous lectures and see how value function approximations can be incorporated.

8.3 Monte Carlo with linear VFA

Algorithm 1 is a modification of First-Visit Monte Carlo Policy Evaluation, and we replace our value function with our linear VFA. We also make a note that, although our return, $Return(s_t)$, is an

Algorithm 1 Monte Carlo Linear Value Function Approximation for Policy Evaluation

Initialize $\mathbf{w} = 0$, $Returns(s) = 0 \; \forall s$, $k = 1$
loop
  Sample k-th episode $(s_{k1}, a_{k1}, r_{k1}, s_{k2}, \ldots, s_k, L_k)$ given π
  for $t = 1, \ldots, L_k$ **do**
    if first visit to (s) in episode k **then**
      Append $\sum_{j=t}^{L_k} r_{kj}$ to $Return(s_t)$
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha(Return(s_t) - \hat{v}(s_t, \mathbf{w}))x(s_t)$
  $k = k + 1$

unbiased estimate, it is often very noisy.

8.4 Temporal Difference (TD(0)) with linear VFA

Recall that in the tabular setting, we approximate V^π via bootstrapping and sampling and update $V^\pi(s)$ by

$$V^\pi(s) = V^\pi(s) + \alpha(r + \gamma V^\pi(s') - V^\pi(s))$$

where $r + \gamma V^\pi(s')$ represents our *target*. Here we use the same VFA for evaluating both our target and value. In later lectures we will consider techniques for using different VFAs (such as in the control setting).

Using VFAs, we replace V^π with $\hat{V}^\pi$ and our update equation becomes:

$$\begin{aligned}\hat{V}^\pi(s, \mathbf{w}) &= \hat{V}^\pi(s, \mathbf{w}) + \alpha(r + \gamma \hat{V}^\pi(s', \mathbf{w}) - \hat{V}^\pi(s, \mathbf{w}))\nabla_{\mathbf{w}} \hat{V}^\pi(s, \mathbf{w}) \\ &= \hat{V}^\pi(s, \mathbf{w}) + \alpha(r + \gamma \hat{V}^\pi(s', \mathbf{w}) - \hat{V}^\pi(s, \mathbf{w}))\mathbf{x}(s)\end{aligned}$$

In value function approximation, although our target is a biased and approximated estimate of the true value $V^\pi(s)$, linear TD(0) will still converge (close) to the global optimum. We will prove this assertion now.

8.5 Convergence Guarantees for Linear VFA for Policy Evaluation

We define the mean squared error of a linear VFA for a particular policy π relative to the true value as:

$$MSVE(\mathbf{w}) = \sum_{s \in \mathbf{s}} \mathbf{d}(\mathbf{s})(\mathbf{v}^\pi(s) - \hat{\mathbf{v}}^\pi(s, \mathbf{w}))^2$$

where $\mathbf{d}(\mathbf{s})$ is the stationary distribution of states under policy π in the true decision process and $\hat{\mathbf{v}}^\pi(s, \mathbf{w}) = \mathbf{x}(s)^T\mathbf{w}$ is our linear VFA.

Theorem 8.1. *Monte Carlo policy evaluation with VFA converges to the weights $\mathbf{w}_{MC}$ with minimum mean squared error.*

$$MSVE(\mathbf{w}_{MC}) = \min_{\mathbf{w}} \sum_{s \in \mathbf{s}} \mathbf{d}(\mathbf{s})(\mathbf{v}^\pi(s) - \hat{\mathbf{v}}^\pi(s, \mathbf{w}))^2$$

Theorem 8.2. *TD(0) policy evaluation with VFA converges to the weights $\mathbf{w}_{TD}$ which is within a constant factor of the minimum mean squared error.*

$$MSVE(\mathbf{w}_{MC}) = \frac{1}{1-\gamma} \min_{\mathbf{w}} \sum_{s \in \mathbf{s}} \mathbf{d}(\mathbf{s})(\mathbf{v}^\pi(s) - \hat{\mathbf{v}}^\pi(s, \mathbf{w}))^2$$

We omit the proofs here and encourage interested readers to look at [1] for a more in-depth discussion.

Algorithm	Tabular	Linear VFA	Nonlinear VFA
Monte-Carlo Control	Yes	(Yes)	No
SARSA	Yes	(Yes)	No
Q-learning	Yes	No	No

Table 1: Summary of convergence of Control Methods with VFA. (Yes) means the result chatters around near-optimal value function.

8.6 Control using VFA

Similar to VFAs, we can also use function approximators for action-values. That is, we let $\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$. We may then interleave **policy evaluation**, by approximating using $\hat{q}(s, a, \mathbf{w})$, and **policy improvement**, by ϵ-greedy policy improvement. To be more concrete, let's write out this mathematically.

First, we define our objective function $J(\mathbf{w})$ as

$$J(\mathbf{w}) = \mathbb{E}_\pi[(q_\pi(s, a) - \hat{q}^\pi(s, a, \mathbf{w}))^2]$$

Similar to what we did earlier, we may then use either gradient descent or stochastic gradient descent to minimize the objective function. For example, for a linear action-value function approximator, this can be summarized as:

$$\begin{aligned}
x(s, a) &= [x_1(s, a)\ x_2(s, a)\ ...\ x_n(s, a)] && \text{state-action value features} \\
\hat{q}(s, a, \mathbf{w}) &= x(s, a)\mathbf{w} && \text{represent state-action value as linear combinations of features} \\
J(\mathbf{w}) &= \mathbb{E}_\pi[(q_\pi(s, a) - \hat{q}^\pi(s, a, \mathbf{w}))^2] && \text{define objective function} \\
-\frac{1}{2}\nabla_{\mathbf{w}} J(\mathbf{w}) &= \mathbb{E}_\pi[(q_\pi(s, a) - \hat{q}^\pi(s, a, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}^\pi(s, a, \mathbf{w})] && \text{compute the gradient} \\
&= (q_\pi(s, a) - \hat{q}^\pi(s, a, \mathbf{w}))x(s, a) && \\
\Delta\mathbf{w} &= -\frac{1}{2}\alpha\nabla_{\mathbf{w}} J(\mathbf{w}) && \text{compute an update step using gradient descent} \\
&= \alpha(q_\pi(s, a) - \hat{q}^\pi(s, a, \mathbf{w}))x(s, a) && \\
\mathbf{w} &\leftarrow \mathbf{w} + \Delta\mathbf{w} && \text{take a step towards the local minimum}
\end{aligned}$$

For Monte Carlo methods, we substitute our target $q_\pi(s, a)$ with a return G_t. That is, our update becomes:

$$\Delta\mathbf{w} = \alpha(G_t - \hat{q}(s, a, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}(s, a, \mathbf{w})$$

For SARSA, we substitute our target with a TD target:

$$\Delta\mathbf{w} = \alpha(r + \gamma\hat{q}(s', a', \mathbf{w}) - \hat{q}(s, a, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}(s, a, \mathbf{w})$$

For Q-learning, we substitute our target with a max TD target:

$$\Delta\mathbf{w} = \alpha(r + \gamma \max_{a'} \hat{q}(s', a', \mathbf{w}) - \hat{q}(s, a, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}(s, a, \mathbf{w})$$

We note that because our use of value function approximations, which can be expansions, to carry out our Bellman backup, convergence is not guaranteed. We refer users to look for Baird's Counterexample for a more concrete illustration. We summarize contraction guarantees in the table 1.

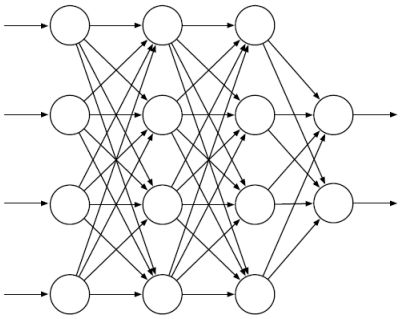


Figure 2: A generic feedforward neural network with four input units, two output units, and two hidden layers.

9 Neural Networks

Although linear VFAs often work well given the right set of features, it can also be difficult to hand-craft such feature set. Neural Networks provide a much richer function approximation class that is able to directly go from states without requiring an explicit specification of features.

Figure 2 shows a generic feedforward ANN. The network in the figure has an output layer consisting of two output units, an input layer with four input units, and two hidden layers: layers that are neither input nor output layers. A real-valued weight is associated with each link. The units are typically semi-linear, meaning that they compute a weighted sum of their input signals and then apply a nonlinear function to the result. This is usually referred to as activation function. In assignment 2, we studied how neural networks with a single hidden layer can have the "universal approximation" property, both experience and theory show that approximating the complex functions needed is made much easier with a hierarchical composition of multiple hidden layers.

In the next lecture, we will take a closer look at some theory and recent results of neural networks being applied to solve reinforcement learning problems.

References

[1] Tsitsiklis and Van Roy. An Analysis of Temporal-Difference Learning with Function Approximation. 1997.

CS234 Notes - Lecture 6
CNNs and Deep Q Learning

Tian Tan, Emma Brunskill

March 20, 2018

7 Value-Based Deep Reinforcement Learning

In this section, we introduce three popular value-based deep reinforcement learning (RL) algorithms: **Deep Q-Network (DQN)** [1], **Double DQN** [2] and **Dueling DQN** [3]. All the three neural architectures are able to learn successful policies directly from *high-dimensional inputs*, e.g. preprocessed pixels from video games, by using *end-to-end* reinforcement learning, and they all achieved a level of performance that is comparable to a professional human games tester across a set of 49 names on Atari 2600 [4].

Convolutional Neural Networks (CNNS) [5] are used in these architectures for feature extraction from pixel inputs. Understanding the mechanisms behind feature extraction via CNNs can help better understand how DQN works. The Stanford CS231N course website contains wonderful examples and introduction to CNNs. Here, we direct the reader to the following link for more details on CNNs: http://cs231n.github.io/convolutional-networks/. The remaining of this section will focus on generalization in RL and value-based deep RL algorithms.

7.1 Recap: Action-Value Function Approximation

In the previous lecture, we use parameterized function approximators to represent the action-value function (a.k.s. Q-function). If we denote the set of parameters as $\mathbf{w}$, the Q-function in this *approximation setting* is represented as $\hat{q}(s, a, \mathbf{w})$.

Let's first assume we have access to an oracle $q(s, a)$, the approximate Q-function can be learned by minimizing the mean-squared error between the true action-value function $q(s, a)$ and its approximated estimates,

$$J(\mathbf{w}) = \mathbb{E}[(q(s, a) - \hat{q}(s, a, \mathbf{w}))^2] \tag{1}$$

We can use *stochastic gradient descent (SGD)* to find a local minimum of J by sampling gradients w.r.t. parameters $\mathbf{w}$ and updating $\mathbf{w}$ as follows:

$$\Delta(\mathbf{w}) = -\frac{1}{2}\alpha\nabla_{\mathbf{w}}J(\mathbf{w}) = \alpha\mathbb{E}[(q(s, a) - \hat{q}(s, a, \mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(s, a, \mathbf{w})] \tag{2}$$

where α is the learning rate. In general, the true action-value function $q(s, a)$ is *unknown*, so we substitute the $q(s, a)$ in Equation (2) with an approximate *learning target*.

In Monte Carlo methods, we use an unbiased return G_t as the substitute target for episodic MDPs:

$$\Delta(\mathbf{w}) = \alpha(G_t - \hat{q}(s, a, \mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(s, a, \mathbf{w}) \tag{3}$$

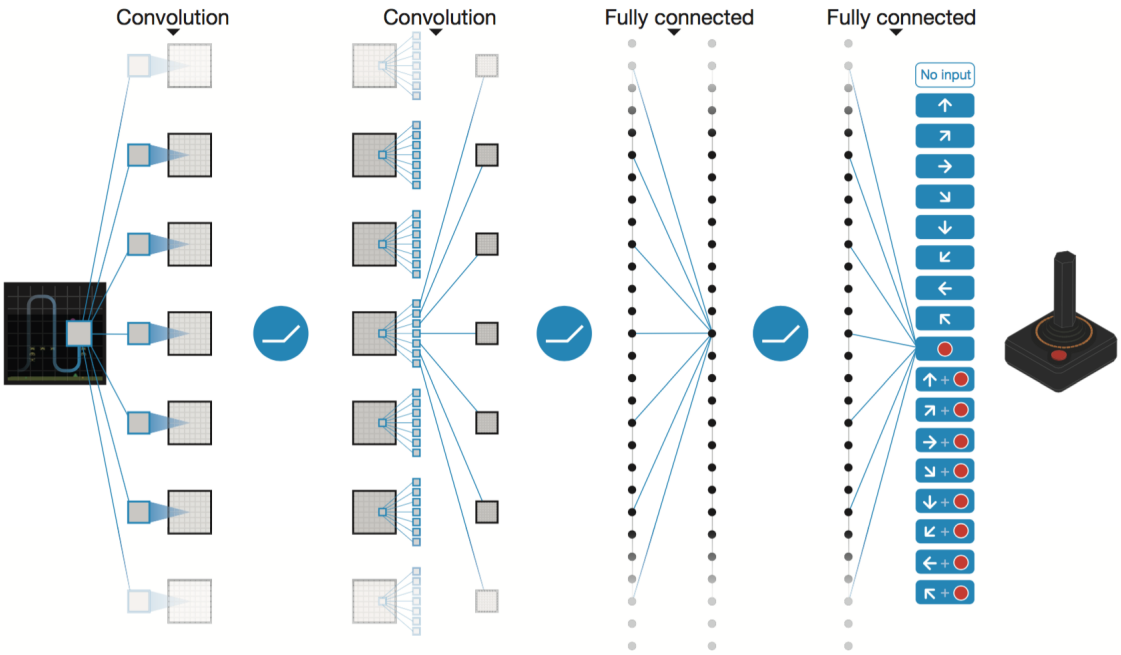


Figure 1: **Illustration of the Deep Q-network**: the input to the network consists of an $84 \times 84 \times 4$ preprocessed image, followed by three convolutional layers and two fully connected layers with a single output for each valid action. Each hidden layer is followed by a rectifier nonlinearity (ReLU) [6].

For SARSA, we instead use *bootstrapping* and present a TD (biased) target $r + \gamma \hat{q}(s', a', \mathbf{w})$, which leverages the current function approximation value,

$$\Delta(\mathbf{w}) = \alpha(r + \gamma \hat{q}(s', a', \mathbf{w}) - \hat{q}(s, a, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}(s, a, \mathbf{w}) \tag{4}$$

where a' is the action taken at the next state s' and γ is a discount factor. For Q-learning, we use a TD target $r + \gamma \max_{a'} \hat{q}(s', a', \mathbf{w})$ and update $\mathbf{w}$ as follows:

$$\Delta(\mathbf{w}) = \alpha(r + \gamma \max_{a'} \hat{q}(s', a', \mathbf{w}) - \hat{q}(s, a, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}(s, a, \mathbf{w}) \tag{5}$$

In subsequent sections, we will introduce how to approximate $\hat{q}(s, a, \mathbf{w})$ by using a deep neural network and learn neural network parameters $\mathbf{w}$ via end-to-end training.

7.2 Generalization: Deep Q-Network (DQN) [1]

The performance of linear function approximators highly depends on the quality of features. In general, handcrafting an appropriate set of features can be difficult and time-consuming. To scale up to making decisions in really *large domains* (e.g. huge state space) and enable automatic feature extraction, deep neural networks (DNNs) are used as function approximators.

7.2.1 DQN Architecture

An illustration of the DQN architecture is shown in Figure 1. The network takes preprocessed pixel image from Atari game environment (see 7.2.2 for preprocessing) as inputs, and outputs a vector containing Q-values for each valid action. The preprocessed pixel input is a summary of the game state s, and a single output unit represents the $\hat{q}$ function for a single action a. Collectively, the

$\hat{q}$ function can be denoted as $\hat{q}(s, \mathbf{w}) \in \mathbb{R}^{|A|}$. For simplicity, we will still use notation $\hat{q}(s, a, \mathbf{w})$ to represent the estimated action-value for a (s, a) pair in the following paragraphs.

Details of the architecture: the input consists of an $84 \times 84 \times 4$ image. The first convolutional layer has 32 filters of size 8×8 with stride 4 and convolves with the input image, followed by a rectifier nonlinearity (ReLU) [6]. The second hidden layer convolves 64 filters of 4×4 with stride 2, again followed by a rectifier nonlinearity. This is followed by a third convolutional layer that has 64 filters of 3×3 with stride 1, followed by a ReLU. The final hidden layer is a fully-connected layer with 512 rectifier (ReLU) units. The output layer is a fully-connected *linear* layer.

7.2.2 Preprocessing Raw Pixels

The raw Atari 2600 frames are of size $(210 \times 160 \times 3)$, where the last dimension is corresponding to the RGB channels. The preprocessing step adopted in [1] aims at reducing the input dimensionality and dealing with some artifacts of the game emulator. We summarize the preprocessing as follows:

- single frame encoding: to encode a single frame, the maximum value for each pixel color value over the frame being encoded and the previous frame is returned. In other words, we return a pixel-wise max-pooling of the 2 consecutive raw pixel frames.
- dimensionality reduction: extract the Y channel, also known as luminance, from the *encoded* RGB frame and rescale it to $(84 \times 84 \times 1)$.

The above preprocessing is applied to the 4 most recent raw RGB frames and the encoded frames are stacked together to produce the input (of shape $(84 \times 84 \times 4)$) to the Q-Network. Stacking together the recent frames as game state is also a way to transform the game environment into a (almost) Markovian world.

7.2.3 Training Algorithm for DQN

The use of large deep neural network function approximators for learning action-value functions has often been avoided in the past since theoretical performance guarantees are impossible, and learning and training tend to be very unstable. In order to use large nonlinear function approximators and scale online Q-learning, DQN introduced two major changes: the use of *experience replay*, and a separate *target network*. The full algorithm is presented in Algorithm 1. Essentially, the Q-network is learned by minimizing the following mean squared error:

$$J(\mathbf{w}) = \mathbb{E}_{(s_t, a_t, r_t, s_{t+1})}[(y_t^{DQN} - \hat{q}(s_t, a_t, \mathbf{w}))^2] \tag{6}$$

where y_t^{DQN} is the one-step ahead learning target:

$$y_t^{DQN} = r_t + \gamma \max_{a'} \hat{q}(s_{t+1}, a', \mathbf{w}^-) \tag{7}$$

where $\mathbf{w}^-$ represents the parameters of the target network, and the parameters $\mathbf{w}$ of the online network are updated by sampling gradients from minibatches of past transition tuples (s_t, a_t, r_t, s_{t+1}).
(*Note*: although the learning target is computed from the target network with $\mathbf{w}^-$, the targets y_t^{DQN} are considered to be fixed when making updates to $\mathbf{w}$.)

Experience replay:

The agent's experiences (or transitions) at each time step $e_t = (s_t, a_t, r_t, s_{t+1})$ are stored in a fixed-sized dataset (or *replay buffer*) $D_t = \{e_1, ..., e_t\}$. The replay buffer is used to store the most recent $k = 1$ million experiences (see Figure 2 for an illustration of replay buffer). The Q-network is updated by SGD with sampled gradients from minibatch data. Each transition sample in the minibatch is

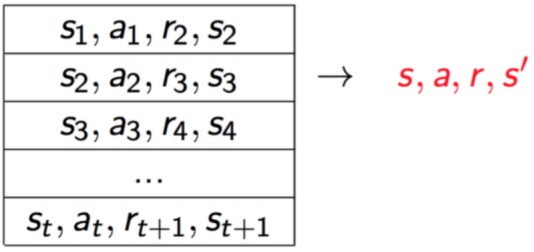


Figure 2: **Illustration of replay buffer**: the transition (s, a, r, s') is uniformly sampled from the replay buffer for updating Q-network.

sampled uniformly at random from the pool of stored experiences, $(s, a, r, s') \sim U(D)$. This approach has the following *advantages* over standard online Q-learning:

- Greater data efficiency: each step of experience can be potentially used for many updates, which improves data efficiency.
- Remove sample correlations: randomizing the transition experiences breaks the correlations between consecutive samples and therefore reduces the variance of updates and stabilizes the learning.
- Avoiding oscillations or divergence: the behavior distribution is averaged over many of its previous states and transitions, smoothing out learning and avoiding oscillations or divergence in the parameters. (Note that when using experience replay, it is required to use off-policy method, e.g. Q-learning, because the current parameters are different from those used to generate the samples).

limitation of experience replay: the replay buffer does not differentiate important transitions or informative transitions and it always overwrites with the recent transitions due to fixed buffer size. Similarly, the uniform sampling from the buffer gives equal importance to all stored experiences. A more sophisticated replay strategy, Prioritized Replay, has been proposed in [7], which replays important transitions more frequently, and therefore the agent learns more efficiently.

Target network:

To further improve the stability of learning and deal with *non-stationary learning targets*, a separate target network is used for generating the targets y_j (line 12 in Algorithm 1) in the Q-learning update. More specifically, every C updates/steps the target network $\hat{q}(s, a, \mathbf{w}^-)$ is updated by copying the parameters' values ($\mathbf{w}^- = \mathbf{w}$) from the online network $\hat{q}(s, a, \mathbf{w})$, and the target network remains unchanged and generates targets y_j for the following C updates. This modification makes the algorithm more stable compared to standard online Q-learning, and $C = 10000$ in the original DQN.

7.2.4 Training Details

In the original DQN paper [1], a different network (or agent) was trained on each game with the same architecture, learning algorithm and hyperparameters. The authors clipped all positive rewards from the game environment at $+1$ and all negative rewards at -1, which makes it possible to use the same learning rate across all different games. For games where there is a life counter (e.g. *Breakout*), the emulator also returns the number of lives left in the game, which was then used to mark the end of an episode during training by explicitly setting future rewards to zeros. They also used a simple frame-skipping technique (or *action repeat*): the agent selects actions on every 4-th frame instead of every frame, and its last action is repeated on skipped frames. This reduces the frequency of decisions

Algorithm 1 deep Q-learning

1: Initialize replay memory D with a fixed capacity
2: Initialize action-value function $\hat{q}$ with random weights $\mathbf{w}$
3: Initialize target action-value function $\hat{q}$ with weights $\mathbf{w}^- = \mathbf{w}$
4: **for** episode $m = 1, \ldots, M$ **do**
5: Observe initial frame x_1 and preprocess frame to get state s_1
6: **for** time step $t = 1, \ldots, T$ **do**
7: Select action $a_t = \begin{cases} \text{random action} & \text{with probability } \epsilon \\ \arg\max_a \hat{q}(s_t, a, \mathbf{w}) & \text{otherwise} \end{cases}$
8: Execute action a_t in emulator and observe reward r_t and image x_{t+1}
9: Preprocess s_t, x_{t+1} to get s_{t+1}, and store transition (s_t, a_t, r_t, s_{t+1}) in D
10: Sample uniformly a random minibatch of N transitions (s_j, a_j, r_j, s_{j+1}) from D
11: Set $y_j = r_j$ if episode ends at step $j + 1$;
12: otherwise set $y_j = r_j + \gamma \max_{a'} \hat{q}(s_{j+1}, a', \mathbf{w}^-)$
13: Perform a stochastic gradient descent step on $J(\mathbf{w}) = \frac{1}{N} \sum_{j=1}^{N} (y_j - \hat{q}(s_j, a_j, \mathbf{w}))^2$ w.r.t. parameters $\mathbf{w}$
14: Every C steps reset $\mathbf{w}^- = \mathbf{w}$

without impacting the performance too much and enables the agent to play roughly 4 times more games during training.

RMSProp (see https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf) was used in [1] for training DQN with minibatches of size 32. During training, they applied ϵ-greedy policy with ϵ linearly annealed from 1.0 to 0.1 over the first million steps, and fixed at 0.1 afterwards. The replay buffer was used to store the most recent 1 million transitions. For evaluation at test time, they used ϵ-greedy policy with $\epsilon = 0.05$.

7.3 Reducing Bias: Double Deep Q-Network (DDQN) [2]

The max operator in DQN (line 12 of Algorithm 1), uses the same network values both to select and to evaluate an action. This setting makes it more likely to select overestimated values and resulting in overoptimistic target value estimates. Van Hasselt et al. also showed in [2] that the DQN algorithm suffers from substantial overestimations in some games in the Atari 2600. To prevent overestimation and reduce bias, we can *decouple* the *action selection* from *action evaluation*.

Recall in Double Q-learning, two action-value functions are maintained and learned by randomly assigning transitions to update one of the two functions, resulting in two different sets of function parameters, denoted here as $\mathbf{w}$ and $\mathbf{w}'$. For computing targets, one function is used to select the greedy action and the other to evaluate its value:

$$y_t^{DoubleQ} = r_t + \gamma \hat{q}(s_{t+1}, \arg\max_{a'} \hat{q}(s_{t+1}, a', \mathbf{w}), \mathbf{w}') \tag{8}$$

Note that the action selection (*argmax*) is due to the function parameters $\mathbf{w}$, while the action value is evaluated by the other set of parameters $\mathbf{w}'$.

The idea of reducing overestimations by decoupling action selection and action evaluation in computing targets can also be extended to deep Q-learning. The target network in DQN architecture provides a natural candidate for the second action-value function, without introducing additional networks. Similarly, the greedy action is generated according to the online network with parameters $\mathbf{w}$, but its value is estimated by the target network with parameters $\mathbf{w}^-$. The resulting algorithm is referred as *Double DQN* [2], which just replaces line 12 in Algorithm 1 by the following update target:

$$y_t^{DoubleDQN} = r_t + \gamma \hat{q}(s_{t+1}, \arg\max_{a'} \hat{q}(s_{t+1}, a', \mathbf{w}), \mathbf{w}^-) \tag{9}$$

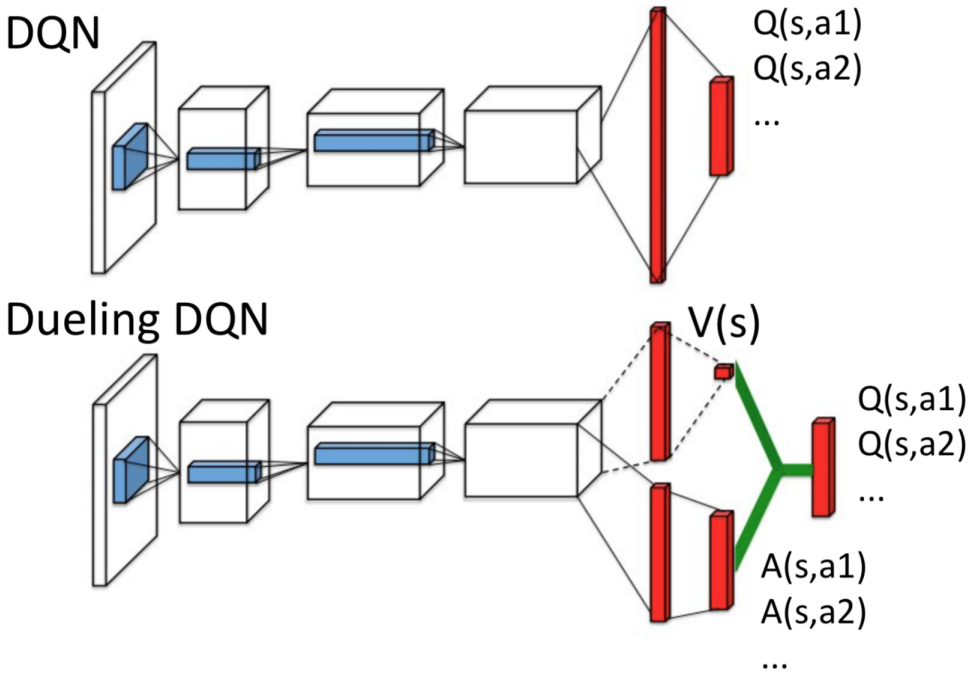


Figure 3: Single stream Deep Q-network (top) and the dueling Q-network (bottom). The dueling network has two streams to separately estimate (scalar) state-value $V(s)$ and the advantages $A(s,a)$ for each action; the green output module implements equation (13) to combine the two streams. Both networks output Q-values for each action.

The update to the target network stays unchanged from DQN, and remains a periodic copy of the online network $\mathbf{w}$. The rest of the DQN algorithm remains intact.

7.4 Decoupling Value and Advantage: Dueling Network [3]

7.4.1 The Dueling Network Architecture

Before we delve into dueling architecture, let's first introduce an important quantity, the *advantage function*, which relates the value and Q functions (assume following a policy π):

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s) \tag{10}$$

Recall $V^\pi(s) = \mathbb{E}_{a\sim\pi(s)}[Q^\pi(s,a)]$, thus we have $\mathbb{E}_{a\sim\pi(s)}[A^\pi(s,a)] = 0$. Intuitively, the advantage function subtracts the value of the state from the Q function to get a relative measure of the importance of each action.

Like in DQN, the dueling network is also a DNN function approximator for learning the Q-function. Differently, it approximates the Q-function by *decoupling* the value function and the advantage function. Figure 3 illustrates the dueling network architecture and the DQN for comparison.

The lower layers of the dueling network are convolutional as in the DQN. However, instead of using a single stream of fully connected layers for Q-value estimates, the dueling network uses two streams of fully connected layers. One stream is used to provide value function estimate given a state, while the other stream is for estimating advantage function for each valid action. Finally, the two streams

are combined in a way to produce and approximate the Q-function. As in DQN, the output of the network is a vector of Q-values, one for each action.

Note that since the inputs and the final outputs (combined two streams) of the dueling network are the same as that of the original DQN, the training algorithm (Algorithm 1) introduced above for DQN and for Double DQN can also be applied here to train the dueling architecture. The separated two-stream design is based on the following observations or intuitions from the authors:

- For many states, it is unnecessary to estimate the value of each possible action choice. In some states, the action selection can be of great importance, but in many other states the choice of action has no repercussion on what happens next. On the other hand, the state value estimation is of significant importance for every state for a bootstrapping based algorithm like Q-learning.
- Features required to determine the value function may be different than those used to accurately estimate action benefits.

Combing the two streams of fully connected layers for Q-value estimate is not a trivial task. This aggregating module (shown in *green* lines in Figure 3), in fact, requires very thoughtful design, which we will see in the next subsection.

7.4.2 Q-value Estimation

From the definition of advantage function (10), we have $Q^\pi(s,a) = A^\pi(s,a) + V^\pi(s)$, and $\mathbb{E}_{a\sim\pi(s)}[A^\pi(s,a)] = 0$. Furthermore, for a deterministic policy (commonly used in value-based deep RL), $a^* = \arg\max_{a'\in A} Q(s,a')$, it follows that $Q(s,a^*) = V(s)$ and hence $A(s,a^*) = 0$. The greedily selected action has zero advantage in this case.

Now consider the dueling network architecture in Figure 3 for function approximation. Let's denote the scalar output value function from one stream of the fully-connected layers as $\hat{v}(s,\mathbf{w},\mathbf{w_v})$, and denote the vector output advantage function from the other stream as $A(s,a,\mathbf{w},\mathbf{w_A})$. We use $\mathbf{w}$ here to denote the shared parameters in the convolutional layers, and use $\mathbf{w_v}$ and $\mathbf{w_A}$ to represent parameters in the two different streams of fully-connected layers. Then, probably the most simple way to design the aggregating module is by following the definition:

$$\hat{q}(s,a,\mathbf{w},\mathbf{w_A},\mathbf{w_v}) = \hat{v}(s,\mathbf{w},\mathbf{w_v}) + A(s,a,\mathbf{w},\mathbf{w_A}) \tag{11}$$

The main problem with this simple design is that Equation (11) is unidentifiable. Given $\hat{q}$, we cannot recover $\hat{v}$ and A uniquely, e.g. adding a constant to $\hat{v}$ and subtracting the same constant from A gives the same Q-value estimates. The unidentifiable issue is mirrored by poor performance in practice.

To make Q-function identifiable, recall in the deterministic policy case discussed above, we can force the advantage function to have zero estimate at the chosen action. Then, we have

$$\hat{q}(s,a,\mathbf{w},\mathbf{w_A},\mathbf{w_v}) = \hat{v}(s,\mathbf{w},\mathbf{w_v}) + \Big(A(s,a,\mathbf{w},\mathbf{w_A}) - \max_{a'\in A} A(s,a',\mathbf{w},\mathbf{w_A})\Big) \tag{12}$$

For a deterministic policy, $a^* = \arg\max_{a'\in A} \hat{q}(s,a',\mathbf{w},\mathbf{w_A},\mathbf{w_v}) = \arg\max_{a'\in A} A(s,a',\mathbf{w},\mathbf{w_A})$, Equation (12) gives $\hat{q}(s,a^*,\mathbf{w},\mathbf{w_A},\mathbf{w_v}) = \hat{v}(s,\mathbf{w},\mathbf{w_v})$. Thus, the stream $\hat{v}$ provides an estimate of the value function, and the other stream A generates advantage estimates.

The authors in [3] also proposed an alternative aggregating module that replaces the max with a mean operator:

$$\hat{q}(s,a,\mathbf{w},\mathbf{w_A},\mathbf{w_v}) = \hat{v}(s,\mathbf{w},\mathbf{w_v}) + \Big(A(s,a,\mathbf{w},\mathbf{w_A}) - \frac{1}{|A|}\sum_{a'} A(s,a',\mathbf{w},\mathbf{w_A})\Big) \tag{13}$$

Although this design in some sense loses the original semantics of $\hat{v}$ and A, the author argued that it improves the stability of learning: the advantages only need to change as fast as the mean, instead of having to compensate any change to the advantage of the optimal action. Therefore, the aggregating module in the dueling network [3] is implemented following Equation (13). When acting, it suffices to evaluate the advantage stream to make decisions.

The advantage of the dueling network lies in its capability of approximating the value function efficiently. This advantage over single-stream Q networks grows when the number of actions is large, and the dueling network achieved state-of-the-art results on Atari games as of 2016.

References

[1] Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature 518.7540 (2015): 529.

[2] Van Hasselt, Hado, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-Learning." AAAI. Vol. 16. 2016.

[3] Wang, Ziyu, et al. "Dueling network architectures for deep reinforcement learning." arXiv preprint arXiv:1511.06581 (2015).

[4] Bellemare, Marc G., et al. "The Arcade Learning Environment: An evaluation platform for general agents." J. Artif. Intell. Res.(JAIR) 47 (2013): 253-279.

[5] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.

[6] Nair, Vinod, and Geoffrey E. Hinton. "Rectified linear units improve restricted boltzmann machines." Proceedings of the 27th international conference on machine learning (ICML-10). 2010.

[7] Schaul, Tom, et al. "Prioritized experience replay." arXiv preprint arXiv:1511.05952 (2015).

CS234 Notes - Lecture 7
Imitation Learning

James Harrison, Emma Brunskill

March 20, 2018

8 Introduction

In reinforcement learning, there are several theoretical and practical hurdles that must be overcome. These include optimization, the effect of delayed consequences, how to do exploration, and how to generalize. Importantly, however, we would like to handle all of the above challenges while also being data efficient and computationally efficient.

We will discuss general approaches to efficient exploration later in the course, for which the techniques are capable of handling general MDPs. However, if we have known structure in the problem, or we have outside knowledge that we can use, we can explore considerably more efficiently. In this lecture, we will talk about how to imitate and learn from human (or expert, generally) behavior on tasks.

9 Imitation Learning

Previously, we have aimed to learn policies from rewards, which are often sparse. For example, a simple reward signal may be whether or not an agent won a game. This approach is successful in situations where data is cheap and easily gathered. This approach fails however, when data gathering is slow, failure must be avoided (e.g. autonomous vehicles), or safety is desired.

One approach to mitigate the sparse reward problem is to manually design reward functions that are dense in time. However, this approach requires a human to hand-design a reward function with the desired behavior in mind. It is therefore desirable to learn by imitating agents performing the task in question.

9.1 Learning from Demonstration

Generally, experts provide a set of demonstration trajectories, which are sequences of states and actions. More formally, we assume we are given:

- State space, action space
- Transition model $P(s' \mid s, a)$
- No reward function, R
- Set of one or more teacher demonstrations $(s_0, a_0, s_1, a_1, \ldots)$, where actions are drawn from the teacher's policy π^*

10 Behavioral Cloning

Can we learn the teacher's policy using supervised learning?

In behavioral cloning, we aim simply to learn the policy via supervised learning. Specifically, we will fix a policy class and aim to learn a policy mapping states to actions given the data tuples $\{(s_0, a_0), (s_1, a_1), \ldots\}$. One notable example of this is ALVINN, which learned to map from sensor inputs to steering angles.

One challenge to this approach is that data is not distributed i.i.d. in the state space. This i.i.d. assumption is standard in the supervised learning literature and theory. However, in the RL context, errors are compounding; they accumulate over the length of the episode. The training data for the learned policy will be tightly clustered around expert trajectories. If a mistake is made that puts the agent in a part of the state space that the expert did not visit, the agent has no data to learn a policy from. In this case, the error scales quadratically in the episode length, as opposed to the linear scaling in standard RL.

10.1 DAGGER: Dataset Aggregation

Dataset Aggregation (DAGGER, algorithm 1, [1]) aims to mitigate the problem of compounding errors by adding data for newly visited states. As opposed to assuming there is a pre-defined set of expert demonstrations, we assume that we can generate more data from an expert. The limitation of this, of course, is that an expert must be available to provide labels, sometimes in real time.

Algorithm 1 DAGGER

Initialize $\mathcal{D} \leftarrow \emptyset$
Initialize $\hat{\pi}_1$ to any policy in Π
for $i = 1$ to N **do**
    Let $\pi_i = \beta_i \pi^* + (1 - \beta_i)\hat{\pi}_i$
    Sample T-step trajectories using π_i
    Get dataset $\mathcal{D}_i = \{(s, \pi^*(s))\}$ of visited states by π_i and actions given by expert
    Aggregate datasets: $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_i$
    Train classifier $\hat{\pi}_{i+1}$ on $\mathcal{D}$
return best $\hat{\pi}_i$ on validation

11 Inverse Reinforcement Learning (IRL)

Can we recover the reward function, R?

In inverse reinforcement learning (also referred to as inverse optimal control), the goal is to learn the reward function (that has not been provided) based on the expert demonstrations. Without assumptions on the optimality of the demonstrations, this problem is intractable as any arbitrary reward function may give rise to the observed trajectories.

11.1 Linear Feature Reward Inverse RL

We consider a reward which is represented as a linear combination of features

$$R(s) = w^T x(s) \tag{1}$$

where $w \in \mathbb{R}^n, x : S \to \mathbb{R}^n$. In this case, the IRL problem is to identify the weight vector w, given a set of demonstrations. The resulting value function for a policy π can be expressed as

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \mid s_0 = s \right] \tag{2}$$

$$= \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t w^T x(s_t) \mid s_0 = s \right] \tag{3}$$

$$= w^T \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t x(s_t) \mid s_0 = s \right] \tag{4}$$

$$= w^T \mu(\pi), \tag{5}$$

where $\mu(\pi \mid s_0 = s) \in \mathbb{R}^n$ is the discounted weighted frequency of state features $x(s)$ under policy π. Note that

$$\mathbb{E}_{\pi^*} \left[\sum_{t=0}^{\infty} \gamma^t R^*(s_t) \mid s_0 = s \right] \geq \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R^*(s_t) \mid s_0 = s \right] \quad \forall \pi, \tag{6}$$

where R^* denotes an optimal reward function. Thus, if an expert's demonstrations are optimal (i.e. actions are drawn from an optimal policy), to identify w it is sufficient to find some w^* such that

$$w^{*T} \mu(\pi^* \mid s_0 = s) \geq w^{*T} \mu(\pi \mid s_0 = s) \quad \forall \pi, \forall s. \tag{7}$$

In a sense, we want to find a parameterization of the reward function such that the expert policy outperforms other policies.

12 Apprenticeship Learning

Can we use the recovered reward to generate a good policy?

For a policy π to perform as well as the expert policy π^*, it suffices that we have a policy such that its discounted, summed feature expectations match the expert's policy [2]. More precisely, if

$$\|\mu(\pi \mid s_0 = s) - \mu(\pi^* \mid s_0 = s)\|_1 \leq \epsilon \tag{8}$$

then for all w with $\|w\|_\infty \leq 1$,

$$|w^T \mu(\pi \mid s_0 = s) - w^T \mu(\pi^* \mid s_0 = s)| \leq \epsilon$$

by the Cauchy-Schwartz inequality. This observation leads to Algorithm 2 for learning a policy that is as good as the expert policy (see [2] for details).

In practice, there are challenges associated with this approach:

- If the expert policy is suboptimal, than the resulting policy is a mixture of somewhat arbitrary policies that have the expert policy in their convex hull. In practice, a practitioner can pick the best policy in this set and pick the corresponding reward function.
- This approach relies on being able to compute an optimal policy given a reward function, which may be expensive or impossible.
- There is an infinite number of reward functions with the same optimal policy, and an infinite number of stochastic policies that can match feature counts.

Algorithm 2 Apprenticeship Learning via Linear Feature IRL

1: Initialize policy π_0
2: **for** $i = 1, 2, \ldots$ **do**
3: Find reward function weights w such that the teacher maximally outperforms all previous controllers:

$$\begin{aligned} &\arg\max_{w} \max_{\gamma} \gamma \\ &\text{s.t. } w^T \mu(\pi^* \mid s_0 = s) \geq w^T \mu(\pi \mid s_0 = s) + \gamma, \ \forall \pi \in \{\pi_0, \pi_1, \ldots, \pi_{i-1}\}, \forall s \\ &\quad \|w\|_2 \leq 1 \end{aligned}$$

4: Find optimal policy π_i for current w
5: **if** $\gamma \leq \epsilon/2$ **then return** π_i

12.1 Maximum Entropy Inverse RL

To address the problem of ambiguity, Ziebart el al. [3] introduced Maximum Entropy (MaxEnt) IRL. Consider the collection of all possible H-step trajectories in a deterministic MDP. For a linear reward model, a policy is completely specified by its distribution over trajectories. Given this, which policy should we choose given a set of m distributions?

Again, assume the reward function is a linear function of the features, $R(s) = w^T x(s)$. Denoting trajectory j as τ_j, we can write the feature counts for this trajectory as

$$\mu_{\tau_j} = \sum_{s_i \in \tau_j} x(s_i). \tag{9}$$

Averaging over m trajectories, we can write the average feature counts

$$\tilde{\mu} = \frac{1}{m} \sum_{j=1}^{m} \mu_{\tau_j}. \tag{10}$$

The Principle of Maximum Entropy [4] motivates choosing a distribution with no additional preferences beyond matching the feature expectations in the demonstration dataset

$$\max_P - \sum_\tau P(\tau) \log P(\tau) \tag{11}$$

$$s.t. \ \sum_\tau P(\tau) \mu_\tau = \tilde{\mu} \tag{12}$$

$$\sum_\tau P(\tau) = 1. \tag{13}$$

In the case of linear rewards, this is equivalent to specifying the weights w that yield a policy with the maximum entropy, constrained to matching the feature expectations. Maximizing the entropy of the distribution over the paths subject to the feature constraints from observed data implies we maximize the likelihood of the observed data under the maximum entropy (exponential family) distribution

$$P(\tau_j \mid w) = \frac{1}{Z(w)} \exp\left(w^T \mu_{\tau_j}\right) = \frac{1}{Z(w)} \exp\left(\sum_{s_i \in \tau_j} w^T x(s_i)\right),$$

with

$$Z(w, s) = \sum_{\tau_s} \exp\left(w^T \mu_{\tau_s}\right).$$

This induces a strong preference for low cost paths, and equal cost paths are equally probable. Many MDPs of interest are stochastic. In these cases, the distribution over paths depends on both the reward weights and on the dynamics

$$P(\tau_j \mid w, P(s'|s,a)) \approx \frac{\exp\left(w^T \mu_{\tau_j}\right)}{Z(w, P(s'|s,a))} \prod_{s_i, a_i \in \tau_j} P(s_{i+1}|s_i, a_i).$$

The weights w are learned by maximizing the likelihood of the data

$$w^* = \arg\max_w L(w) = \arg\max_w \sum_{\text{examples}} \log P(\tau \mid w).$$

The gradient is the difference between expected empirical feature counts and the learner's expected feature counts, which can be expressed in terms of expected state visitation frequencies

$$\nabla L(w) = \tilde{\mu} - \sum_\tau P(\tau \mid w)\mu_\tau = \tilde{\mu} - \sum_{s_i} D(s_i)x(s_i),$$

where $D(s_i)$ denotes the state visitation frequency. This approach has been hugely influential. It provides a principled way to select among the many possible reward functions. However, the original formulation of the algorithm requires knowledge of the transition model or the ability to simulate/act in the world to gather samples of the transition model.

Algorithm 3 Maximum Entropy IRL

Backward pass

1: Set $Z_{s_i,0} = 0$

2: Recursively compute for N iterations

$$Z_{a_{i,j}} = \sum_k P(s_k \mid s_i, a_{i,j}) \exp(R(s_i \mid w)) Z_{s_k} \tag{14}$$

$$Z_{s_i} = \sum_{a_{i,j}} Z_{a_{i,j}} \tag{15}$$

Local action probability computation

3: $P(a_i, j \mid s_i) = \frac{Z_{a_{i,j}}}{Z_{s_i}}$

Forward pass

4: Set $D_{s_i,t} = P(s_i = s_{\text{initial}})$

5: Recursively compute for $t = 1$ to N

$$D_{s_i,t+1} = \sum_{a_{i,j}} \sum_k D_{s_k,t} P(a_{i,j} \mid s_i) P(s_k \mid a_{i,j} s_i) \tag{16}$$

Summing frequencies

6: $D_{s_i} = \sum_t D_{s_i,t}$

References

[1] Ross, Stéphane, Geoffrey Gordon, and Drew Bagnell. "A reduction of imitation learning and structured prediction to no-regret online learning." Proceedings of the fourteenth international conference on artificial intelligence and statistics. 2011.

[2] Abbeel, Pieter, and Andrew Y. Ng. "Apprenticeship learning via inverse reinforcement learning." Proceedings of the twenty-first international conference on Machine learning. 2004.

[3] Ziebart, Brian D., et al. "Maximum Entropy Inverse Reinforcement Learning." AAAI. Vol. 8. 2008.

[4] Jaynes, Edwin T. "Information theory and statistical mechanics." Physical review 106.4 (1957): 620.

CS234 Notes - Lecture 8 & 9
Policy Gradient

Luke Johnston, Emma Brunskill

March 20, 2018

1 Introduction to Policy Search

So far, in order to learn a policy, we have focused on **value-based** approaches where we find the optimal state value function or state-action value function with parameters θ,

$$V_\theta(s) \approx V^\pi(s)$$

$$Q_\theta(s,a) \approx Q^\pi(s,a)$$

and then use V_θ or Q_θ to extract a policy, e.g. with ϵ-greedy. However, we can also use a **policy-based** approach to directly parameterize the policy:

$$\pi_\theta(a|s) = \mathbb{P}[a|s;\theta]$$

In this setting, our goal is to directly find the policy with the highest value function V^π, rather than first finding the value-function of the optimal policy and then extracting the policy from it. Instead of the policy being a look-up table from states to actions, we will consider stochastic policies that are parameterized. Finding a good policy requires two parts:

- Good policy parameterization: our function approximation and state/action representations must be expressive enough
- Effective search: we must be able to find good parameters for our policy function approximation

Policy-based RL has a few advantages over value-based RL:

- Better convergence properties (see Ch 13.3 of Sutton and Barto)
- Effectiveness in high-dimensional or continuous action spaces, e.g. robotics. One method for continuous action spaces is covered in section 6.2.
- Ability to learn stochastic policies. See the following section.

The disadvantages of policy-based RL methods are:

- They typically converge to locally rather than globally optimal policies, since they rely on gradient descent.
- Evaluating a policy is typically data inefficient and high variance.

2 Stochastic policies

In this section, we will briefly go over two environments in which a stochastic policy is better than any deterministic policy.

2.1 Rock-paper-scissors

For a relatable example, in the popular zero-sum game of rock-paper-scissors, any policy that is not uniformly random:

$$P(\texttt{rock}|s) = 1/3$$
$$P(\texttt{scissors}|s) = 1/3$$
$$P(\texttt{paper}|s) = 1/3$$

can be exploited.

2.2 Aliased gridworld

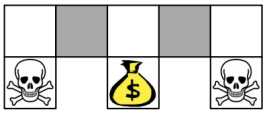


Figure 1: In this partially observable gridworld environment, the agent cannot distinguish between the gray states.

In the above gridworld environment, suppose that the agent can move in the four cardinal directions, so its actions space is $A = \{N, S, E, W\}$. However, suppose that it can only sense the walls around its current location. Specifically, it observes features of the following form for each direction:

$$\phi(s) = \begin{bmatrix} \mathbf{1}(\text{wall to N}) \\ \dots \\ \mathbf{1}(\text{wall to W}) \end{bmatrix}$$

Note that its observations are not fully representative of the environment, as it cannot distinguish between the two gray squares. This also means that its domain is not Markov. Hence, a deterministic policy must either learn to always go left in the gray squares, or always go right. Neither of these policies is optimal, since the agent can get stuck in one corner of the environment:

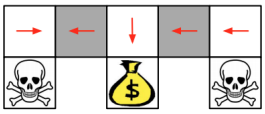


Figure 2: For this deterministic policy, the agent cannot "escape" from the upper-left two states.

However, a stochastic policy can learn to randomly select a direction in the gray states, guaranteeing that it will eventually reach the reward from any starting location. In general, stochastic policies can help overcome an adversarial or non-stationary domain and cases where the state-representation is not Markov.

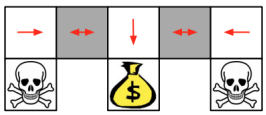


Figure 3: A stochastic policy which moves E or W with equal probability in the gray states will reach the goal in a few time steps with high probability.

3 Policy optimization

3.1 Policy objective functions

Once we have defined a policy $\pi_\theta(a|s)$, we need to able to measure how it is performing in order to optimize it. In an episodic environment, a natural measurement is the **start value** of the policy, which is the expected value of the start state:

$$J_1(\theta) = V^{\pi_\theta}(s_1) = \mathbb{E}_{\pi_\theta}[v_1]$$

In continuing environments we can use the **average value** of the policy, where $d^{\pi_\theta}(s)$ is the stationary distribution of π_θ:

$$J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s)$$

or alternatively we can use the **average reward** per time-step:

$$J_{avR}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(a|s) R(s,a)$$

In these notes we discuss the episodic case, but all the results we derive can be easily extended to the non-episodic case. We will also focus on the case where the discount $\gamma = 1$, but again, the results are easily extended to general γ.

3.2 Optimization methods

With an objective function, we can treat our policy-based reinforcement learning problem as an optimization problem. In these notes, we focus on gradient descent, because recently that has been the most common optimization method for policy-based RL methods. However, it is worth considering some **gradient-free optimization methods**, including the following:

- Hill climbing
- Simplex / amoeba / Nelder Mead
- Genetic algorithms
- Cross-Entropy method (CEM)
- Covariance Matrix Adaptation (CMA)
- Evolution strategies

These methods have the advantage over gradient-based methods in that they do not have to compute a gradient of the objective function. This allows the policy parameterization to be non-differentiable, and these methods are also often easy to parallelize. Gradient-free methods are often a useful baseline to try, and sometimes they can work embarrassingly well [1]. However, this methods are usually not very sample efficient because they ignore the temporal structure of the rewards - updates only take into account the total reward over the entire episode, and they do not break up the reward into different rewards for each state in the trajectory. (See section 4.3).

4 Policy gradient

Let us define $V(\theta)$ to be the objective function we wish to maximize over θ. Policy gradient methods search for a *local* maximum in $V(\theta)$ by ascending the gradient of the policy, w.r.t parameters θ

$$\Delta\theta = \alpha \nabla_\theta V(\theta)$$

where α is a step-size parameter and $\nabla_\theta V(\theta)$ is the policy gradient

$$\nabla_\theta V(\theta) = \begin{pmatrix} \frac{\partial V(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial V(\theta)}{\partial \theta_n} \end{pmatrix}$$

4.1 Computing the gradient

With this setup, all we have to do is compute the gradient of the objective function $V(\theta)$, and we can optimize it! The method of **finite difference** from calculus provides an approximation of the gradient:

$$\frac{\partial V(\theta)}{\partial \theta_k} \approx \frac{V(\theta + \epsilon u_k) - V(\theta)}{\epsilon}$$

where u_k is a unit vector with 1 in kth component, 0 elsewhere. This method uses n evaluations to compute the policy gradient in n dimensions, so it is quite inefficient, and it usually only provides a noisy approximation of the true policy gradient. However, it has the advantage that it works for non-differentiable policies. An example of a successful use of this method to train the AIBO robot gait can be found in [2].

4.1.1 Analytic gradients

Let us set the objective function $V(\theta)$ to be the expected rewards for an episode,

$$V(\theta) = \mathbb{E}_{(s_t, a_t) \sim \pi_\theta} \left[\sum_{t=0}^{T} R(s_t, a_t) \right] = \mathbb{E}_{\tau \sim \pi_\theta} \left[R(\tau) \right] = \sum_\tau P(\tau; \theta) R(\tau)$$

where τ is a trajectory,

$$\tau = (s_0, a_0, r_0, \cdots, s_{T-1}, a_{T-1}, r_{T-1}, s_T)$$

$P(\tau; \theta)$ denotes the probability over trajectories when following policy π_θ, and $R(\tau)$ is the sum of rewards for a trajectory. Note that this objective function is the same as the **start value** $J_1(\theta)$ as mentioned in section 3.1 when the discount $\gamma = 1$.

If we can mathematically compute the policy gradient $\nabla_\theta \pi_\theta(a|s)$, then we can go right ahead and

compute the gradient of this objective function with respect to θ:

$$\nabla_\theta V(\theta) = \nabla_\theta \sum_\tau P(\tau;\theta) R(\tau) \tag{1}$$

$$= \sum_\tau \nabla_\theta P(\tau;\theta) R(\tau) \tag{2}$$

$$= \sum_\tau \frac{P(\tau;\theta)}{P(\tau;\theta)} \nabla_\theta P(\tau;\theta) R(\tau) \tag{3}$$

$$= \sum_\tau P(\tau;\theta) R(\tau) \frac{\nabla_\theta P(\tau;\theta)}{P(\tau;\theta)} \tag{4}$$

$$= \sum_\tau P(\tau;\theta) R(\tau) \nabla_\theta \log P(\tau;\theta) \tag{5}$$

$$= \mathbb{E}_{\tau\sim\pi_\theta} \left[R(\tau) \nabla_\theta \log P(\tau;\theta) \right] \tag{6}$$

The expression $\frac{\nabla_\theta P(\tau;\theta)}{P(\tau;\theta)}$ in equation (4) is known as the **likelihood ratio**.

The trick in steps (3)-(6) helps for two reasons. First, it helps us get the gradient into the form $\mathbb{E}_{\tau\sim\pi_\theta}[\dots]$, which allows us to approximate the gradient by sampling trajectories $\tau^{(i)}$:

$$\nabla_\theta V(\theta) \approx \hat{g} = \frac{1}{m} \sum_{i=1}^{m} R(\tau^{(i)}) \nabla_\theta \log P(\tau^{(i)};\theta) \tag{7}$$

Second, computing $\nabla_\theta \log P(\tau^{(i)};\theta)$ is easier than working with $P(\tau^{(i)};\theta)$ directly:

$$\nabla_\theta \log P(\tau^{(i)};\theta) = \nabla_\theta \log \left[\underbrace{\mu(s_0)}_{\text{Initial state distribution}} \prod_{t=0}^{T-1} \underbrace{\pi_\theta(a_t|s_t)}_{\text{policy}} \underbrace{P(s_{t+1}|s_t,a_t)}_{\text{dynamics model}} \right] \tag{8}$$

$$= \nabla_\theta \left[\log \mu(s_0) + \sum_{t=0}^{T-1} \log \pi_\theta(a_t|s_t) + \log P(s_{t+1}|s_t,a_t) \right] \tag{9}$$

$$= \sum_{t=0}^{T-1} \underbrace{\nabla_\theta \log \pi_\theta(a_t|s_t)}_{\text{no dynamics model required!}} \tag{10}$$

Working with $\log P(\tau^{(i)};\theta)$ instead of $P(\tau^{(i)};\theta)$ allows us to represent the gradient without reference to the initial state distribution, or even the environment dynamics model!

The expression $\nabla_\theta \log \pi_\theta(a_t|s_t)$ is known as the **score function**.

Putting equations (7) and (10) together, we get

$$\nabla_\theta V(\theta) \approx \hat{g} = \frac{1}{m} \sum_{i=1}^{m} R(\tau^{(i)}) \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta \left(a_t^{(i)} \mid s_t^{(i)} \right)$$

which we can convert into a concrete algorithm for optimizing π_θ (section 5). But before that, we will mention the generalized version of this result and cover an optimization of the above derivation that takes advantage of decomposing $R(\tau^{(i)})$ into a sum of reward terms $r_t^{(i)}$ (section 4.3).

4.2 The policy gradient theorem

Theorem 4.1. *For any differentiable policy $\pi_\theta(a|s)$ and for any of the policy objective functions $V(\theta) = J_1, J_{avR}$, or $\frac{1}{1-\gamma} J_{avV}$, the policy gradient is*

$$\nabla_\theta V(\theta) = \mathbb{E}_{\pi_\theta}[Q^{\pi_\theta}(s,a) \cdot \nabla_\theta \log \pi_\theta(a|s)]$$

We will not go over the derivation of this more general theorem, but the same concepts discussed in this lecture apply to non-episodic (continuing) environments. In our discussion thus far, the total episode rewards $R(\tau)$ have been substituted in place of the Q values of this theorem, but in the following section we will use the temporal structure to get our result into a form that looks more like this theorem, where the future returns G_t (which are unbiased estimates of $Q(s_t, a_t)$) appear in place of $Q^{\pi_\theta}(s,a)$.

4.3 Using temporal structure of rewards for the policy gradient

Equation (6) above can be written

$$\nabla_\theta V(\theta) = \nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)] = \mathbb{E}_{\tau \sim \pi_\theta}\left[R(\tau) \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\right] \tag{11}$$

Notice that the rewards $R(\tau^{(i)})$ are treated as a single number which is a function of an entire trajectory $\tau^{(i)}$. We can break this down into the sum of all the rewards encountered in the trajectory,

$$R(\tau) = \sum_{t=1}^{T-1} R(s_t, a_t)$$

Using this knowledge, we can derive the gradient estimate for a single reward term $r_{t'}$ in exactly the same way we derived equation (11):

$$\nabla_\theta \mathbb{E}_{\pi_\theta}[r_{t'}] = \mathbb{E}_{\pi_\theta}\left[r_{t'} \sum_{t=0}^{t'} \nabla_\theta \log \pi_\theta(a_t|s_t)\right]$$

Since $\sum_{t'=t}^{T-1} r_{t'}^{(i)}$ is the return $G_t^{(i)}$, we can sum this up over all time steps for a trajectory to get

$$\nabla_\theta V(\theta) = \nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)] = \mathbb{E}_{\pi_\theta}\left[\sum_{t'=0}^{T-1} r_{t'} \sum_{t=0}^{t'} \nabla_\theta \log \pi_\theta(a_t|s_t)\right] \tag{12}$$

$$= \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T-1} r_{t'}\right] \tag{13}$$

$$= \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T-1} G_t \cdot \nabla_\theta \log \pi_\theta(a_t|s_t)\right] \tag{14}$$

Going from (12) to (13) may not be obvious, so let's go over a quick example. Say we have a trajectory that is three time steps long. Then equation (12) becomes

$$\begin{aligned}\nabla_\theta V(\theta) = \mathbb{E}_{\pi_\theta}\big[&r_0 \nabla_\theta \log \pi_\theta(a_0|s_0) + \\ &r_1(\nabla_\theta \log \pi_\theta(a_0|s_0) + \nabla_\theta \log \pi_\theta(a_1|s_1)) + \\ &r_2(\nabla_\theta \log \pi_\theta(a_0|s_0) + \nabla_\theta \log \pi_\theta(a_1|s_1) + \nabla_\theta \log \pi_\theta(a_2|s_2))\big]\end{aligned}$$

Regrouping the terms, we get

$$\begin{aligned}\nabla_\theta V(\theta) = \mathbb{E}_{\pi_\theta}\big[&\nabla_\theta \log \pi_\theta(a_0|s_0)(r_0 + r_1 + r_2) + \\ &\nabla_\theta \log \pi_\theta(a_1|s_1)(r_1 + r_2) + \\ &\nabla_\theta \log \pi_\theta(a_2|s_2)(r_2)\big]\end{aligned}$$

which equals equation (13) as expected. The main idea is that the policy's choice at a particular time step t only affects rewards received in later steps of the episode, and has no effect on rewards received in previous time steps. Our original expression in equation (11) did not take this into account.

Our final expression that we will use in the policy gradient algorithm in the next section is

$$\nabla_\theta V(\theta) = \nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)] \approx \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{T-1} G_t^{(i)} \cdot \nabla_\theta \log \pi_\theta(a_t^{(i)}|s_t^{(i)}) \tag{15}$$

5 REINFORCE: A Monte-Carlo policy gradient algorithm

We've done most of the work towards our first policy gradient algorithm in the sections above. The algorithm simply samples multiple trajectories following the policy π_θ while updating θ using the estimated gradient (15).

Algorithm 1 REINFORCE: Monte-Carlo policy gradient algorithm

```
procedure REINFORCE(α)
    Initialize policy parameters θ arbitrarily
    for each episode {s_1, a_1, r_2, ··· , s_{T−1}, a_{T−1}, r_T} ∼ π_θ do
        for t = 1 to T − 1 do
            θ ← θ + α · G_t ∇_θ log π_θ(a_t|s_t)
    return θ
```

6 Differentiable policy classes

6.1 Discrete action space: softmax policy

In discrete action spaces, the softmax function is commonly used to parameterize the policy:

$$\pi_\theta(a|s) = \frac{e^{\phi(s,a)^T \theta}}{\sum_{a'} e^{\phi(s,a')^T \theta}}$$

The score function is then

$$\begin{aligned}
\nabla_\theta \log \pi_\theta(a|s) &= \nabla_\theta \left[\phi(s,a)^T\theta - \log \sum_{a'} e^{\phi(s,a')^T\theta} \right] \\
&= \phi(s,a) - \frac{1}{\sum_{a'} e^{\phi(s,a')^T\theta}} \nabla_\theta \sum_{a'} e^{\phi(s,a')^T\theta} \\
&= \phi(s,a) - \frac{1}{\sum_{a'} e^{\phi(s,a')^T\theta}} \sum_{a'} \phi(s,a') e^{\phi(s,a')^T\theta} \\
&= \phi(s,a) - \sum_{a'} \phi(s,a') \frac{e^{\phi(s,a')^T\theta}}{\sum_{a''} e^{\phi(s,a'')^T\theta}} \\
&= \phi(s,a) - \sum_{a'} \phi(s,a') \pi_\theta(a'|s) \\
&= \phi(s,a) - \mathbb{E}_{a' \sim \pi_\theta(a'|s)}[\phi(s,a')]
\end{aligned}$$

6.2 Continuous action space: Gaussian policy

For continuous action spaces, a common choice is a Gaussian policy $a \sim \mathcal{N}(\mu(s), \sigma^2)$.

- The mean action is a linear combination of state features: $\mu(s) = \phi(s)^T\theta$
- The variance σ^2 can be fixed, or also parameterized

The score function is

$$\nabla_\theta \log \pi_\theta(a|s) = \frac{(a - \mu(s))\phi(s)}{\sigma^2}$$

7 Variance reduction with a baseline

A weakness of Monte-Carlo policy gradient algorithms is that the returns $G_t^{(i)}$ often have high variance across multiple episodes. One way to address this is to subtract a **baseline** $b(s)$ from each $G_t^{(i)}$. The baseline can be any function, as long as it does not vary with a.

$$\nabla_\theta V(\theta) = \nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)] = \mathbb{E}_{\pi_\theta} \left[\sum_{t=0}^{T-1} (G_t - b(s_t)) \nabla_\theta \log \pi_\theta(a_t|s_t) \right]$$

First, why do we want to do this? Intuitively, we can think of $(G_t - b(s_t))$ as an estimate of how much better we did after time step t than is expected by the baseline $b(s_t)$. So, if the baseline is approximately equal to the expected return $b(s_t) \approx \mathbb{E}[r_t + r_{t+1} + \cdots + r_{T-1}]$, then we will be increasing the log-probability of action a_t proportionally to how much better the return G_t is than expected. Previously, we were increasing the log-probability proportionally to the magnitude of G_t, so even if the policy always achieved exactly its expected returns, we would still be applying gradient updates that could cause it to diverge. The quantity $(G_t - b(s_t))$ is usually called the *advantage*, A_t. We can estimate the true advantage from a sampled trajectory $\tau^{(i)}$ with

$$\hat{A}_t = (G_t^{(i)} - b(s_t))$$

Secondly, why *can* we do this? It turns out that subtracting a baseline in this manner does not introduce any bias into the gradient calculation. $\mathbb{E}_\tau[b(s_t)\nabla_\theta \log \pi_\theta(a_t|s_t)]$ evaluates to zero, and hence

has no effect on the gradient update.

$$
\begin{aligned}
&\mathbb{E}_{\tau\sim\pi_\theta}[b(s_t)\nabla_\theta \log \pi_\theta(a_t|s_t)] \\
&= \mathbb{E}_{s_{0:t},a_{0:(t-1)}}\left[\mathbb{E}_{s_{(t+1):T},a_{t:(T-1)}}[\nabla_\theta \log \pi_\theta(a_t|s_t)b(s_t)]\right] \text{ (break up expectation)} \\
&= \mathbb{E}_{s_{0:t},a_{0:(t-1)}}\left[b(s_t)\mathbb{E}_{s_{(t+1):T},a_{t:(T-1)}}[\nabla_\theta \log \pi_\theta(a_t|s_t)]\right] \text{ (pull baseline term out)} \\
&= \mathbb{E}_{s_{0:t},a_{0:(t-1)}}\left[b(s_t)\mathbb{E}_{a_t}[\nabla_\theta \log \pi_\theta(a_t|s_t)]\right] \text{ (remove irrelevant variables)} \\
&= \mathbb{E}_{s_0:t,a_0:(t-1)}\left[b(s_t)\sum_a \pi_\theta(a_t|s_t)\frac{\nabla_\theta \pi_\theta(a_t|s_t)}{\pi_\theta(a_t|s_t)}\right] \text{ (expand expectation + take derivative of logarithm)} \\
&= \mathbb{E}_{s_0:t,a_0:(t-1)}\left[b(s_t)\sum_{a_t} \nabla_\theta \pi_\theta(a_t|s_t)\right] \\
&= \mathbb{E}_{s_0:t,a_0:(t-1)}\left[b(s_t)\nabla_\theta \sum_{a_t} \pi_\theta(a_t|s_t)\right] \\
&= \mathbb{E}_{s_0:t,a_0:(t-1)}\left[b(s_t)\nabla_\theta 1\right] \\
&= \mathbb{E}_{s_0:t,a_0:(t-1)}\left[b(s_t)\cdot 0\right] \\
&= 0
\end{aligned}
$$

7.1 Vanilla policy gradient

Using the baseline as described above, we introduce the "vanilla" policy gradient algorithm. Suppose that the baseline function has parameters $\mathbf{w}$.

Algorithm 2 Vanilla Policy Gradient Algorithm

procedure POLICY GRADIENT(α)
  Initialize policy parameters θ and baseline values $b(s)$ for all s, e.g. to 0
  for iteration $= 1, 2, \ldots$ **do**
    Collect a set of m trajectories by executing the current policy π_θ
    for each time step t of each trajectory $\tau^{(i)}$ **do**
      Compute the *return* $G_t^{(i)} = \sum_{t'=t}^{T-1} r_{t'}$
      Compute the *advantage estimate* $\hat{A}_t^{(i)} = G_t^{(i)} - b(s_t)$
    Re-fit the baseline to the empirical returns by updating $\mathbf{w}$ to minimize

$$\sum_{i=1}^{m}\sum_{t=0}^{T-1} \|b(s_t) - G_t^{(i)}\|^2$$

    Update policy parameters θ using the policy gradient estimate $\hat{g}$

$$\hat{g} = \sum_{i=1}^{m}\sum_{t=0}^{T-1} \hat{A}_t^{(i)} \nabla_\theta \log \pi_\theta(a_t^{(i)}|s_t^{(i)})$$

    with an optimizer like SGD ($\theta \leftarrow \theta + \alpha \cdot \hat{g}$) or Adam
  return θ and baseline values $b(s)$

One natural choice for the baseline is the state value function, $b(s_t) = V(s_t)$. Under this formulation, we can define the advantage function as $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$. However, since we do not know the true state values, we instead use an estimate $\hat{V}(s_t;\mathbf{w})$ for some weight vector $\mathbf{w}$. We can simultaneously learn the weight vector $\mathbf{w}$ for the baseline (state-value) function and policy parameters θ using the Monte-Carlo trajectory samples.

Note that in the above algorithm, we usually do not compute the gradients $\sum_t \hat{A}_t \nabla_\theta \log \pi_\theta(a_t|s_t)$ individually. Rather, we accumulate data from a batch into a loss function

$$L(\theta) = \sum_t \hat{A}_t \log \pi_\theta(a_t|s_t)$$

and then apply the gradients all at once by computing $\nabla_\theta L(\theta)$. We can also introduce a component into this loss to fit the baseline function:

$$L(\theta, \mathbf{w}) = \sum_t \left(\hat{A}_t \log \pi_\theta(a_t|s_t) - \|b(s_t) - G_t^{(i)}\|^2 \right)$$

We can then compute the gradients of $L(\theta, \mathbf{w})$ w.r.t. θ and $\mathbf{w}$ to perform SGD updates.

7.2 N-step estimators

In the above derivations, we have used the Monte-Carlo estimates of the reward in the policy gradient approximation. However, if we have access to a value function (for example, the baseline), then we can also us TD methods for the policy gradient update, or any intermediate blend between TD and MC methods:

$$\begin{aligned}
\hat{G}_t^{(1)} &= r_t + \gamma V(s_{t+1}) \\
\hat{G}_t^{(2)} &= r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) \\
&\cdots \\
\hat{G}_t^{(\text{inf})} &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots
\end{aligned}$$

which we can also use to compute advantages:

$$\begin{aligned}
\hat{A}_t^{(1)} &= r_t + \gamma V(s_{t+1}) - V(s_t) \\
\hat{A}_t^{(2)} &= r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) - V(s_t) \\
&\cdots \\
\hat{A}_t^{(\text{inf})} &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots - V(s_t)
\end{aligned}$$

$A_t^{(1)}$ is a purely TD estimate, and has low variance, but high bias. $A_t^{(\text{inf})}$ is a purely MC estimate, and has zero bias, but high variance. If we choose an intermediate value of k for $A_t^{(k)}$, we can get an intermediate amount of bias and an intermediate amount of variance.

References

[1] https://blog.openai.com/evolution-strategies/

[2] Kohl and Stone. Policy gradient reinforcement learning for fast quadrupedal locomotion. ICRA 2004. http://www.cs.utexas.edu/ ai-lab/pubs/icra04.pdf Emma Brunskill (CS234 Reinforcement Learning.) Lecture 8: Policy Gradient I 28 Winter 201

CS234 Notes - Lecture 9
Advanced Policy Gradient

Patrick Cho, Emma Brunskill

February 11, 2019

1 Policy Gradient Objective

Recall that in Policy Gradient, we parameterize the policy π_θ and directly optimize for it using experience in the environment. We first define the probability of a trajectory given our current policy π_θ, which we denote as $\pi_\theta(\tau)$.

$$\pi_\theta(\tau) = \pi_\theta(s_1, a_1, ..., s_T, a_T) = P(s_1)\prod_{t=1}^{T}\pi_\theta(a_t|s_t)P(s_{t+1}|s_t, a_t)$$

Parsing the function above, $P(s_1)$ is the probability of starting at state s_1, $\pi_\theta(a_t|s_t)$ is the probability of our current policy selecting action a_t given that we are in state s_t, and $P(s_{t+1}|s_t, a_t)$ is the probability of the environment's dynamics transiting us to state s_{t+1} given that we start at s_t and take action a_t. Note that the we overload the notation for π_θ here to either mean the probability of a trajectory ($\pi_\theta(\tau)$) or the probability of an action given a state ($\pi_\theta(a|s)$).

The goal of Policy Gradient, similar to most other RL objectives that we have discussed thus far, is to maximize the discounted sum of rewards.

$$\theta^* = \arg\max_\theta \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\sum_t \gamma^t r(s_t, a_t)\right]$$

We denote our objective function as $J(\theta)$ which can be estimated using Monte Carlo. We also use $r(\tau)$ to represent the discounted sum of rewards over trajectory τ.

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\sum_t \gamma^t r(s_t, a_t)\right] = \int \pi_\theta(\tau)r(\tau)d\tau \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\gamma^t r(s_{i,t}, a_{i,t})$$

$$\theta^* = \arg\max_\theta J(\theta)$$

We define $P_\theta(s, a)$ to be the probability of seeing (s,a) pair in our trajectory. Note that in the case of infinite horizon where a stationary distribution of states exist, we can write $P_\theta(s, a) = d^{\pi_\theta}(s)\pi_\theta(a|s)$ where $d^{\pi_\theta}(s)$ is the stationary state distribution under policy π_θ.

In the infinite horizon case, we have

$$\begin{aligned}\theta^* &= \arg\max_\theta \sum_{t=1}^{\infty}\mathbb{E}_{(s,a)\sim P_\theta(s,a)}[\gamma^t r(s, a)] \\ &= \arg\max_\theta \frac{1}{1-\gamma}\mathbb{E}_{(s,a)\sim P_\theta(s,a)}[r(s, a)] \\ &= \arg\max_\theta \mathbb{E}_{(s,a)\sim P_\theta(s,a)}[r(s, a)]\end{aligned}$$

In the finite horizon case, we have

$$\theta^* = \arg\max_\theta \sum_{t=1}^{T} \mathbb{E}_{(s_t,a_t)\sim P_\theta(s_t,a_t)}[\gamma^t r(s_t, a_t)]$$

We can use gradient based methods to do the above optimization. In particular, we need to find the gradient of $J(\theta)$ with respect to θ.

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \int \pi_\theta(\tau) r(\tau) d\tau \\
&= \int \nabla_\theta \pi_\theta(\tau) r(\tau) d\tau \\
&= \int \pi_\theta(\tau) \frac{\nabla_\theta \pi_\theta(\tau)}{\pi_\theta(\tau)} r(\tau) d\tau \\
&= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau) r(\tau)]
\end{aligned}$$

As seen above, we have moved the gradient from outside of the expectation to inside of the expectation. This is commonly known as the log derivative trick. The advantage of doing so is that now we do not need to take gradient over the dynamics function as seen below.

$$\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau) r(\tau)] \\
&= \mathbb{E}_{\tau\sim\pi_\theta(\tau)} \left[\nabla_\theta \left[\log P(s_1) + \sum_{t=1}^{T} (\log \pi_\theta(a_t|s_t) + \log P(s_{t+1}|s_t, a_t)) \right] r(\tau) \right] \\
&= \mathbb{E}_{\tau\sim\pi_\theta(\tau)} \left[\nabla_\theta \left[\sum_{t=1}^{T} (\log \pi_\theta(a_t|s_t)) \right] r(\tau) \right] \\
&= \mathbb{E}_{\tau\sim\pi_\theta(\tau)} \left[\sum_{t=1}^{T} \left(\nabla_\theta \left(\log \pi_\theta(a_t|s_t) \right) \left(\sum_{t=1}^{T} \gamma^t r(s_t, a_t) \right) \right) \right] \\
&\approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \left(\nabla_\theta \left(\log \pi_\theta(a_{i,t}|s_{i,t}) \right) \left(\sum_{t=1}^{T} \gamma^t r(s_{i,t}, a_{i,t}) \right) \right)
\end{aligned}$$

In the third equality, the terms cancel out because they do not involve θ. In the last step, we use Monte Carlo estimates from rollout trajectories.

Note that there are many similarities between the above formulation and the Maximum Likelihood Estimate (MLE) in the supervised learning setting. For MLE in supervised learning, we have likelihood, $J'(\theta)$, and log-likelihood, $J(\theta)$:

$$J'(\theta) = \prod_{i=1}^{N} P(y_i|x_i)$$

$$J(\theta) = \log J'(\theta) = \sum_{i=1}^{N} \log P(y_i|x_i)$$

$$\nabla_\theta J(\theta) = \sum_{i=1}^{N} \nabla_\theta \log P(y_i|x_i)$$

Comparing with the Policy Gradient derivation, the key difference is the sum of rewards. We can even view MLE as policy gradient with a return of 1 for all examples. Although this difference may seem minor, it can cause the problem to become much harder. In particular, the summation of rewards drastically increases variance. Hence, in the next section, we discuss two methods to reduce variance.

2 Reducing Variance in Policy Gradient

2.1 Causality

We first note that the action taken at time t' cannot affect reward at time t for all $t < t'$. This is known as causality since what we do now should not affect the past. Hence, we can change the summation of rewards, $\sum_{t=1}^{T} \gamma^t r(s_{i,t}, a_{i,t})$, to the reward-to-go, $\hat{Q}_{i,t} = \sum_{t'=t}^{T} \gamma^{t'} r(s_{i,t'}, a_{i,t'})$. We use $\hat{Q}$ here to denote that this is a Monte Carlo estimate of Q. Doing so helps to reduce variance since we effectively reduce noise from prior rewards. In particular, our objective changes to:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \left(\nabla_\theta \log \pi_\theta(a_{i,t}, s_{i,t}) \left(\sum_{t'=t}^{T} \gamma^{t'} r(s_{i,t'}, a_{i,t'}) \right) \right) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \left(\nabla_\theta \log \pi_\theta(a_{i,t}, s_{i,t}) \hat{Q}_{i,t} \right)$$

2.2 Baselines

Now, we consider subtracting a baseline from the reward-to-go. That is, we change our objective into the following form:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}, s_{i,t}) \left[\left(\sum_{t'=t}^{T} \gamma^{t'} r(s_{i,t'}, a_{i,t'}) \right) - b \right]$$

We first note that subtracting a constant baseline, b, is unbiased. That is under expectation of trajectories from our current policy π_θ, the term we have just included is 0.

$$\begin{aligned}
\mathbb{E}_{\tau \sim \pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau) b] &= \int \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) b d\tau \\
&= \int \pi_\theta(\tau) \frac{\nabla_\theta \pi_\theta(\tau)}{\pi_\theta(\tau)} b d\tau \\
&= \int \nabla_\theta \pi_\theta(\tau) b d\tau \\
&= b \nabla_\theta \int \pi_\theta(\tau) d\tau \\
&= b \nabla_\theta 1 = 0
\end{aligned}$$

In the last equality, the integral of the probability of a trajectory over all trajectories is 1. In the second last equality, we are able to take b out of the integral since b is a constant (e.g. average return, $b = \frac{1}{N} \sum_{i=1}^{N} r(\tau)$). However, we can also show that this term is unbiased if b is a function of state s.

$$\begin{aligned}
\mathbb{E}_{\tau \sim \pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(a_t|s_t) b(s_t)] &= \mathbb{E}_{s_{0:t}, a_{0:(t-1)}} \left[\mathbb{E}_{s_{(t+1):T}, a_{t:(T-1)}} [\nabla_\theta \log \pi_\theta(a_t|s_t) b(s_t)] \right] \\
&= \mathbb{E}_{s_{0:t}, a_{0:(t-1)}} \left[b(s_t) \mathbb{E}_{s_{(t+1):T}, a_{t:(T-1)}} [\nabla_\theta \log \pi_\theta(a_t|s_t)] \right] \\
&= \mathbb{E}_{s_{0:t}, a_{0:(t-1)}} \left[b(s_t) \mathbb{E}_{a_t} [\nabla_\theta \log \pi_\theta(a_t|s_t)] \right] \\
&= \mathbb{E}_{s_{0:t}, a_{0:(t-1)}} \left[b(s_t) \cdot 0 \right] = 0
\end{aligned}$$

As seen above, if no assumptions on the policy are made, the baseline cannot be a function of actions since the proof depends on being able factor out $b(s_t)$. Exceptions exist if we make some assumptions. See [3] for an example of action-dependent baselines.

One common baseline that is used is the value function, $V^{\pi_\theta}(s)$. Since the reward-to-go estimates the state-action value function $Q^{\pi_\theta}(s,a)$, by subtracting this baseline from Q, we are essentially calculating the advantage, $A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s)$. In terms of implementation, this means training a separate value function $V_\phi(s)$.

As a side note, instead of using actual returns from the environment to estimate $Q^{\pi_\theta}(s,a)$, we can train another state-action value function $Q_w(s,a)$ to approximate the policy gradient. This approach is known as actor-critic where the Q_w function is the critic. Essentially, the critic does policy evaluation and the actor does policy improvement.

One can ask: what is the optimal baseline to subtract in order to minimize variance? The optimal baseline is in fact the expected reward weighted by the square of the gradients as shown below.

$$Var[x] = \mathbb{E}[X^2] - \mathbb{E}[X]^2$$
$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau)(r(\tau) - b)]$$

$$\begin{aligned} Var &= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[(\nabla_\theta \log \pi_\theta(\tau)(r(\tau) - b))^2\right] - \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\nabla_\theta \log \pi_\theta(\tau)(r(\tau) - b)\right]^2 \\ &= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[(\nabla_\theta \log \pi_\theta(\tau)(r(\tau) - b))^2\right] - \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\nabla_\theta \log \pi_\theta(\tau)(r(\tau))\right]^2 \end{aligned}$$

In the equation above, we are able to remove b in the second term since we have proven that b is unbiased in expectation. To minimize variance, we set its gradient with respect to b to 0. The second term in the variance equation does not depend on b and therefore disappears.

$$\begin{aligned} \frac{dVar}{db} &= \frac{d}{db}\mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[(\nabla_\theta \log \pi_\theta(\tau)(r(\tau) - b))^2\right] \\ &= \frac{d}{db}(-2\mathbb{E}_{\tau\sim\pi_\theta(\tau)}[(\nabla_\theta \log \pi_\theta(\tau))^2 r(\tau) b] + \mathbb{E}_{\tau\sim\pi_\theta(\tau)}[(\nabla_\theta \log \pi_\theta(\tau))^2 b^2]) \\ &= -2\mathbb{E}[(\nabla_\theta \log \pi_\theta(\tau))^2 r(\tau)] + 2\mathbb{E}[(\nabla_\theta \log \pi_\theta(\tau))^2 b] = 0 \end{aligned}$$

$$b = \frac{\mathbb{E}[(\nabla_\theta \log \pi_\theta(\tau))^2 r(\tau)]}{\mathbb{E}[(\nabla_\theta \log \pi_\theta(\tau))^2]}$$

3 Off Policy Policy Gradient

In the analysis above, our objective involves taking an expectation over trajectories drawn from $\pi_\theta(\tau)$. This means that Policy Gradient with the above objective will result in an on policy algorithm. Whenever we change our parameters θ, our policy changes and all our old trajectories cannot be reused. Compare this with DQN which is able to store prior experience to be reused since it is an off policy algorithm. Hence, Policy Gradient is in general less sample efficient than Q learning. To resolve this, we discuss the use of Importance Sampling to generate an Off Policy Policy Gradient algorithm. In particular, we consider the changes that need to be made if we estimate $J(\theta)$ using trajectories drawn from a prior policy π_θ instead of current policy $\pi_{\theta'}$.

$$
\begin{aligned}
\theta^* &= \arg\max_{\theta'} J(\theta') \\
&= \arg\max_{\theta'} \mathbb{E}_{\tau\sim\pi_{\theta'}(\tau)}[r(\tau)] \\
&= \arg\max_{\theta'} \mathbb{E}_{\tau\sim\pi_{\theta}(\tau)}\left[\frac{\pi_{\theta'}(\tau)}{\pi_{\theta}(\tau)} r(\tau)\right] \\
&= \arg\max_{\theta'} \mathbb{E}_{\tau\sim\pi_{\theta}(\tau)}\left[\frac{P(s_1)\prod_{t=1}^{T}\pi_{\theta'}(a_t|s_t)P(s_{t+1}|s_t,a_t)}{P(s_1)\prod_{t=1}^{T}\pi_{\theta}(a_t|s_t)P(s_{t+1}|s_t,a_t)} r(\tau)\right] \\
&= \arg\max_{\theta'} \mathbb{E}_{\tau\sim\pi_{\theta}(\tau)}\left[\frac{\prod_{t=1}^{T}\pi_{\theta'}(a_t|s_t)}{\prod_{t=1}^{T}\pi_{\theta}(a_t|s_t)} r(\tau)\right]
\end{aligned}
$$

Hence, we have for old parameters θ:

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_{\theta}(\tau)}[r(\tau)]$$

and for new parameters θ':

$$J(\theta') = \mathbb{E}_{\tau\sim\pi_{\theta}(\tau)}[\frac{\pi_{\theta'}(\tau)}{\pi_{\theta}(\tau)} r(\tau)]$$

$$
\begin{aligned}
\nabla_{\theta'} J(\theta') &= \mathbb{E}_{\tau\sim\pi_{\theta}(\tau)}\left[\frac{\nabla_{\theta'}\pi_{\theta'}(\tau)}{\pi_{\theta}(\tau)} r(\tau)\right] \\
&= \mathbb{E}_{\tau\sim\pi_{\theta}(\tau)}\left[\frac{\pi_{\theta'}(\tau)}{\pi_{\theta}(\tau)} \nabla_{\theta'}\log\pi_{\theta'}(\tau) r(\tau)\right] \\
&= \mathbb{E}_{\tau\sim\pi_{\theta}(\tau)}\left[\left(\prod_{t=1}^{T}\frac{\pi_{\theta'}(a_t|s_t)}{\pi_{\theta}(a_t|s_t)}\right)\left(\sum_{t=1}^{T}\nabla_{\theta'}\left(\log\pi_{\theta'}(a_t|s_t)\right)\right)\left(\sum_{t=1}^{T}\gamma^t r(s_t,a_t)\right)\right] \\
&= \mathbb{E}_{\tau\sim\pi_{\theta}(\tau)}\left[\sum_{t=1}^{T}\left(\nabla_{\theta'}\left(\log\pi_{\theta'}(a_t|s_t)\right)\left(\prod_{t'=1}^{t}\frac{\pi_{\theta'}(a_{t'}|s_{t'})}{\pi_{\theta}(a_{t'}|s_{t'})}\right)\left(\sum_{t'=t}^{T}\gamma^{t'} r(s_{t'},a_{t'})\right)\right)\right]
\end{aligned}
$$

In the last equality, we invoke causality. In particular, the probability of the first k transitions only depends on the first k actions and not future actions.

4 Relative Policy Performance Identity

One issue with directly taking gradient steps according to the gradient of the objective $J(\theta)$ with respect to the parameters θ is that moving in the parameter space is not the same as moving in the policy space. This causes a problem in the choice of step sizes. Small step sizes causes learning to be slow but large step sizes may cause the policy to become bad.

In the case of supervised learning, this is usually fine since the following updates will usually fix this problem. However, in the context of reinforcement learning, a bad policy will cause the next batch of data to be collected under the bad policy. Hence, stepping to a bad policy may cause a collapse in performance from which the algorithm cannot recover. Simple line search in the direction of the gradient could be performed to mitigate this issue. For example, we could try multiple learning rates

for each update and choose the learning rate that gives best performance. However, doing so is naive and will cause slow convergence in cases where the first order approximation (gradient) is bad.

Trust Region Policy Optimization, discussed in the next section, is an algorithm that tries to resolve this issue. Building towards this, we first derive an identity with regards to relative policy performance, that is $J(\pi') - J(\pi)$. Here we use the following notations: $J(\pi') = J(\theta')$, $J(\pi) = J(\theta)$, $\pi' = \pi_{\theta'}$ and $\pi = \pi_\theta$.

Lemma 4.1.

$$J(\pi') - J(\pi) = \mathbb{E}_{\tau\sim\pi'}\left[\sum_{t=0}^{\infty}\gamma^t A^\pi(s_t, a_t)\right]$$

Proof.

$$\begin{aligned}
\mathbb{E}_{\tau\sim\pi'}\left[\sum_{t=0}^{\infty}\gamma^t A^\pi(s_t, a_t)\right] &= \mathbb{E}_{\tau\sim\pi'}\left[\sum_{t=0}^{\infty}\gamma^t [r(s_t, a_t) + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)]\right] \\
&= \mathbb{E}_{\tau\sim\pi'}\left[\sum_{t=0}^{\infty}\gamma^t [r(s_t, a_t)]\right] + \mathbb{E}_{\tau\sim\pi'}\left[\sum_{t=0}^{\infty}\gamma^t [\gamma V^\pi(s_{t+1}) - V^\pi(s_t)]\right] \\
&= J(\pi') - \mathbb{E}_{\tau\sim\pi'}[V^\pi(s_0)] \\
&= J(\pi') - J(\pi)
\end{aligned}$$

□

Hence, we have

$$\begin{aligned}
\max_{\pi'} J(\pi') &= \max_{\pi'} J(\pi') - J(\pi) \\
&= \max_{\pi'} \mathbb{E}_{\tau\sim\pi'}\left[\sum_{t=0}^{\infty}\gamma^t A^\pi(s_t, a_t)\right]
\end{aligned}$$

The issue with the above expression is that we require trajectories from π'. This makes optimization impossible since we have not yet found π' but need to draw samples from π'. Once again, we use likelihood ratios to circumvent this issue.

$$\begin{aligned}
J(\pi') - J(\pi) &= \mathbb{E}_{\tau\sim\pi'}\left[\sum_{t=0}^{\infty}\gamma^t A^\pi(s_t, a_t)\right] \\
&= \frac{1}{1-\gamma}\mathbb{E}_{\substack{s\sim d^{\pi'} \\ a\sim\pi'}}[A^\pi(s, a)] \\
&= \frac{1}{1-\gamma}\mathbb{E}_{\substack{s\sim d^{\pi'} \\ a\sim\pi}}[\frac{\pi'(a|s)}{\pi(a|s)}A^\pi(s, a)] \\
&\approx \frac{1}{1-\gamma}\mathbb{E}_{\substack{s\sim d^{\pi} \\ a\sim\pi}}[\frac{\pi'(a|s)}{\pi(a|s)}A^\pi(s, a)] \\
&= \frac{1}{1-\gamma}L_\pi(\pi')
\end{aligned}$$

We call $L_\pi(\pi')$ the surrogate objective. One key question is when we can make the above approximation. Clearly, when $\pi = \pi'$, the approximation holds with equality. However, this would not be useful since we want to improve our current policy π to a better policy π'. In the derivations of Trust Region Policy Optimization (TRPO) below, we give bounds for the approximation.

5 Trust Region Policy Optimization

The key idea in TRPO [5] is to define a trust region that constrains updates to the policy. This constraint is in the policy space rather than in the parameter space and becomes the new "step size" of the algorithm. In this way, we can approximately ensure that the new policy after the policy update performs better than the old policy.

5.1 Problem Setup

Consider a finite state and action MDP $\mathcal{M} = (S, A, M, R, \gamma)$ where M is the transition function. In this section, we assume that $|S|$ and $|A|$ are both finite, and that $0 < \gamma < 1$. Although the derivation is for finite states and actions, the algorithm works for continuous states and actions as well. We define

$$d^\pi(s) = (1-\gamma)\sum_{t=0}^{\infty}\gamma^t M(s_t = s|\pi) \tag{1}$$

to be the discounted state visitation distribution following policy π with dynamics M starting at state s, and

$$V^\pi = \frac{1}{1-\gamma}\mathop{\mathbb{E}}_{\substack{s\sim d^\pi \\ a\sim\pi(\cdot|s) \\ s'\sim M(\cdot|s,a)}}[R(s,a,s')] \tag{2}$$

to be the discounted expected sum of rewards when following policy π on with transition dynamics M. Note that V^π here has the same definition as $J(\theta)$ from the previous sections.

Let $\rho_\pi^t \in \mathbb{R}^{|S|}$ where $\rho_\pi^t(s) = M(s_t = s|\pi)$. This is the probability of being in state s at timestep t when following policy π with dynamics M.

Let $P_\pi \in \mathbb{R}^{|S|\times|S|}$ where $P_\pi(s'|s) = \sum_a M(s'|s,a)\pi(a|s)$. This is the probability of transitioning from state s to next state s' in one step by taking actions following policy π and using transitions from dynamics M.

Let μ be starting state distribution for $\mathcal{M}$. Then, we have:

$$\begin{aligned} d^\pi &= (1-\gamma)\sum_{t=0}^{\infty}\gamma^t M(s_t = s|\pi) \\ &= (1-\gamma)\sum_{t=0}^{\infty}\gamma^t P_\pi\mu \\ &= (1-\gamma)(I-\gamma P_\pi)^{-1}\mu \end{aligned} \tag{3}$$

where the second equality holds true because $\rho_\pi^t = P_\pi\rho_\pi^{t-1}$ and the third equality can be derived from geometric series.

The goal in our proof is to give a lower bound for $V^{\pi'} - V^\pi$. We start our proof with a lemma on reward shaping.

Lemma 5.1. For any function $f : S \mapsto \mathbb{R}$ and any policy π, we have:

$$(1-\gamma)\mathop{\mathbb{E}}_{s\sim\mu}[f(s)] + \mathop{\mathbb{E}}_{\substack{s\sim d^\pi \\ a\sim\pi(\cdot|s) \\ s'\sim M(\cdot|s,a)}}[\gamma f(s')] - \mathop{\mathbb{E}}_{s\sim d^\pi}[f(s)] = 0 \tag{4}$$

The proof can be found in [4] and is reproduced in Section A.1 of the Appendix.

We can add this term to the R.H.S. of equation (2). Doing so, we get

$$V^{\pi}(s) = \frac{1}{1-\gamma}\left(\mathop{\mathbb{E}}_{\substack{s\sim d^{\pi}\\ a\sim\pi(\cdot|s)\\ s'\sim M(\cdot|s,a)}}[R(s,a,s') + \gamma f(s') - f(s)]\right) + \mathop{\mathbb{E}}_{s\sim\mu}[f(s)] \tag{5}$$

This can be seen as a form of reward shaping where the shaping function is only a function of states and not actions. Notice that if we substitute $f(s) = V^{\pi}(s)$, we get the advantage function.

5.2 Bounding Difference in State Distributions

When we update $\pi \to \pi'$, we will have different discounted state visitation distributions, d^{π} and $d^{\pi'}$, respectively. Now let us bound their difference.

Lemma 5.2.

$$\left\|d^{\pi'} - d^{\pi}\right\|_1 \leq \frac{2\gamma}{1-\gamma}\left[\mathop{\mathbb{E}}_{s\sim d^{\pi}}[D_{TV}(\pi'\|\pi)[s]]\right]$$

The proof can be found in [4] and is reproduced in Section A.2 of the Appendix.

5.3 Bounding Difference in Returns

Now, we bound the difference in value for the update $\pi \to \pi'$, namely

$$V^{\pi'} - V^{\pi}$$

Lemma 5.3. Defining the following terms,

$$L_{\pi}(\pi') = \mathop{\mathbb{E}}_{\substack{s\sim d^{\pi}\\ a\sim\pi(\cdot|s)}}\left[\frac{\pi'(a|s)}{\pi(a|s)}A^{\pi}(s,a)\right]$$

$$\epsilon_f^{\pi'} = \max_s\left[\mathop{\mathbb{E}}_{\substack{a\sim\pi'(\cdot|s)\\ s'\sim M(\cdot|s,a)}}[R(s,a,s') + \gamma f(s') - f(s)]\right]$$

we get the following upper bound

$$V^{\pi'} - V^{\pi} \leq \frac{1}{1-\gamma}\left(L_{\pi}(\pi') + \left\|d^{\pi'} - d^{\pi}\right\|_1 \epsilon_f^{\pi'}\right) \tag{6}$$

and the following lower bound

$$V^{\pi'} - V^{\pi} \geq \frac{1}{1-\gamma}\left(L_{\pi}(\pi') - \left\|d^{\pi'} - d^{\pi}\right\|_1 \epsilon_f^{\pi'}\right) \tag{7}$$

The proof can be found in [4] and is reproduced in Section A.3 of the Appendix.

We remind the reader that an upper bound for $\left\|d^{\pi'} - d^{\pi}\right\|_1$ is given in Lemma (5.2) which can be substituted into (6) and (7).

5.4 Bounding Maximum Advantage

Now we need to consider the term

$$\epsilon_f^{\pi'} = \max_s \left| \mathop{\mathbb{E}}_{\substack{a\sim\pi'(\cdot|s)\\ s'\sim M(\cdot|s,a)}} [R(s,a,s') + \gamma f(s') - f(s)] \right| \tag{8}$$

We set $f(s) = V^\pi(s)$ to be the value function at state s following policy π. Hence, we have

$$\epsilon_{V^\pi}^{\pi'} = \max_s \left| \mathop{\mathbb{E}}_{a\sim\pi'(\cdot|s)} [A^\pi(s,a)] \right| \tag{9}$$

This allows us to formulate the following:

Lemma 5.4.

$$\epsilon_{V^\pi}^{\pi'} \leq 2 \max_s D_{TV}(\pi \| \pi') \max_{s,a} |A^\pi(s,a)|$$

The proof can be found in [5] and is reproduced in Section A.4 of the Appendix.

5.5 TRPO

To recap, setting $f = V^\pi$, we have

$$\frac{1}{1-\gamma} L_\pi(\pi') - \frac{4\epsilon\gamma}{(1-\gamma)^2}\alpha^2 \leq V^{\pi'} - V^\pi \leq \frac{1}{1-\gamma} L_\pi(\pi') + \frac{4\epsilon\gamma}{(1-\gamma)^2}\alpha^2 \tag{10}$$

where

$$L_\pi(\pi') = \mathop{\mathbb{E}}_{\substack{s\sim d^\pi\\ a\sim\pi(\cdot|s)}} \left[\frac{\pi'(a|s)}{\pi(a|s)} A^\pi(s,a) \right]$$

$$\epsilon = \max_{s,a} |A^\pi(s,a)|$$

$$\alpha = \max_s D_{TV}(\pi \| \pi')$$

Comparing the above equation with the equation we got in the previous section on Relative Policy Performance Identity, we have given lower bounds and upper bounds rather than just an approximation. By optimizing the lower bound of $V^{\pi'} - V^\pi$, we get an optimization problem that guarantees improvement to our policy. Concretely, we solve the following optimization problem:

$$\max_{\pi'} L_\pi(\pi') - \frac{4\epsilon\gamma}{(1-\gamma)}\alpha^2$$

Unfortunately, solving this optimization problem results in very small step sizes. In [5], the authors change this optimization problem to a constraint optimization problem to get larger step sizes when implementing a practical algorithm. Concretely, this results in the following optimization problem:

$$\max_{\pi'} L_\pi(\pi') \quad \text{s.t. } \alpha^2 \leq \delta$$

where δ is a hyperparameter.

The max constraint in α is impractical to solve due to the large number of states. Hence, in [5], the authors use a heuristic approximation which considers the average KL divergence only. This approximation is useful since we can approximate expectation with samples but we cannot approximate max with samples. Hence, we have

$$\max_{\pi'} L_{\pi}(\pi')$$
$$\text{s.t. } \bar{D}_{KL}(\pi, \pi') \leq \delta$$

where $\bar{D}_{KL}(\pi, \pi') = \mathbb{E}_{s\sim d^{\pi}}[D_{KL}(\pi \| \pi')[s]]$

6 Exercises

Exercise 6.1. In the lecture slides, for episodic environments, the objective function is given as $J(\theta) = V^{\pi_\theta}(s_{start})$. What assumption is made in this objective function?

Solution. We make the assumption that there is a single start state, s_{start}. In general, there can be a distribution of start states, in which case there should be an expectation over the distribution of start states, μ. Hence, the more general objective function is $J(\theta) = \mathbb{E}_{s_{start}\sim\mu}[V^{\pi_\theta}(s_{start})]$

Exercise 6.2. In the infinite horizon setting, we discussed in this set of lecture notes that a possible objective function is $J(\theta) = \mathbb{E}_{(s,a)\sim P_\theta(s,a)}[r(s,a)]$. In the lecture slides, we discuss two different objectives. We could either use Average Value given by $J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s)V^{\pi_\theta}(s)$, or Average Reward per Time Step given by $J_{avR}(\theta) = \sum_s d^{\pi_\theta}(s)\sum_a \pi_\theta(a|s)r(s,a)$. Is $J(\theta)$ equivalent to $J_{avV}(\theta)$ or $J_{avR}(\theta)$ or neither?

Solution. $J(\theta)$ is equivalent to Average Reward per Time Step. In particular, the expectation over (s,a) drawn from $P_\theta(s,a)$ can be expanded into an expectation over s drawn from stationary state distribution $d^{\pi_\theta}(s)$ and an expectation over a drawn from policy $\pi_\theta(a|s)$.

Exercise 6.3. What is the key advantage of using finite difference to estimate policy gradients?

Solution. This method works for arbitrary policies, even if the policy is not differentiable.

Exercise 6.4. What is the point of the log derivative trick in policy gradients?

Solution. The log derivative trick allows the gradient estimation to be independent of the dynamics model which, in general, is unknown.

Exercise 6.5. In the derivation to prove that baseline as a function of state is unbiased, we used the fact that $\mathbb{E}_{a_t}[\nabla_\theta \log \pi_\theta(a_t|s_t)] = 0$. Provide steps to show this result.

Solution.

$$\begin{aligned}\mathbb{E}_{a_t}[\nabla_\theta \log \pi_\theta(a_t|s_t)] &= \int_a \pi_\theta(a_t|s_t)\frac{\nabla_\theta \pi_\theta(a_t|s_t)}{\pi_\theta(a_t|s_t)}da \\ &= \nabla_\theta \int_a \pi_\theta(a_t|s_t)da = \nabla_\theta 1 = 0\end{aligned}$$

Exercise 6.6. Why can't we perform the following optimization directly?

$$\max_{\pi'} J(\pi') = \max_{\pi'} J(\pi') - J(\pi) = \max_{\pi'} \mathbb{E}_{\tau\sim\pi'}\left[\sum_{t=0}^{\infty}\gamma^t A^{\pi}(s_t, a_t)\right]$$

Solution. We want to find π' but to do that we need to do rollouts using π'. This process is too slow. We need to use Importance Sampling.

Exercise 6.7. Here is some pseudocode to perform Maximum Likelihood Estimation using automatic differentiation for discrete action space.

```
logits = policy.predictions(states)
negative_likelihoods = tf.nn.softmax_cross_entropy_with_logits(
                        labels=actions, logits=logits)
loss = tf.reduce_mean(negative_likelihoods)
gradients = loss.gradients(loss, variables)
```

Given that we rollout N episodes, each with a horizon of T and there are d_a distinct actions and d_s state dimensions, what are the shapes of `actions` and `states`?

We are also given a tensor for `q_values`. What should the shape of this tensor be?

Given `q_values`, how would you change the above pseudocode to do policy gradient training?

Solution. The shape of `actions` should be $(N * T, d_a)$. The shape of `states` should be $(N * T, d_s)$. The shape of `q_values` should be $(N * T, 1)$.

```
logits = policy.predictions(states)
negative_likelihoods = tf.nn.softmax_cross_entropy_with_logits(
                        labels=actions, logits=logits)
weighted_negative_likelihoods = tf.multiply(negative_likelihoods, q_values)
loss = tf.reduce_mean(weighted_negative_likelihoods)
gradients = loss.gradients(loss, variables)
```

Hence, Policy Gradient can be viewed as a form of weighted MLE where the weight is the expected discounted return of taking action a from state s. The higher the expected discounted return, the higher the weight, the larger the gradient and hence the bigger the update.

References

[1] http://rail.eecs.berkeley.edu/deeprlcourse/static/slides/lec-5.pdf

[2] http://rail.eecs.berkeley.edu/deeprlcourse/static/slides/lec-9.pdf

[3] Wu, C., Rajeswaran, A., Duan, Y., Kumar, V., Bayen, A. M., Kakade, S., Abbeel, P. (2018). Variance reduction for policy gradient with action-dependent factorized baselines. arXiv preprint arXiv:1803.07246.

[4] Joshua Achiam, David Held, Aviv Tamar, and Pieter Abbeel. Constrained policy optimization.arXiv preprint arXiv:1705.10528, 2017.

[5] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region-policy optimization. InInternational Conference on Machine Learning, pages 1889–1897,2015.

A TRPO Proofs

A.1 Reward Shaping

Here we provide a proof for Lemma 5.1.

Proof.

$$\begin{aligned} d^\pi &= (1-\gamma)(I-\gamma P_\pi)^{-1}\mu \\ (I-\gamma P_\pi)d^\pi &= (1-\gamma)\mu \end{aligned}$$

Now, by taking a dot product with $f(s)$, we have

$$\mathop{\mathbb{E}}_{s\sim d^\pi}[f(s)] - \mathop{\mathbb{E}}_{\substack{s\sim d^\pi \\ a\sim\pi(\cdot|s) \\ s'\sim M(\cdot|s,a)}}[\gamma f(s')] = (1-\gamma)\mathop{\mathbb{E}}_{s\sim\mu}[f(s)] \tag{11}$$

Rearranging the terms completes the proof. □

A.2 Bounding Difference in State Distributions

Here we provide a proof for Lemma 5.2.

Proof. Recall from (3) that $d^\pi = (1-\gamma)(I-\gamma P_\pi)^{-1}\mu$.

Define $G = (I-\gamma P_\pi)^{-1}$, $\bar{G} = (I-\gamma P_{\pi'})^{-1}$, and $\Delta = P_{\pi'} - P_\pi$.

We have

$$\begin{aligned} G^{-1} - \bar{G}^{-1} &= (I-\gamma P_\pi) - (I-\gamma P_{\pi'}) \\ &= \gamma(P_{\pi'} - P_\pi) \\ &= \gamma\Delta \\ \Rightarrow \bar{G} - G &= \bar{G}(G^{-1} - \bar{G}^{-1})G = \gamma\bar{G}\Delta G \end{aligned}$$

This allows us to derive

$$\begin{aligned} d^{\pi'} - d^\pi &= (1-\gamma)(\bar{G} - G)\mu \\ &= (1-\gamma)\gamma\bar{G}\Delta G\mu \\ &= \gamma\bar{G}\Delta(1-\gamma)G\mu \\ &= \gamma\bar{G}\Delta d^\pi \end{aligned} \tag{12}$$

Taking the l_1-norm of (12), we have by property of operator norm

$$\left\|d^{\pi'} - d^\pi\right\|_1 = \gamma\left\|\bar{G}\Delta d^\pi\right\|_1 \le \gamma\left\|\bar{G}\right\|_1 \|\Delta d^\pi\|_1 \tag{13}$$

Let us first bound $\left\|\bar{G}\right\|_1$.

$$\left\|\bar{G}\right\|_1 = \left\|(I-\gamma P_\pi)^{-1}\right\|_1 = \left\|\sum_{t=0}^{\infty}\gamma^t P_\pi^t\right\|_1 \le \sum_{t=0}^{\infty}\gamma^t \|P_\pi\|_1^t = \frac{1}{1-\gamma} \tag{14}$$

We are left with bounding $\|\Delta d^\pi\|_1$.

$$
\begin{aligned}
\|\Delta d^\pi\|_1 &= \sum_{s'} \left| \sum_s \Delta(s'|s) d^\pi(s) \right| \\
&\leq \sum_{s',s} |\Delta(s'|s)| d^\pi(s) \\
&= \sum_{s',s} \left| \sum_a \left(M(s'|s,a)\pi'(a|s) - M(s'|s,a)\pi(a|s) \right) \right| d^\pi(s) \\
&= \sum_{s',s} \left| \sum_a (M(s'|s,a)(\pi'(a|s) - \pi(a|s)) \right| d^\pi(s) \\
&\leq \sum_{s',a,s} M(s'|s,a) \left| \pi'(a|s) - \pi(a|s) \right| d^\pi(s) \\
&= \sum_{s,a} \left| \pi'(a|s) - \pi(a|s) \right| d^\pi(s) \\
&= \sum_s d^\pi(s) \sum_a |\pi'(a|s) - \pi(a|s)| \\
&= 2 \mathop{\mathbb{E}}_{s \sim d^\pi} [D_{TV}(\pi' \| \pi)[s]] \qquad (15)
\end{aligned}
$$

Combining (13), (14) and (15), we have:

$$
\left\| d^{\pi'} - d^\pi \right\|_1 \leq \frac{2\gamma}{1-\gamma} \left[\mathop{\mathbb{E}}_{s \sim d^\pi} [D_{TV}(\pi' \| \pi)[s]] \right] \qquad (16)
$$

as desired. □

A.3 Bounding Difference in Returns

Here we provide a proof for Lemma 5.3.

Proof. Define $\delta_f(s,a,s') = R(s,a,s') + \gamma f(s') - f(s)$.

Since $V^\pi = \frac{1}{1-\gamma} \left(\mathop{\mathbb{E}}_{\substack{s \sim d^\pi \\ a \sim \pi(\cdot|s) \\ s' \sim M(\cdot|s,a)}} [\delta_f(s,a,s')] \right) + \mathbb{E}_{s \sim \mu}[f(s)]$, we have:

$$
V^{\pi'} - V^\pi = \frac{1}{1-\gamma} \left(\mathop{\mathbb{E}}_{\substack{s \sim d^{\pi'} \\ a \sim \pi'(\cdot|s) \\ s' \sim M(\cdot|s,a)}} [\delta_f(s,a,s')] - \mathop{\mathbb{E}}_{\substack{s \sim d^\pi \\ a \sim \pi(\cdot|s) \\ s' \sim M(\cdot|s,a)}} [\delta_f(s,a,s')] \right) \qquad (17)
$$

Let us first focus on the first term.

Let $\bar{\delta_f}^{\pi'} \in \mathbb{R}^{|S|}$ where $\bar{\delta_f}^{\pi'}(s) = \mathop{\mathbb{E}}_{\substack{a \sim \pi'(\cdot|s) \\ s' \sim M(\cdot|s,a)}} [\delta_f(s,a,s')]$.

Now we derive an upper bound

$$\begin{aligned}
\mathop{\mathbb{E}}_{\substack{s\sim d^{\pi'}\\ a\sim\pi'\\ s'\sim M}}[\delta_f(s,a,s')] &= \left\langle d^{\pi'}, \bar{\delta_f}^{\pi'} \right\rangle \\
&= \left\langle d^{\pi}, \bar{\delta_f}^{\pi'} \right\rangle + \left\langle d^{\pi'} - d^{\pi}, \bar{\delta_f}^{\pi'} \right\rangle \\
&\leq \left\langle d^{\pi}, \bar{\delta_f}^{\pi'} \right\rangle + \left\| d^{\pi'} - d^{\pi} \right\|_1 \left\| \bar{\delta_f}^{\pi'} \right\|_\infty \\
&= \left\langle d^{\pi}, \bar{\delta_f}^{\pi'} \right\rangle + \left\| d^{\pi'} - d^{\pi} \right\|_1 \epsilon_f^{\pi'} \qquad (18)
\end{aligned}$$

where $\epsilon_f^{\pi'} = \max_s \left[\mathop{\mathbb{E}}_{\substack{a\sim\pi'(\cdot|s)\\ s'\sim M(\cdot|s,a)}} [R(s,a,s') + \gamma f(s') - f(s)] \right]$.

We also have a lower bound

$$\begin{aligned}
\mathop{\mathbb{E}}_{\substack{s\sim d^{\pi'}\\ a\sim\pi'\\ s'\sim M}}[\delta_f(s,a,s')] &= \left\langle d^{\pi'}, \bar{\delta_f}^{\pi'} \right\rangle \\
&= \left\langle d^{\pi}, \bar{\delta_f}^{\pi'} \right\rangle - \left\langle d^{\pi} - d^{\pi'}, \bar{\delta_f}^{\pi'} \right\rangle \\
&\geq \left\langle d^{\pi}, \bar{\delta_f}^{\pi'} \right\rangle - \left\| d^{\pi} - d^{\pi'} \right\|_1 \left\| \bar{\delta_f}^{\pi'} \right\|_\infty \\
&= \left\langle d^{\pi}, \bar{\delta_f}^{\pi'} \right\rangle - \left\| d^{\pi'} - d^{\pi} \right\|_1 \epsilon_f^{\pi'} \qquad (19)
\end{aligned}$$

Now, we apply (18) to (17) to get an upper bound:

$$\begin{aligned}
(1-\gamma)(V^{\pi'} - V^{\pi}) &\leq \left\langle d^{\pi}, \bar{\delta_f}^{\pi'} \right\rangle + \left\| d^{\pi'} - d^{\pi} \right\|_1 \epsilon_f^{\pi'} - \mathop{\mathbb{E}}_{\substack{s\sim d^{\pi}\\ a\sim\pi(\cdot|s)\\ s'\sim M(\cdot|s,a)}} [\delta_f(s,a,s')] \\
&= \mathop{\mathbb{E}}_{\substack{s\sim d^{\pi'}\\ a\sim\pi'(\cdot|s)\\ s'\sim M(\cdot|s,a)}} [\delta_f(s,a,s')] - \mathop{\mathbb{E}}_{\substack{s\sim d^{\pi}\\ a\sim\pi(\cdot|s)\\ s'\sim M(\cdot|s,a)}} [\delta_f(s,a,s')] + \left\| d^{\pi'} - d^{\pi} \right\|_1 \epsilon_f^{\pi'} \\
&= \mathop{\mathbb{E}}_{\substack{s\sim d^{\pi}\\ a\sim\pi(\cdot|s)\\ s'\sim M(\cdot|s,a)}} \left[\left(\frac{\pi'(a|s)}{\pi(a|s)} - 1 \right) \delta_f(s,a,s') \right] + \left\| d^{\pi'} - d^{\pi} \right\|_1 \epsilon_f^{\pi'} \qquad (20)
\end{aligned}$$

Let us define

$$L_\pi(\pi') = \mathop{\mathbb{E}}_{\substack{s\sim d^{\pi}\\ a\sim\pi(\cdot|s)\\ s'\sim M(\cdot|s,a)}} \left[\left(\frac{\pi'(a|s)}{\pi(a|s)} - 1 \right) (R(s,a,s') + \gamma f(s') - f(s)) \right]$$

Note that if we select $f = V^\pi$, then

$$L_\pi(\pi') = \mathop{\mathbb{E}}_{\substack{s\sim d^{\pi}\\ a\sim\pi(\cdot|s)}} \left[\frac{\pi'(a|s)}{\pi(a|s)} A^\pi(s,a) \right]$$

This is because the advantage of choosing an action from the same policy is 0.

$$\mathop{\mathbb{E}}_{\substack{s\sim d^{\pi}\\ a\sim\pi(\cdot|s)}} [A^\pi(s,a)] = 0$$

We complete (20) to obtain an upper bound:

$$V^{\pi'} - V^{\pi} \leq \frac{1}{1-\gamma}\left(L_{\pi}(\pi') + \left\|d^{\pi'} - d^{\pi}\right\|_1 \epsilon_f^{\pi'}\right) \tag{21}$$

Similarly, we derive a lower bound from (17) and (19).

$$\begin{aligned} V^{\pi'} - V^{\pi} &\geq \frac{1}{1-\gamma}\left(\mathop{\mathbb{E}}_{\substack{s\sim d^{\pi}\\ a\sim\pi(\cdot|s)\\ s'\sim M(\cdot|s,a)}}\left[\left(\frac{\pi'(a|s)}{\pi(a|s)} - 1\right)\delta_f(s,a,s')\right] - \left\|d^{\pi'} - d^{\pi}\right\|_1 \epsilon_f^{\pi'}\right) \\ &= \frac{1}{1-\gamma}\left(L_{\pi}(\pi') - \left\|d^{\pi'} - d^{\pi}\right\|_1 \epsilon_f^{\pi'}\right) \end{aligned} \tag{22}$$

□

A.4 Maximum Advantage

Here we provide a proof of Lemma 5.4.

Proof. As in Section A of [5], we say (π, π') is an α-coupled policy pair if it defines a joint distribution $(a, \hat{a}) \mid s$, such that $\forall s, \Pr[a \neq \hat{a} \mid s] \leq \alpha$. Define $\alpha_{\pi,\pi'}$ such that (π, π') are $\alpha_{\pi,\pi'}$-coupled. Let $\bar{A}(s)$ be the expected value of $A^{\pi}(s, \hat{a})$ under the expectation of $\hat{a}$ drawn from policy π' given state s.

$$\bar{A}(s) = \mathop{\mathbb{E}}_{\hat{a}\sim\pi'(\cdot|s)}\left[A^{\pi}(s,\hat{a})\right] \tag{23}$$

Because $\mathbb{E}_{a\sim\pi(\cdot|s)}\left[A^{\pi}(s,a) \mid s\right] = 0$ from the definition of advantage, we rewrite (23) as

$$\begin{aligned} \bar{A}(s) &= \mathop{\mathbb{E}}_{(a,\hat{a})\sim(\pi,\pi')}\left[A^{\pi}(s,\hat{a}) - A^{\pi}(s,a)\right] \\ &= \Pr\left[a \neq \hat{a} \mid s\right] \mathop{\mathbb{E}}_{(a,\hat{a})\sim(\pi,\pi')}\left[A^{\pi}(s,\hat{a}) - A^{\pi}(s,a) \mid a \neq \hat{a}\right] \\ &+ \Pr\left[a = \hat{a} \mid s\right] \mathop{\mathbb{E}}_{(a,\hat{a})\sim(\pi,\pi')}\left[A^{\pi}(s,a) - A^{\pi}(s,a)\right] \\ &\leq \alpha_{\pi,\pi'} \mathop{\mathbb{E}}_{(a,\hat{a})\sim(\pi,\pi')}\left[A^{\pi}(s,\hat{a}) - A^{\pi}(s,a) \mid a \neq \hat{a}\right] + \Pr\left[a = \hat{a} \mid s\right] * 0 \\ &\leq 2\alpha_{\pi,\pi'} \max_a \left|A^{\pi}(s,a)\right| \end{aligned} \tag{24}$$

Therefore, we have

$$\begin{aligned} \epsilon_{V^{\pi}}^{\pi'} &= \max_s \left| \mathop{\mathbb{E}}_{\substack{s\sim d^{\pi'}\\ a\sim\pi'(\cdot|s)\\ s'\sim M(\cdot|s,a)}}\left[R(s,a,s') + \gamma f(s') - f(s)\right] \right| \\ &\leq \max_s \left| 2\alpha_{\pi,\pi'} \max_a \left|A^{\pi}(s,a)\right| \right| \\ &\leq 2\alpha_{\pi,\pi'} \max_{s,a} \left|A^{\pi}(s,a)\right| \end{aligned} \tag{25}$$

Suppose p_X and p_Y are distributions with $D_{TV}(p_X \| p_Y) = \alpha$, then there exists a joint distribution (X, Y) whose marginals are p_X, p_Y for which $X = Y$ with probability $1 - \alpha$ [5]. Taking $\alpha_{\pi,\pi'} = \max_s D_{TV}(\pi \| \pi')$ we have

$$\epsilon_{V^\pi}^{\pi'} \leq 2 \max_s D_{TV}(\pi \| \pi') \max_{s,a} |A^\pi(s,a)| \tag{26}$$

as desired. □

CS234 Notes - Lecture 11,12
Exploration and Exploitation

Anchit Gupta, James Harrison, Emma Brunskill

June 14, 2018

1 Introduction

We have talked previously about desiderata for reinforcement learning algorithms. In particular, we would like to achieve good empirical performance in addition to asymptotic convergence. In many real applications such as education, health care, or robotics, asymptotic convergence rates are not a useful metric for comparing reinforcement learning algorithms. To achieve good real world performance, we want rapid convergence to good policies, which relies on good, strategic exploration.

Online decision-making involves a fundamental tradeoff between exploitation and exploration. Exploitation is making the best possible decision (by maximizing future reward), and exploration involves taking an immediately suboptimal action to gather information. While the suboptimal action will necessarily involve a reduced amount of reward in the immediate future, it may allow you to learn better strategies that enable policy improvement in the long term.

2 Multi-Armed Bandits

We will begin by discussing exploration in the context of multi-armed bandits (MABs), as opposed to full MDPs. An MAB is a tuple $(\mathcal{A}, \mathcal{R})$, where $\mathcal{A}$ denotes a set of actions, and $\mathcal{R}$ is a collection of probability distributions, where each action has a corresponding distribution over rewards $\mathcal{R}^a(r) = \mathbb{P}(r|a)$. At each time step, an agent selects an action a_t. As in MDPs, the agent aims to maximize cumulative reward. This setting does not have the state transitions of an MDP, and thus there is no notion of delayed rewards or consequences.

Let $Q(a) = \mathbb{E}[r|a]$ denote the true expected reward for taking action a. We will consider algorithms that estimate $\hat{Q}_t(a) \approx Q(a)$. The value is estimated by Monte-Carlo evaluation,

$$\hat{Q}_t(a) = \frac{1}{N_t(a)} \sum_{t=1}^{T} r_t \mathbf{1}(a_t = a) = \hat{Q}_{t-1}(a) + \frac{1}{N_t(a)} \left(r_t - \hat{Q}_{t-1}(a) \right) \tag{1}$$

where $N_t(a)$ is the number of times action a has been taken at time t. The second equality is useful for computing estimates $\hat{Q}_t$ incrementally.

The **greedy** algorithm selects the action with the highest estimated value, $a_t^* = \arg\max_{a \in \mathcal{A}} \hat{Q}_t(a)$. However, the greedy approach may lock onto a suboptimal action forever, as an exercise try constructing such a bandit problem. Like in MDPs, we can also take a (fixed) **ϵ-greedy** algorithm selects a greedy action with probability $1 - \epsilon$, and a random action with probability ϵ. Another algorithm is **decaying ϵ_t-greedy**, in which ϵ_t decays according to some schedule.

A simple alternative to ϵ-greedy based approaches is **optimistic initialization** which initializes $\hat{Q}_0(a)$ for all $a \in \mathcal{A}$ to be some large value greater than the true value $Q(a)$. In other words, we start off "wildly optimistic" about all action choices. On each step, we can use a greedy (or ϵ-greedy) approach for selecting the action with the highest $\hat{Q}_t(a)$. Since the true rewards are all less than our initial estimates, actions that have been explored will see decreases in their estimated $\hat{Q}$ values, thus encouraging exploration among unexplored actions that still have large $\hat{Q}$ values. Thus, all actions will be tried at least once, and likely many more times. Furthermore, we can initialize $N_0(a) > 0$ to trade off how fast the optimistic initializations converge towards the true values.

2.1 Regret

These exploration strategies naturally give rise to the question of which metric we are using to compare them. Possible metrics include empirical performance (although this is environment dependent), asymptotic convergence guarantees, finite sample guarantees, or PAC guarantees. The standard in the MAB literature is typically **regret**. We will define regret and related quantities.

- Action-value $Q(a) = \mathbb{E}[r|a]$
- Optimal Value $V^* = Q(a^*) = \max_{a \in \mathcal{A}} Q(a)$
- Gap $\Delta_a = V^* - Q(a)$
- Regret $l_t = \mathbb{E}[V^* - Q(a_t)]$
- Total regret $L_t = \mathbb{E}\left[\sum_{\tau=1}^{t} (V^* - Q(a_\tau))\right] = t \cdot V^* - \mathbb{E}\left[\sum_{\tau=1}^{t} Q(a_\tau)\right]$

Minimizing total regret is thus equivalent to maximizing cumulative reward. If we define the count $\bar{N}_t(a)$ as the expected number of selections of action a, we can see that the total regret is a function of the gap and the counts,

$$L_t = \mathbb{E}\left[\sum_{\tau=1}^{t} (V^* - Q(a_\tau))\right] \tag{2}$$

$$= \sum_{a \in \mathcal{A}} \mathbb{E}\left[N_t(a)\right] (V^* - Q(a)) \tag{3}$$

$$= \sum_{a \in \mathcal{A}} \bar{N}_t(a) \Delta_a. \tag{4}$$

A good algorithm will ensure that the counts are small for large gaps. However, the gaps are not known in advance, and must be learned by interacting with the MAB.

2.2 Regret Bounds

It is desirable to guarantee that the regret of some algorithm can be quantified and bounded. There are two types of regret bounds: problem dependent and problem independent. Problem dependent regret bounds are a function of the number of times each arm is pulled and the gaps. Problem independent bounds are strictly functions of T, the total number of steps the algorithm operates for.

An algorithm that explores forever or chooses a suboptimal action forever will experience linear regret. It is desirable therefore to achieve sublinear regret. The regret bounds of the previously discussed algorithms are as follows:

- **Greedy:** linear total regret

- **Constant ϵ-greedy:** linear total regret
- **Decaying ϵ-greedy:** Sublinear regret but schedule for decaying ϵ requires knowledge of gaps
- **Optimistic initialization:** Sublinear regret if initial values are sufficiently optimistic, else linear regret.

To give a sense of how hard the problem is, it is useful to establish a lower bound on regret. Generally, the performance of any algorithm is determined by the similarity between the optimal arm and other arms. Hard problems will have similar arms with slightly different means. This can be described by the gaps Δ_a and the similarity in distributions (via the KL divergence), $KL(\mathcal{R}^a \| \mathcal{R}^{a^*})$. Then, we may establish a bound on the asymptotic total regret.

Theorem 1 (Lai and Robbins, 1985)**.** *Any algorithm on a MAB has an asymptotic lower bound on total regret of at least*

$$\lim_{t \to \infty} L_t \geq \log t \sum_{a | \Delta_a > 0} \frac{\Delta_a}{KL(\mathcal{R}^a \| \mathcal{R}^{a^*})}.$$

2.3 Optimism in the Face of Uncertainty

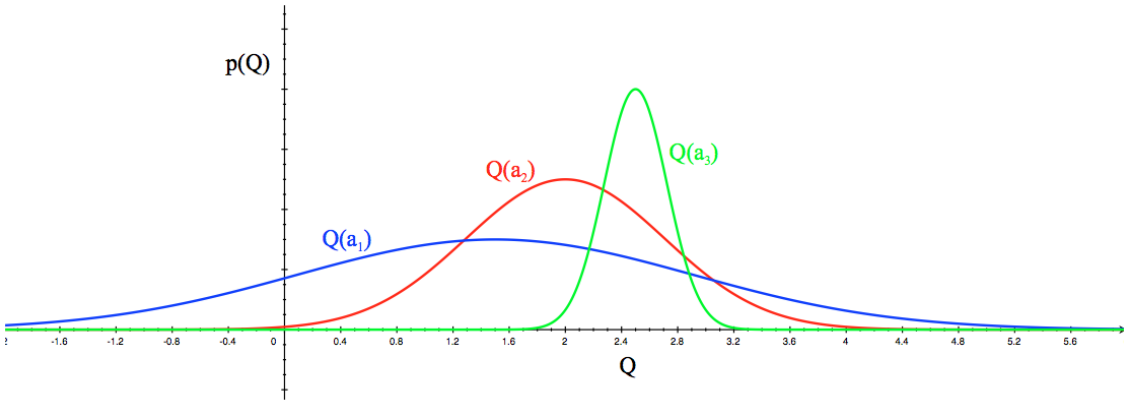


Figure 1: Reward distributions for actions a_1, a_2, a_3

Consider Figure 1, showing estimated distributions for a collection of actions. Which action should we choose? The principle of optimism in the face of uncertainty says that we should bias our choice toward actions which *may* be good. Intuitively, this will lead to either learning that the action does in fact result in high reward, or we learn that the action is not as good as we may have hoped, and we have learned valuable information about our problem.

This approach gives rise to the **Upper Confidence Bound** algorithm, which proceeds as follows. First, we estimate an upper confidence $\hat{U}_t(a)$ for each action value, such that $Q(a) \leq \hat{Q}_t(a) + \hat{U}_t(a)$ with high probability. This depends on the number of times action a has been selected. Then, we select actions to maximize the upper confidence bound

$$a_t = \arg\max_{a \in \mathcal{A}} \left\{ \hat{Q}_t(a) + \hat{U}_t(a) \right\} \tag{5}$$

This can be derived via Hoeffding's inequality.

Theorem 2 (Hoeffding's Inequality)**.** *Let $X_1, \ldots, X_t$ be i.i.d. random variables in $[0,1]$, $\bar{X} = \frac{1}{t}\sum_{\tau=1}^{t} X_\tau$ be the sample mean, and u denote a constant. Then,*

$$\mathbb{P}[\mathbb{E}[X] > \bar{X}_t + u] \leq \exp(-2tu^2).$$

Applying the above to the bandit problem, we get

$$\mathbb{P}[Q(a) > \hat{Q}_t(a) + U_t(a)] \leq \exp(-2N_t(a)U_t(a)^2). \tag{6}$$

Choosing a probability p that exceeds the true upper confidence bound, we get

$$\exp(-2N_t(a)U_t(a)^2) = p \tag{7}$$

$$U_t(a) = \sqrt{\frac{-\log p}{2N_t(a)}}. \tag{8}$$

We will reduce p as we observe more rewards. In particular, choosing $p = t^{-4}$ yields the **UCB1** algorithm,

$$a_t = \arg\max_{a \in \mathcal{A}} \left\{ Q(a) + \sqrt{\frac{2\log t}{N_t(a)}} \right\}, \tag{9}$$

which ensures asymptotically optimal action selection i.e it matches the [1] lower bound upto constant factors.

Theorem 3 (Auer et al., 2002, [2])**.** *The UCB algorithm achieves asymptotic total regret*

$$\lim_{t\to\infty} L_t \leq 8\log t \sum_{a|\Delta_a > 0} \Delta_a.$$

2.4 Bayesian Bandits

With the exception of assumption boundedness of rewards, we have so far made no assumptions about the reward distribution $\mathcal{R}$. **Bayesian bandits** exploit prior knowledge of rewards to compute the probability distribution of rewards in the next time step i.e $p[\mathcal{R}]$ where $h_t = (a_1, r_1, \ldots, a_{t-1}, r_{t-1})$, and then use this posterior to guide action selection. For example, consider a Bayesian approach to UCB. We will assume the reward distribution is Gaussian, $\mathcal{R}^a(r) = \mathcal{N}(r; \mu_a, \sigma_a^2)$, and our posterior over μ_a and σ_a^2 may be computed by

$$p[\mu_a, \sigma_a^2 | h_t] \propto p[\mu_a, \sigma_a^2] \prod_{t|a_t=a} \mathcal{N}(r_t; \mu_a, \sigma_a^2). \tag{10}$$

We can pick the action that maximizes the standard deviation of $Q(a)$,

$$a_t = \arg\max_{a \in \mathcal{A}} \left\{ \mu_a + c\frac{\sigma_a}{\sqrt{N(a)}} \right\} \tag{11}$$

An alternative approach is **probability matching**, which selects an action a according to the the probability that a is the optimal action.

$$\pi(a|h_t) = \mathbb{P}(Q(a) > Q(a'), \forall a' \neq a \mid h_t). \tag{12}$$

This is optimistic in the face of uncertainty, as uncertain actions have a higher probability of being the best action. However, this probability may be difficult to compute analytically.

An approach to probability matching is **Thompson sampling**, in which a reward distribution $\hat{\mathcal{R}}$ is sampled from the posterior for each $a \in \mathcal{A}$. Then, the action-value function $\hat{Q}(a) = \mathbb{E}[\hat{\mathcal{R}}^a]$ may be

computed, and we select the action with the highest $\hat{Q}(a)$. This approach achieves the Lai and Robbins lower bound [1], and can be extremely effective in practice.

As an example in the case of Bernoulli bandits it can be shown that if the prior is a Beta distribution then the required posterior distribution is also a Beta distribution. Hence we can start off from a beta prior over the arms play according to this in the next time step and subsequently update the parameters of the posterior. More concretely if S_i is the number of 1s observed for arm i so far and F_i is the number of 0s observed the following pseduo-code 1 implements the algorithm.

Algorithm 1 Thompson Sampling for Bernoulli MAB using Beta priors

1: **for** $i = 1, 2, \ldots$ **do**
2: For each arm $k = 1, ..N$, independently sample $\theta_k \sim Beta(S_k + 1, F_k + 1)$
3: Play arm $a_i = arg\ \max_k \theta_k$
4: Observe r_i and update S_{a_i}, F_{a_i} accordingly

A similar approach can be applied in the case of gaussian MAB with unit variance. It can be seen that that after time t the posterior for arm k equals $N(\mu_k, \frac{1}{S_k+F_k+1})$ where μ_k is the empirical average reward. As an exercise try to verify the above claim.

2.5 PAC bandits

All the algorithms described above seek to achieve regret bounds in terms of T. But this does not give us an idea of the types of mistakes the algorithm is making, it could be making large mistakes infrequently or smaller ones frequently. In many applications we may care about the bounding the number of large mistakes.

Formally a PAC algorithm guarantees that the algorithm will choose an action whose values is ϵ optimal i.e $Q(a) \geq Q(a^*) - \epsilon$ with probability atleast $1 - \delta$ on all but a polynomial number(usually in terms of ϵ, δ, N) of time steps. There exist variants of the optimism under uncertainty and Thompson sampling algorithms with such PAC guarantees.

3 Information State Search

The fundamental conflict between exploration and exploitation stems from the fact that exploration gains information which might help in the future but is sub optimal as of now. If we are able to quantify this "Value of Information" in terms of how much reward you should be prepared to pay for that information we can much more efficiently balance the exploration-exploitation trade-off. As a concrete example refer the seismologist example in the slides.

3.1 Information State Space

So far we viewed MAB's as a simple fully observable MDP with a single state.
Main Idea: Frame a MAB problem as a partially observable MDP where the hidden state is the actual mean reward of each arm. Actions as before correspond to pulling the arms. The observations we get are the rewards sampled from the hidden state. Hence finding a optimal policy for this POMDP gives us and optimal bandit algorithm. In other words and MAB instance can be reduce to an instance of POMDP planning.

A main idea from POMDP planning is that of a belief state $\tilde{s}$ which can be though of as an information state in our context. This is a posterior over the hidden state of the POMDP i.e the true mean rewards

in our case. $\tilde{s}$ is a statistic computed using the history i.e $\tilde{s_t} = f(h_t)$. Each action and it consequential observation (reward) results in a probabilistic transition to a new state $\tilde{s_{t+1}}$ in the information(belief) state space. This turns out be an MDP over the augmented information state space.

3.2 Bernoulli Bandits

In the case of simple Bernoulli bandits the information state is just the counts of the 1,0 rewards for each arm. As these counts are unbounded we now have an infinite MDP over these information states. This MDP can be solved by RL for eg. using model free techniques like Q-Learning or Bayesian model based RL. This approach is known as Bayes-adaptive RL where we seek to find the Bayes optimal exploration/exploitation trade-off given the prior i.e select the action that maximizes expected reward given current information.

The Thompson sampling algorithm described earlier for Bernoulli Bandits can be thought of in these terms with the the S_k, F_k values for each arm representing the information state and the updates to the them the transitions.

3.2.1 Gittins Index

The above Bayes-adaptive MDP can be solved using Dynamic programming and this solution is called the Gittins index. The exact solution to this is typically intractable due to the size of state space but recently simulation based search has been applied to this problem to get very good results. You will learn about such simulation methods in lecture 14.

4 Application to MDP's

All of the above methods can be extended too apply to full fledged MDP's

4.1 Optimistic Initialization

In the model free RL case systematic exploration can be very easily encouraged using optimistic initialization of the value function. We can initialize $Q(s, a)$ to $\frac{R_{\max}}{1-\gamma}$ and then run algorithms like Q learning, Sarsa, Monte-carlo etc. The high initial values ensure that each s, a pair is explored a number of times to get a better estimate of its true value.
A similar idea works in he case of model based methods where we construct and optimistic model of the MDP and initialize transitions to go to the terminal state with the maximum reward. The RMax algorithm [3] is an example of such an algorithm.

4.2 UCB: Model Based

Similar to the MAB case we can chose a action which maximises the upper confidence bound on the available actions.

$$a_t = arg \max_A Q(s_t, a) + U_1(s_t, a) + U_2(s_t, a)$$

where U_1 is the uncertainty in the policy evaluation which is easy to quantify and the second term arises from the uncertainty in the policy improvement step and is typically hard to compute.

4.3 Thompson Sampling: Model Based

As before this implements probability matching i.e

$$\pi(a|h_t) = \mathbb{P}(Q(s,a) > Q(s,a'), \forall a' \neq a \mid h_t). \tag{13}$$

We can use Bayes law to compute the posterior $\mathbb{P}[P, R|h_t]$, and then sample a MDP from this distribution, solve it using a planning algorithm and act accordingly.

4.4 Information State Search

As in the MAB case we can construct a augmented MDP with the information state appended to the MDP's state to get an augmented state space to get a Bayes adaptive MDP. Solving this would then give us the optimal exploration/exploitation trade off. However the size of the state space restricts us to using simulation based search methods to solve these.

5 Conclusion

In summary there are a variety of exploration techniques some more principled than others.

- Naive exploration - ϵ-greedy methods
- Optimistic Initialization - A very simple idea that usually works very well
- Optimism under uncertainty - prefer actions with uncertain values eg. UCB
- Probability matching - pick action with largest probability of being optimal eg. Thompson sampling
- Information state space - construct augmented MDP and solve it, hence directly incorporating the value of information.

References

[1] Lai, Tze Leung, and Herbert Robbins. "Asymptotically efficient adaptive allocation rules." Advances in applied mathematics 6.1 (1985): 4-22.

[2] Auer, Peter, Nicolo Cesa-Bianchi, and Paul Fischer. "Finite-time analysis of the multiarmed bandit problem." Machine learning 47.2-3 (2002): 235-256.

[3] Brafman, Ronen I. and Tennenholtz, Moshe. "R-max - a General Polynomial Time Algorithm for Near-optimal Reinforcement Learning" JMLR (2003) 213-231.

CS234 Notes - Lecture 14
Model Based RL, Monte-Carlo Tree Search

Anchit Gupta, Emma Brunskill

June 14, 2018

1 Introduction

In this lecture we will learn about model based RL and simulation based tree search methods. So far we have seen methods which attempt to learn either a value function or a policy from experience. In contrast model based approaches first learn a model of the world from experience and then use this for planning and acting. Model-based approaches have been shown to have better sample efficiency and faster convergence in certain settings. We will also have a look at MCTS and its variants which can be used for planning given a model. MCTS was one of the main ideas behind the success of AlphaGo.

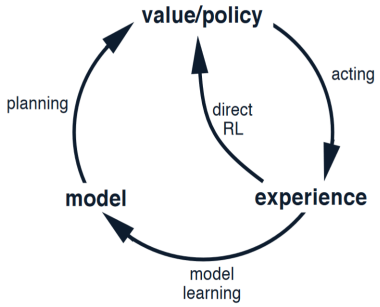


Figure 1: Relationships among learning, planning, and acting

2 Model Learning

By model we mean a representation of an MDP $< S, A, R, T, \gamma >$ parametrized by η. In the model learning regime we assume that the state and action space S, A are known and typically we also assume conditional independence between state transitions and rewards i.e

$$P[s_{t+1}, r_{t+1}|s_t, a_t] = P[s_{t+1}|s_t, a_t]P[r_{t+1}|s_t, a_t]$$

Hence learning a model consists of two main parts the reward function $R(.|s, a)$ and the transition distribution $P(.|s, a)$.

Given a set of real trajectories $\{S_t^k, A_t^k R_t^k, ..., S_T^k\}_{k=1}^K$ model learning can be posed as a supervised learning problem. Learning the reward function $R(s, a)$ is a regression problem whereas learning the transition function $P(s'|s, a)$ is a density estimation problem. First we pick a suitable family of parametrized models, these may include Table lookup models, linear expectation, linear gaussian,

gaussian process, deep belief network etc. Subsequently choose an appropriate loss function eg. mean squared error, KL divergence etc. to optimize the choice of parameters that minimize this loss.

3 Planning

Given a learned model of the environment planning can be accomplished by any of the methods we have studied so far like value based methods, policy search or tree search which we describe soon.

A contrasting approach to planning uses the model to only generate sample trajectories and methods like Q learning , Monte-Carlo control, Sarsa etc. for control. These sample based planning methods are often more data efficient.

The model we learn can be inaccurate and as a result the policy learned by planning for it would also be suboptimal i.e Model based RL is dependent on the quality of the learned model. Techniques based on the exploration/exploitation section can be used to explicitly reason about this uncertainty in the model while planning. Alternatively if we ascertain the model to be wrong in certain situations model free RL methods can be used as a fall back.

4 Simulation based search

Given access to a model of the world either an approximate learned model or an accurate simulation in the case of games like Go, these methods seek to identify the best action to take based on forward search or simulations. A search tree is built with the current state as the root and the other nodes generated using the model. Such methods can give big savings as we do not need to solve the whole MDP but just the sub MDP starting from the current state. In general once we have gathered this set of simulated experience $\{S_t^k, A_t^k R_t^k, ..., S_T^k\}_{k=1}^K$ we can apply model free methods for control like Monte-Carlo giving us an Monte-Carlo search algorithm or Sarsa giving us a TD search algorithm.

More concretely in a simple MC search algorithm given a model M and a simulation policy π, for each action $a \in A$ we simulate K episodes of the form $\{S_t^k, a, R_t^k, ..., S_T^k\}_{k=1}^K$ (following π after the first action onwards). The $Q(s_t, a)$ value is evaluated as the average reward of the above trajectories and we subsequently pick the action which maximizes this estimated $Q(s_t, a)$ value.

4.1 Monte Carlo Tree Search

This family of algorithms is based on two principles. 1) The true value of a state can be estimated using average returns of random simulations and 2) These values can be used to iteratively adjust the policy in a best first nature allowing us to focus on high value regions of the search space.

We progressively construct a partial search tree starting out with the current node set as the root. The tree consists of nodes corresponding to states s. Additionally each node stores statistics such as the total visitation count $N(s)$, a count for each pair $N(s, a)$ and the monte-carlo $Q(s, a)$ value estimates. A typical implementation builds this search tree until some preset computational budget is exhausted with the value estimates (particularly for the promising moves) becoming more and more accurate as the tree grows larger. Each iteration can be roughly divided into four phases.

1. Selection: Starting at the root node, we select child nodes recursively in the tree till a non-terminal leaf node is reached.
2. Expansion: The chosen leaf node is added to the search tree
3. Simulation: Simulations are run from this node to produce an estimate of the outcomes

4. Backprop: The values obtained in the simulations are back-propagated through the tree by following the path from the root to the chosen leaf in reverse and updating the statistics of the encountered nodes.

Variants of MCTS generally contain modifications to the two main policies involved -

- **Tree policy** to chose actions for nodes in the tree based on the stored statistics. Variants include greedy, UCB
- **Roll out policy** for simulations from leaf nodes in the tree. Variants include random simulation, default policy network in the case of AlphaGo

Algorithm 1 General MCTS algorithm

function MCTS(s_0)
  Create root node v_0 corresponding to s_o
  while within computational budget **do**
    $v_k \leftarrow TreePolicy(v_o)$
    $\Delta \leftarrow Simuation(v_k)$
    $Backprop(v_k, \Delta)$
  return $arg\max_a Q(s_o, a)$

These steps are summarized in 1 we start out from the current state and iteratively grow the search tree, using the tree policy at each iteration to chose a leaf node to simulate from. Then we backpropagate the result of the simulation and finally output the action which has the maximum value estimate form the root node.

A simple variant of this algorithm choses actions greedily amongst the tree nodes in the first stage as implemented in 2, and generates roll outs using a random policy in the simulation stage.

Algorithm 2 Greedy Tree policy

function TREEPOLICY(v)
  $v_{next} \leftarrow v$
  if $|Children(v_{next})| \neq 0$ **then**
    $a \leftarrow arg\max_{a \in A} Q(v, a)$
    $v_{next} \leftarrow nextState(v, a)$
    $v_{next} \leftarrow TreePolicy(v_{next})$
  return v_{next}

Various modifications of the above scheme exist to improve memory usage by adding a limited set of nodes to the tree, smart pruning and tree policies based on using more complicate statistics stored in the nodes.

A run through of this is visualized in 2. We start out from the root node and simulate a trajectory which gives us a reward of 1 in this case. We then use the tree policy which greedily selects the node to add to the tree and subsequently simulate an episode from it using the simulation(default) policy. This episode gives us a reward of 0 and we update the statistics in the tree nodes. This process is then repeated, to check your understanding you should verify in the below example that the statistics have been updated correctly and that the tree policy chooses the correct node to expand.

The main advantages of MCTS include

- Its tree structure makes it massively parallelisable.
- Dynamic state evaluation i.e solve MDP from current state onwards unlike DP.

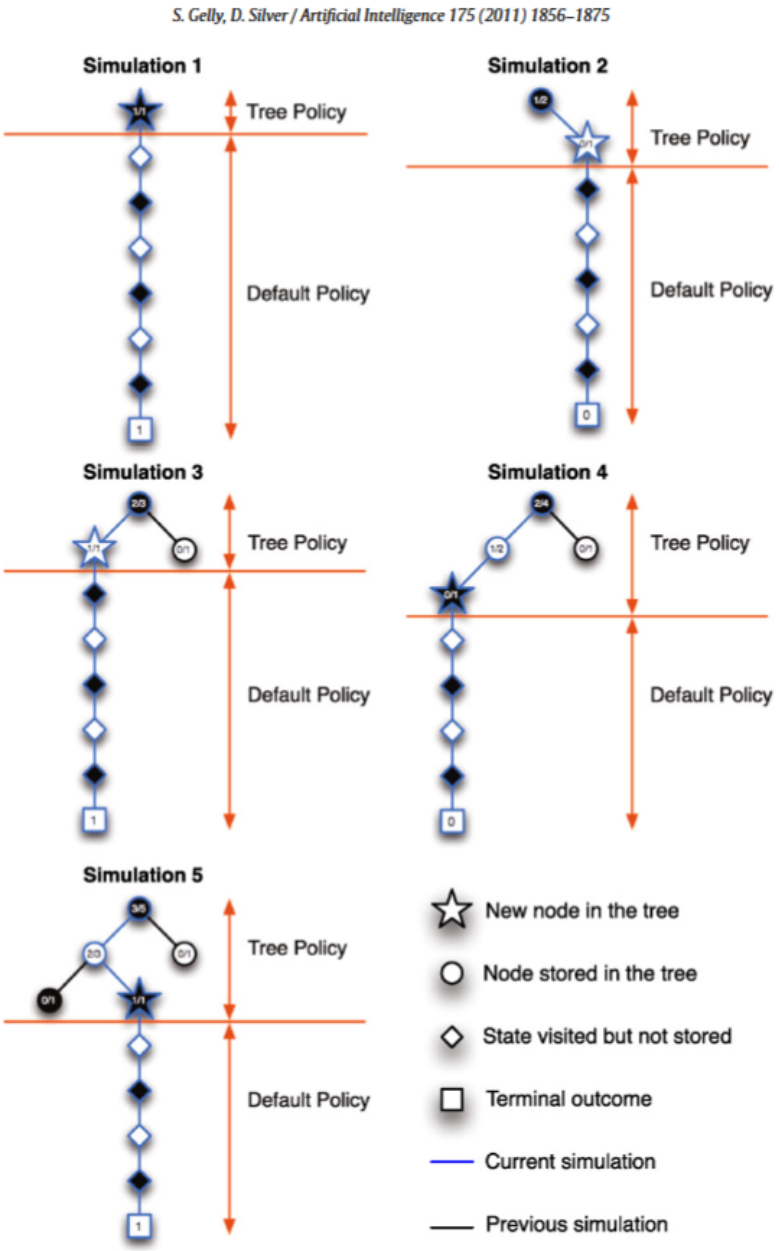


Figure 2: Demonstration of a simple MCTS. Each state has two possible actions (left/right) and each simulation has a reward of 1 or 0. At each iteration a new node (star) is added into the search tree. The value of each node in the search tree (circles and star) and the total number of visits is then updated

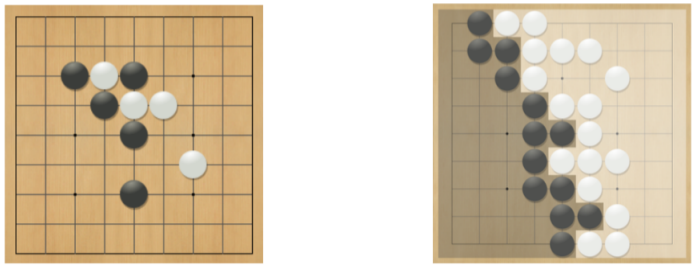


Figure 3: Go

- No need for domain specific engineering i.e it works for black box models and needs only samples
- Efficiently combines planning and sampling to break the curse of dimensionality in games like Go.

4.2 Upper Confidence Tree Search

Similar to the multi armed bandit case using a greedy policy as the tree policy can often be suboptimal, making us avoid actions after even one bad outcome even though there is significant uncertainty about its true value. As an example consider the rightmost node in 2 from which we do a single rollout, receive a 0 reward and never visit it again even though this reward might have just been due to bad luck. To fix this we can apply the principle of optimism under uncertainty to MCTS using the UCB algorithm. More specifically the tree policy instead of picking the action greedily picks the action which maximizes the upper confidence bound on the value of the action i.e $Q(s,a) + \sqrt{\frac{2 \log N(s)}{N(s,a)}}$

Refer 3 for pseudo-code of the tree policy used in UCT which can be plugged into the general MCTS algorithm described earlier. The $nextState(s,a)$ function uses the model of the MDP to sample a next state when picking action a from state s.

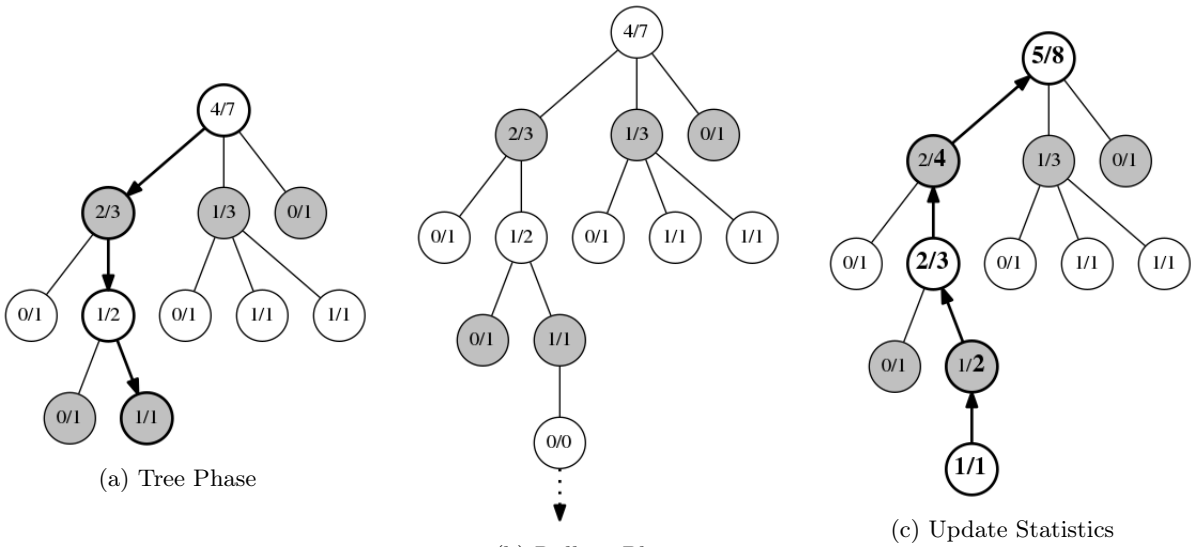
Algorithm 3 Upper Confidence Tree policy

function TREEPOLICY(v)
  $v_{next} \leftarrow v$
  if $|Children(v_{next})| \neq 0$ **then**
    $a \leftarrow arg\max_{a \in A} Q(v,a) + \sqrt{\frac{2 \log N(v)}{N(v,a)}}$
    $v_{next} \leftarrow nextState(v,a)$
    $v_{next} \leftarrow TreePolicy(v_{next})$
  return v_{next}

5 Case Study: Go

Go is the oldest continuously played board game in the world. Solving it had been a long standing challenge for AI and before AlphaGO traditional game tree search algorithms had failed to achieve professional human level performance on it. Go is a two player game(B/W), it is played on a 19x19 board (smaller variants exist). Black, White alternatingly place down stones on the board. The main goal of the game is to surround and capture territory. Additionally stones surrounded by the opponent are removed.

The simplest reward function gives a reward of +1 if Black wins and 0 if White wins on terminal



(a) Tree Phase

(b) Rollout Phase

(c) Update Statistics

Figure 4: UCT on Go, colored nodes represent the players and each node contains the probability of that player winning. (Image credits: [3])

states and 0 elsewhere. As a result the goal of the Black player is to maximize the reward and white tries to minimize it. Given a policy $\pi =< \pi_B, \pi_W >$ for both players the value function $V_\pi(s) = E_\pi[R_T|s] = P[\text{Black wins}|s]$ and the optimal value function is $V^*(s) = \max_{\pi_B} \min_{\pi_W} V_\pi(s)$.

5.1 MCTS for Go

Go being a two player game warrants some fairly natural extensions to the previosuly discussed MCTS algorithm. We now build a minimax tree with nodes alternating players across levels. The white nodes seek to minimize while the black ones maximize. We use UCB as described above at the black nodes and LCB (Lower Confidence Bound) i.e $\min_a Q(s,a) - \sqrt{\frac{2 \log N(s)}{N(s,a)}}$ at the white nodes as they seek to minimize the reward.

Consider the following state of the tree in 4a with the statistics (win/total games) recorded in the nodes and the colors representing the players. The first phase of the algorithm or the Tree policy would use these statistics in the nodes, treating each of them as an independent MAB instances and would sequentially choose actions using UCB(LCB) from the root onwards. These are highlighted with bold arrows (taking $c = \sqrt{2}$).

Once we reach a leaf node in this tree 4b we use our rollout policy to simulate a game. The outcome of this game is then back propagated through the tree 4c and the statistics updated.

This process is continued until desired and the best action to take from the root returned subsequently. For detailed pseudocode you can refer to [4] and [3] for a python implementation.

AlphaGo [1] used a deep policy network for the rollout phase, this allows the simulations to be much more realistic than just using random rollouts. Also in a complex game like Go it is not feasible to simulate till completion, early stopping is used along with a Value network to get the required win probabilities. Recently AlphaGo Zero [2] has been proposed which uses a single network to output both the policy and the value function and is trained purely using self play with no expert knowledge built in. This gets even more impressive performance than AlphaGo.

References

[1] Silver, D. et al. "Mastering the game of Go with deep neural networks and tree search." Nature 529, 484–489 (2016).

[2] Silver, D. et al. "Mastering the game of Go without human knowledge" Nature doi:10.1038/nature24270 (2017).

[3] Bradberry, J "Introduction to Monte Carlo Tree Search" https://jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/

[4] Gelly, S. et al. "Monte-Carlo tree search and rapid action value estimation in computer Go" Artificial Intelligence 175 (2011) 1856–1875