REINFORCEMENT LEARNING

Dpt: Signals, Systems and Radiocommunications

Contents

Part I

Fundamentals of Reinforcement Learning

Chapter 1

Introduction

This course intends to establish the fundamentals of a challenging and very relevant learning procedure that is envisioned to play a important role in Machine Learning and in particular in Artificial Intelligence (AI). The organization of the course is a combination of theoretical description of foundations and algorithms with some selected problems to be solved in Matlab. Matlab has been selected as a prototyping language because it is familiar to students, its interface is simple and offers powerful debugging capabilities. In a second step, some of these programs / algorithms will be also written in Python, which is, at this moment, one of the most demanded software packages in the field. Python is also friendly and permits running complex iterative algorithms more efficiently than Matlab. Moreover, there are several packages available with many reinforcement learning (RL) algorithms already implemented (e.g., RLLab, Keras-RL, PyBrain or RLPy) and, perhaps more importantly, there are also packages with multitude of environments specifically designed for developing and benchmarking RL algorithms (e.g., OpenAI Gym, OpenAI Universe, Project Malmo or DeepMind Lab).

1.1 Sequential decision problem

The problem we are facing in this course is how an agent can learn to predict and control the response of its environment. This discipline is generally known as *stochastic optimal control*. When the model of the environment is known, we say that the agent can *plan* its sequence of optimal decisions. Dynamic Programming (DP) is a powerful tool for solving this kind of planning problems. However, there are many cases where the model is unknown. In this case, rather than planning, we say that the agent has to *learn* from interaction with the environment. In particular, the agent aims to predict

how the environment will react to each action at every state and to figure out which actions would yield the most favourable response by repeating testing different actions at the same situations. This problem is known as reinforcement learning (RL) because the agent will tend to reinforce those actions that have led to more favourable responses. In the general case, the environment changes its state according to its current state and agent's action. Hence, the agent's actions may affect not only to the immediate response at the current interaction, but it may also influence every future response.

A classical approach to the RL problem uses the Markov Decision Process (MDP) formulation as a simple model of the interaction between the agent and its environment. The agent is able to sense the state of the environment and to take actions to obtain some reward. The environment state transitions are governed by some rule. Any algorithm that can solve this problem of learning to accomplish a goal from interaction with the environment can be considered as an RL method. The main elements of the RL problem are:

- the observables seen by the agent, O_t, that indicate the state of the environment, S_t, which belongs to some state set $\mathcal{S}$;
- the actions taken by the agent, A_t, that belong to some action set $\mathcal{A}$;
- and rewards given by the environment,$R_t \in \Re$.

The state and action sets can be discrete or subsets of real vector spaces. The reward R_t is a scalar feedback signal that indicates how well the agent is doing at step t. The agent's job is to maximize the long term cumulative reward assuming the so named "reward hypothesis: All goals can be described by the maximization of expected cumulative reward." Therefore, the reward signal must be designed to represent properly the problem objective that agent aims to solve. For instance, if the goal of the agent is to find the exit of a maze in the shortest time, then the agent could get negative reward at every time-step and zero reward only when it finds the exit. Another more complex example is when the agent drives a driver-less car. In this case, the reward could be positive when it gets to destination and negative when it collides another vehicle or gets out of the road. Figure ([fig:The-agent--environ]) shows schematically the interaction between agent and environment.

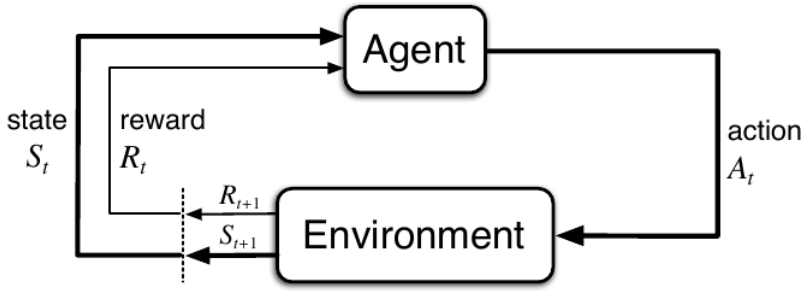


Figure 1.1: The interaction between agent and environment in RL

Note that this learning from interaction approach is somehow different from standard supervised or unsupervised learning (although we will see connections too). Indeed, the classical taxonomy for machine learning methods distinguishes between supervised, unsupervised and reinforcement learning problems and algorithms. The standard argument is that RL is different from supervised learning in the sense that we do not have a set of labelled examples provided by an external knowledgeable supervisor that informs about the correct action the system should take to that situation. But this argument is debatable because the reward signal can be seen as some form of weak supervision. The standard argument also says that RL is different from unsupervised learning because we are trying to maximize a reward signal instead to finding a hidden structure in unlabelled data. Although the standard argument is correct, we remark that it applies more to the early years of RL, where the research was focused in toy problems with very simple and fully observable state-spaces. For more realistic problems, with large state-action spaces and states that can be only partially observed from features, researchers have found that using supervised methods for predicting the environment response or including unsupervised learning as a preprocessing stage can boost the performance of the RL algorithm. Indeed, most recent RL algorithms use supervised and/or unsupervised methods as internal subroutines.

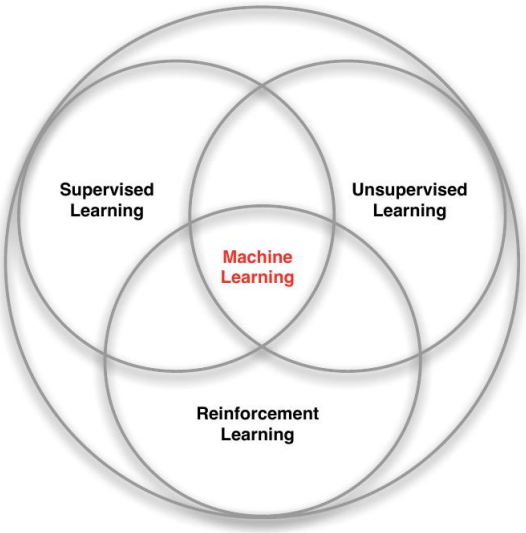


Figure 1.2: Branches of Machine Learning

What is unquestionable is that there are some special features that together make RL different from other machine learning problems. These features are:

- **Goal**. This is usually defined as some environment state desired by the agent, so that the agent will take a sequence of actions that cause state transitions until the environment reaches the desired goal state. Another definition could be in terms of the reward signal, such that the agent's goal is to maximize its expected long term cumulative reward. Clearly, both definitions must be consistent, i.e., the agent must take actions that lead the environment to the desired state, such that the agent receives the highest possible cumulative reward.

- **Sequential**. This is a sequential decision problem. Agent's actions influence the subsequent observables and rewards. Hence, the agent learns from correlated data.

- **Reward signal**. This is the feedback that the agent receives from the environment and gives some hint on whether the agent is approaching its goal.

- **Delayed feedback**. The actions influence both the current and future rewards. Sometimes, the reward obtained at some time-step is not significant but its influence on future rewards could be. Thus, the agent may have to sacrifice immediate reward in order to gain more

long-term reward (e.g., in order to get out of a maze, the agent has to take a detour rather than going toward the shortest distance to the exit).

- **Exploration vs. exploitation**. The dilemma between exploration and exploitation is as follows. When the agent faces some situation, it could make the best decision given its current information (this is exploitation). Since we consider that the state transitions and reward signal are stochastic, an alternative decision could yield a better, worse or equal result, but in any case, the agent will gather more information and gain more knowledge about the environment response (this is exploration). In other words, the agent can take actions that have already proved effective in the past (exploitation) or to try to discover actions that may yield even better results (exploration). In practice, RL algorithms have to establish some trade-off between exploitation and exploration.

Examples of RL problems in *engineering*:

- AI bots playing board games (e.g., they can defeat the world champion at Backgammon or Go).
- Manage an investment portfolio.
- Drive a driver-less car.
- Control a power station.
- Chooses packet routes for network routing scenarios.
- AI bots that can support customers.
- AI bots that play video games.

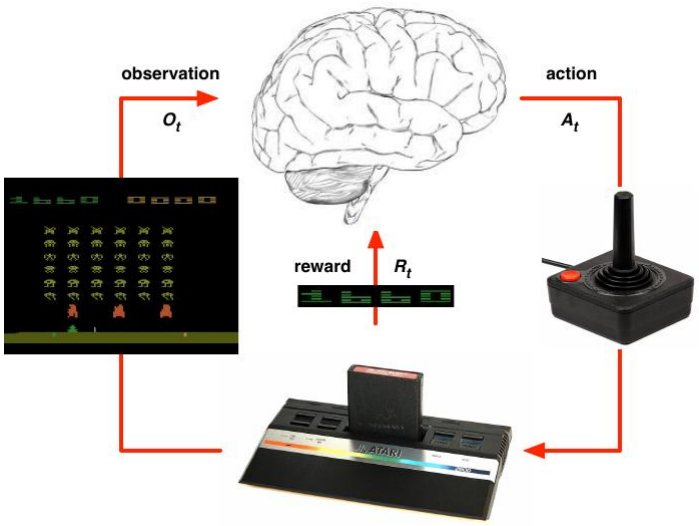


Figure 1.3: Example of an RL application

1.2 Important concepts in sequential learning

1.2.1 Agent and environment

We consider sequential learning problems where, at each time step t, the agent executes action A_t, perceives observation O_t and obtains scalar reward R_t. Hence, from the complementary point of view of the environment, the action is an input, while the observation and reward are outputs.

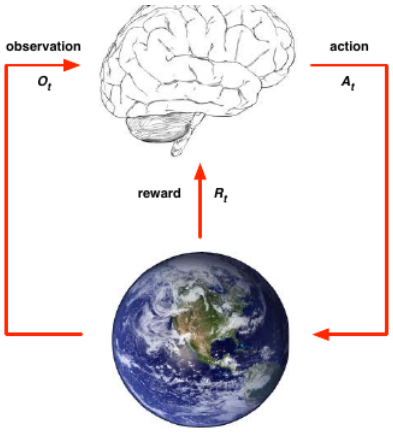


Figure 1.4: Agent and Environment

The *history* at time t, denoted H_t, is the sequence of observations, actions and rewards, i.e., $H_t = O_1 = o_1, R_1 = r_1, A_1 = a_1, \cdots, A_{t-1} = a_{t-1}, O_t = o_t, R_t = r_t$, which are all observable variables at time t. Note that we use capital letters to denote random variables and lower case letters to denote realizations of these variables. Now, the question relevant to the agent is: what will happen at time $t + 1$? We say that this question is relevant because if the agent can predict the future sequence of rewards for every possible sequence of actions, then it will be able to take the actions that maximize the cumulative reward. It is clear that *what happens next* depends on the history (i.e., on the trajectory in the state-action space). We use the term environment *state* at time t, denoted S_t, as the part of the history H_t that determines (in a stochastic sense) the observation and reward at time $t + 1$, i.e., $O_{t+1}$and R_{t+1}. More formally, the state is a function of the history $S_t = f(H_t)$. In our course we are going to assume full observable environments, such that $O_t = S_t$. Indeed, when we say that the environment is modelled as an MDP, we implicitly assume that the agent can fully observe the state (i.e., an MDP assumes that $O_t = S_t$). One key feature of MDPs is that they satisfy the *Markov property* for the state transitions, which means that the transition from the current state to the next one only depends on the current state and the current action, rather than on the whole history, i.e.:

$$\mathbb{P}(S_{t+1} \mid H_t) = \mathbb{P}(S_{t+1} \mid S_t, A_t) \tag{1.1}$$

A more general case would be when the state is only be partially visible to the agent, $O_t \neq S_t$, and the state is defined as the whole history, such that $S_t = H_t$. Examples of formulations that extend the MDP model to this more general partially-observable problem include the Partially Observable Markov Decision Process (POMDP) and the Predicted State Representations (PSR). This partially observable case does not satisfy the Markov property (1.1) for the observations (i.e., $\mathbb{P}(O_{t+1} \mid H_t) \neq \mathbb{P}(O_{t+1} \mid O_t, A_t)$), although smart domain transformation can yield something similar.

In this course we will classify problems based on the size of the state action sets. When the state and action sets are discrete and small, we talk about *small-scale problems*, and they are characterized by the fact that most of the involved functions can be represented as lookup tables. This problems are easily solved. However, the tabular representation becomes computationally unmanageable when the state-action sets are very large or continuous. In this case, we talk about *large-scale problems*. In this case, the efficient representation of these large or continuous spaces becomes an issue and we have to rely on approximations. Both small and large scale problems will be described in Part (II) and Part (III) of this document, respectively.

1.2.2 Policies and value functions

Apart of the agent and the environment, we identify three main elements of any RL problem: *polic*ies, *reward signal*, and *value functions.*

A *policy* π dictates how the agent behaves and is defined as a function of the states. A policy could be deterministic or stochastic. If the policy is deterministic, it is formally defined as a mapping from states to actions, such that the action taken at any state is deterministically given as the output of the policy. For instance, if we have $S_t = s$, then the action taken by the agent at time t will be given by $a = \pi(s)$. If the policy is stochastic, then the policy is a conditional probability distribution, such that the the probability of taken action a given that $S_t = s$, is given by$\pi(a \mid s) = \mathbb{P}(A_t = a \mid S_t = s)$.

As we have already mentioned, the *reward* signal, R_t, defines the goal for an RL problem. At each time step, the environment sends a scalar signal to the agent. The agent's sole objective is to maximize the long term reward. This idea of long term reward is formally captured by *value functions.* A state value function, is a function of the environment state and the agent's policy, usually denoted $v^{\pi}(s)$, and represents the total reward that an agent can expect to accumulate in the future, when the environment is at state s and the agent behaves according to policy π. Hence, since the agent aims to find the policy that maximizes the long term reward— i.e., the value function— the learning process should be driven by the value function, rather than by the immediate rewards. However, while rewards are given directly by the environment, the value function must be estimated from H_t (i.e., the history of state-action-rewards), and this estimate should be updated over its entire lifetime. In fact, most of the most efficient RL algorithms include some method for efficiently estimating the value function. As we will see in Chapter (3), the reason is that the policy can be improved once the value function is known (however, we will also review other approaches intending to directly estimate the optimal policy without estimating the value function, but these approaches usually suffer from high variance and can be usually improved by including the value function estimation as an intermediate step).

Indeed, when solving an RL problem, we usually consider two different problems:

- *Prediction.* It refers to the problem of estimating the value function $v^{\pi}(s)$ for some policy π and every possible state s.
- *Control.* It refers to the task of learning the optimal policy that maximizes the value function. We denote the optimal policy and optimal value function as $\pi^*$and v^*, respectively.

As we will see, the prediction and control tasks are usually intertwined and different algorithms have different methods for solving each one. Example (1.1) shows the value-function for the random policy, as well as the optimal policy and optimal value function for the Gridworld environment.

Example 1.1. In this rectangular grid, the cells correspond to the states of the environment. Think a little bit about it. Suppose that the agent is embedded into a robot. We remark that by agent we mean the AI software that is able to learn how to maximize the reward. Since the reward obtained by the agent depends on where the chassis of the robot is located, the chassis of the robot can be considered as part of the environment. From this point of view, it makes sense that the cells —that denote the possible positions of the chassis, which we assumed can be fully observed— represent the states of the environment. At each cell, four actions are possible: *north, south, east, west,* which deterministically cause the agent to move in the respective direction of the grid. Note that the future location depends only on the current location (hence our consideration of location as state, without having to consider the whole history). Actions that would take the agent off the grid leave its location unchanged, but also result in a negative reward of -1. Other actions result in a reward of 0 except for cells denoted A and B. For cell A all four actions yield a reward of $+10$ and take the agent to A'. For cell B all four actions yield a reward of $+5$ and take the agent to B'.

- a) Visualize the solution of the prediction problem where the agent selects all four actions with equal probability in all states. We have not shown how to compute value functions yet, so do not worry if you do not understand how we got to these values. The idea is to try to understand the diagrams, appreciate the relative differences between values and give an intuitive explanation for them.ç

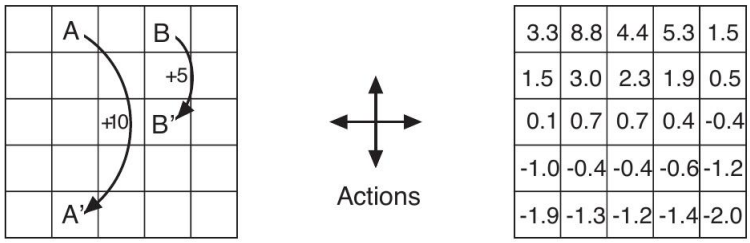


Figure 1.5: Prediction task. Gridworld example

- b) Visualize the solution for the optimum policy and value function

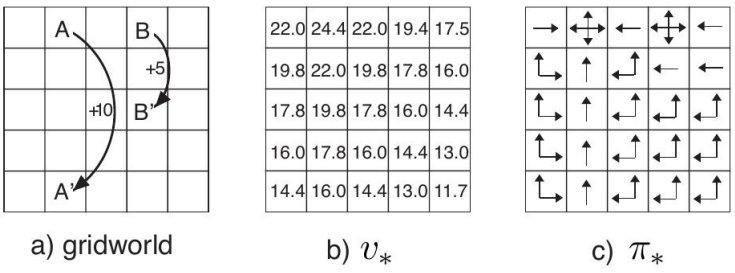


Figure 1.6: Control task. Gridworld example

Although state value functions $v^{\pi}(s)$ play a key role in RL, it is often more convenient to use a related function, known as state-action value function and denoted $q^{\pi}(s,a)$, that represents the value (i.e., the expected cumulative reward) of state s when taking action a and following policy π thereafter . Both functions are interrelated as we will see in subsequent chapters.

1.3 Taxonomies of RL procedures

The main challenge of DP and RL is therefore to find the optimal policy that maximizes the long-term performance given by the value function. The main difference between DP and RL is that the former assumes that the model is known—including the state-to-state transition probabilities as well as the reward distribution for every possible transition—, while the latter considers the model unknown and learns from interaction (sample-based or trajectory based) with the environment.

A taxonomy of DP and RL algorithms derived from the algorithmic approach is detailed now and will be followed in the following chapters:

1. *Value iteration.* The agent searches the optimal value function. Then, the optimal value function is used to compute an optimal policy.

2. *Policy iteration.* The agent chooses a policy and computes its value function. Then, it uses these value function to improve the policy. The process is repeated until convergence to the optimal policy.

3. *Policy search.* The agent uses optimization techniques to search the optimal policy directly in the policy space.

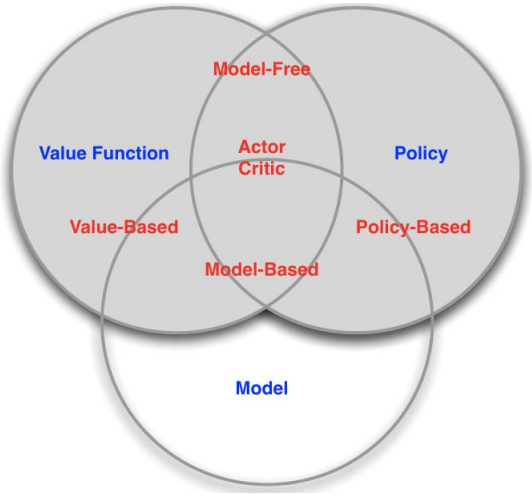


Figure 1.7: RL approaches taxonomy

This taxonomy is not exclusive (for instance, value iteration can be considered as an extreme case of policy iteration). Moreover, many algorithms combine ideas from more than one of these classes.

From an implementation perspective, we will also distinguish between *offline* and *online* algorithms:

1. Offline algorithms. The agent learns from data collected in advance, before the algorithm starts. The main challenge here is that the available data may not be very representative of the optimal policy, leading to high variance solutions. On the other hand, having all the data in advance could be beneficial for batch processing, which usually offers higher sample efficiency than streaming algorithms.

2. Online algorithms. The agent learns at the same time that gathers the data. This offers the possibility of collecting data that is more relevant to the optimal policy. On the other hand, online algorithms are usually associated with streaming data, which usually offers less sample efficiency.

Again, we have some flexibility with this taxonomy and we can find batch algorithms working on online data, online algorithms learning from policies that are not very relevant to the optimal policy and so on.

1.4 Outline of the document.

This document has three main parts:

1. *Part 1.* Introduces the main concepts and ideas associated to RL problems whose main priority is to let the student become familiar with notation and mathematical fundamentals. The content is complemented with mainly theoretical examples to be solved by the student.

2. *Part 2.* Includes the planning and learning problems for small discrete states-action sets that can be efficiently described in tabular form. The content is complemented with some programming exercises in Matlab for some of the most representative cases.

3. *Part 3.* Addresses problems with large state-action sets (e.g., high dimensional vector spaces), where tabular methods are computationlly too expensive and alternative approximate procedures have to be used. This part includes some programming exercises in Matlab for some of the most representative cases, as well as a programming exercises in Python for a state-of-the-art method.

Chapter 2

Multiarmed bandits

This chapter presents *multiarmed bandit problems*, which can be seen as a simple (perhaps the simplest) case of the RL problem. We say it is simple because the state space contains only one single state. Since the environment can be only in one state, there is no state transitions at all. The L-armed Bandit Problem (so named by analogy to a slot machine) the agent can choose between L possible actions. You can think Think the following analogy: you are in a casino with a bag of coins and there is a row of L slot machines, your goal is to decide a strategy in the form of to which machine you play each coin of the bag in order to maximize the return, where the prize of each machine follows its own stationary probability distribution.

Again, this model is much simpler than the MDPs we introduced in the previous chapter. However, multiarmed bandit problems are interesting since they allow to introduce and apply concepts that will be used in more general RL problems, like value functions and the exploration vs exploitation tradeoff.

For simplicity, suppose that the agent's goal is to maximize the expected reward (this makes sense, e.g., when all arms have same variance or when there are unlimited number of coins in the agent's bag). Let $\mathcal{A} = 1, \ldots, L$ denote the set of slot machines and, hence, the set of possible actions for the agent. Let A_t and R_t denote the action taken and reward obtained by the agent at time t. We define the action value function $q : \mathcal{A} \rightarrow \Re^L$ as the expected reward for every action, such that $q(a)$ denotes the expected reward when the agent takes action $a \in \mathcal{A}$:

$$q(a) \triangleq \mathbb{E}[R_t | A_t = a]$$

Note that this is a simplified version of the state-action value function, $q(s, a)$, mentioned in the Introduction, which is natural since we have only one single state.

In our RL setting, the reward distributions of the arms are unknown to the agent. Hence, the agent has to estimate the action value from interaction with the machines. One natural way to estimate $q_t(a)$ is by averaging (*sample average method*) the rewards received when taking action a, i.e.:

$$q_t(a) = \frac{\text{sum of rewards when } a \text{ taken prior to } t}{\text{number of times } a \text{ taken prior to } t} \tag{2.1}$$

When the agent has estimates of the action values for every action, then it can choose the action with highest estimated value. We use the term *greedy action* to denote the action that provides highest estimated value, which is formally expressed as:

$$a_t^{\text{greedy}} = \arg\max_a q_t(a) \tag{2.2}$$

Note that when the agent takes a greedy action, we say that it is *exploiting* its current knowledge of the values of the actions. If instead the agent selects one of the non-greedy actions, then we say it is *exploring* the environment. *Exploitation* is fine for maximizing the expected reward in the short term; but *exploration* may improve the estimate and lead to better decisions in the future, which will produce greater reward in the long run. This conflict between exploitation and exploration is inherent to many sequential learning problems and the simplified multiarmed bandit problem is an excellent laboratory for studying exploration-exploitation tradeoffs.

One of the simplest tradeoff is the ϵ-greedy policy, which consists in behaving greedily with high probability, but also taking random exploratory actions with a small probability$\epsilon \ll 1$. The method is sometimes extended by proposing a schedule for the exploratory probability term, such that $\epsilon_t < \epsilon_{t-1}$. One typical ϵ-greedy policy is as follows:

$$a_t = \begin{cases} \arg\max_{a \in \mathcal{A}} q(a) & \text{with probability } 1-\epsilon \\ \mathbb{P}\left(A_t \sim \mathcal{U}(\mathcal{A})\right) = \epsilon & \text{with probability } \epsilon \end{cases} \tag{2.3}$$

Let us consider a very simple example for 10-armed bandit problem as shown in figure (2.1).

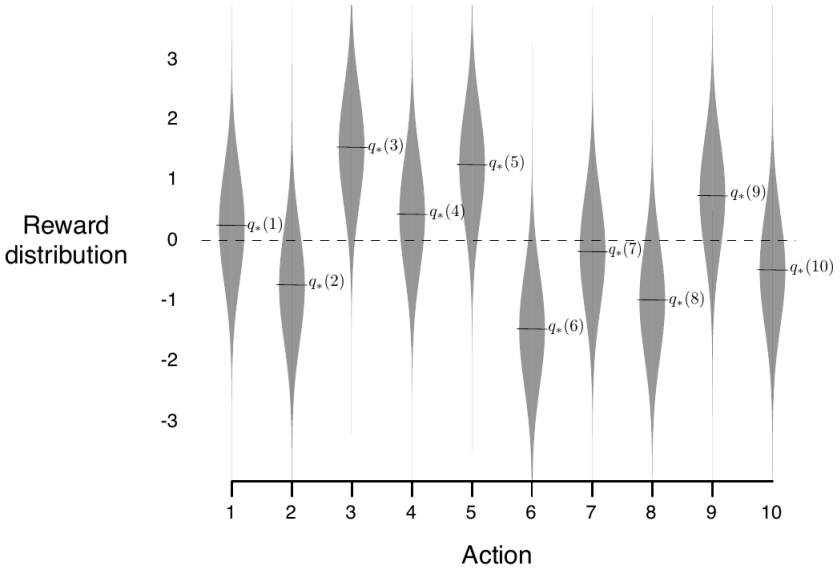


Figure 2.1: An exemplary bandit problem where all arms have Normally distributed reward of different mean but equal variance.

with optimal action value function $q^* = [\ 0.2,\ -0.8,\ 1.5,\ 0.4,\ 1.1,\ -1.5,\ -0.2,\ -1,\ 1,$

It is very interesting to represent the average performance of ε-greedy methods for the first 1000 iterations, as presented in Figure 2.2. It can be noticed that more exploration (i.e., higher ε) implies better performance in terms of the average reward and the percentage of selecting in fact the optimal action.

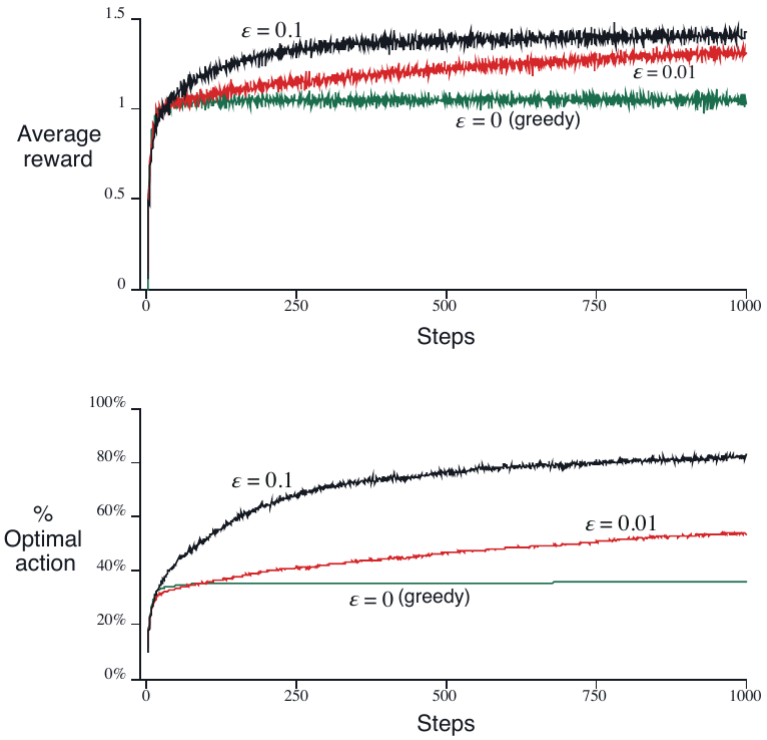


Figure 2.2: Performance of the 10-armed bandit problem in terms of the ε-greedy effect. Results have been averaged over 2000 independent episodes.

Exercise 2.1. Write a Matlab code to reproduce results shown in Figure 2.2 using the ϵ-greedy policy 2.3 and the reward distributions displayed in Figure 2.1.

Chapter 3

Markov Decision Processes

3.1 Markov processes (MP)

A Markov process (MP) is the simplest model for a non-white signal because the probability of any sample depends only upon the value of the preceding sample. It is defined as a tuple $\langle \mathcal{S}, \mathcal{P} \rangle$ where $\mathcal{S}$ is the set of states and $\mathcal{P}$ is the state-transition probability kernel. For simplicity, we assume that $\mathcal{S}$ is finite (which implies that it is countable) and non-empty, the state-transitions occurs at discrete time (extensions to infinite state sets and continuous time are also possible but out of the scope of this course) and the state-transition probability kernel is stationary. The state-transition probability kernel assigns to each state $s \in \mathcal{S}$ a probability measure over $\mathcal{S}$. Let S_t denote the state at time t, where we use upper case letters to denote random variables and lower case letters to denote realizations of the random variable. Since we are assuming a finite state set, the number of possible transitions from any state $s \in \mathcal{S}$ to any other state $s' \in \mathcal{S}$ is $|\mathcal{S}|^2$, where $|\cdot|$ denotes the cardinality of a set. Hence, we can conveniently express the transition probability kernel in matrix form. In particular, we say that $\mathcal{P}$ is a $|\mathcal{S}| \times |\mathcal{S}|$ matrix with elements

$$\mathcal{P}_{ss'} \triangleq \mathbb{P}\left[S_{t+1} = s' | S_t = s\right], \quad s, s' \in \mathcal{S} \tag{3.1}$$

where $\mathcal{P}_{ss'}$ denotes the element at row s and column s'. Note that $\mathcal{P}$ is a stochastic matrix since its rows are probability vectors, i.e.,:

$$\mathcal{P}_{ss'} \geq 0, \quad \forall s, s' \in \mathcal{S} \tag{3.2}$$

$$\sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} = 1 \tag{3.3}$$

We say that the process is Markov because the state transition probability depends on the current state only, not on the history of states before time t.

In other words, the transition is influenced by the state where we currently are, not how we got here, i.e.:

$$\begin{aligned}\mathbb{P}(S_{t+1} = s_{t+1} \,|S_t = s_t, S_{t-1} = s_{t-1}, S_{t-2} = s_{t-2}, \ldots, S_0 = s_0) \\ = \mathbb{P}\left[S_{t+1} = s_{t+1} | S_t = s_t\right]\end{aligned} \tag{3.4}$$

Let us introduce some definitions and properties for this kind of stochastic processes, also known as *Markov chains*.

Definition 3.1. Let $\mathcal{P}_{ss'}^{(n)} \triangleq \mathbb{P}(S_{t+n} = s' | S_t = s)$ denote the probability of reaching state $s' \in \mathcal{S}$ in n steps when we are at state s.

1. State s' is said to be reachable from state s, if there exists $n \geq 1$ so that $\mathcal{P}_{ss'}^{(n)} > 0$. A Markov chain is said to be irreducible if a state s' is reachable from any other state s, for all $s' \in \mathcal{S}$.

2. A Markov chain is said to be time homogeneous if the transition matrix $\mathcal{P}$ remains the same after each step.

3. State s' is said to be aperiodic if there exists n such that for all $n' > n$ we have that$\mathcal{P}_{ss'}^{(n)} > 0$. A Markov chain is said to be aperiodic if all its states are aperiodic.

Definition 3.2. Let $d(t) = (d_s(t))_{s \in \mathcal{S}}$ denote the vector of state-visitation probabilities at time t, with components $d_s(t) \triangleq \mathbb{P}(S_t = s)$.

1. The vector of state-visitation probabilities evolves according to the equation:
$$d(t)^T = d(t-1)^T \mathcal{P} \tag{3.5}$$

2. This type of vector is known as *probability vector*.
$$d_s(t) \geq 0 \quad \text{for all } s \in \mathcal{S} \tag{3.6}$$
$$\sum_{s \in \mathcal{S}} d_s(t) = 1 \tag{3.7}$$

We are interested in the asymptotic behavior of the Markov chain. In particular, later in the course, we will see that in order to learn from interaction, the environment should behave consistently along time. This means that the

state-transition probabilities should remain constant (this has been implicitly assumed when we wrote $\mathcal{P}$ independent on t, and it is known as time homogeneity, as described below) and that there should exist stationary a state-visitation distribution for the Markov chain (i.e., the state-visitation distribution tends to a limiting distribution), which should also be independent on the initial distribution. Learning in non-stationary or even adversarial environments is out of the scope of this course.

Let $d(t) = (d_s(t))_{s\in\mathcal{S}}$ denote the vector of state-visitation probabilities at time t, with components $d_s(t) \triangleq \mathbb{P}(S_t = s)$. The following propositions shows how to compute this vector recursively.

Proposition 3.1. *The vector of state-visitation probabilities evolves according to the equation:*

$$d(t)^\top = d(t-1)^\top \mathcal{P} \tag{3.8}$$

Proposition 3.2. *For a time-homogeneous Markov chain we have:*

$$\begin{aligned} d(t)^T &= d(t-1)^T\mathcal{P} \\ &= d(t-2)^T\mathcal{P}\mathcal{P} = d(t-2)^T\mathcal{P}^2 \\ &\vdots \\ &= d(0)^T\mathcal{P}^t \end{aligned} \tag{3.9}$$

Now, we introduce formally the concept of stationary distribution.

Definition 3.3. A distribution d is said to be a stationary distribution for the Markov chain if

$$d^T = d^T\mathcal{P} \tag{3.10}$$

We remark that such stationary distribution does not necessarily exist, nor is it necessarily unique. In the case that it does exist and it is unique, then it can always be interpreted as the state-visitation distribution, i.e., the average proportion of time spent by the chain in each state.

Definition 3.4. A distribution d is said to be the limiting distribution of the Markov chain if for every initial distribution we have

$$d = \lim_{t\to\infty} d(t) \tag{3.11}$$

We remark that a limiting distribution does not necessarily exists, but if it exists, then it is unique. Moreover, if d is a limiting distribution, then it

is clear that it is also the stationary distribution of the Markov chain. Note that $\lim_{t\to\infty} d(t) = \lim_{t\to\infty} d(t-1)$. Hence, we have:

$$d = \lim_{t\to\infty} d(t) = \lim_{t\to\infty} d(t-1)\mathcal{P} = d\mathcal{P} \tag{3.12}$$

The following theorem states conditions for the existence of d.

Theorem 3.1. *A time-homogeneous, irreducible, and aperiodic Markov chain has a limiting state distribution if and only if the transition matrix has a* single *real eigenvalue equal to* 1 *and no complex eigenvalues with magnitudes equal to* 1.

From 3.12 and Theorem 3.1, we conclude that the limiting distribution vector d is the eigenvector corresponding to an eigenvalue of 1 of $\mathcal{P}$.

Exercise 3.1. Write the state transition probability matrix for the Class state diagram shown in Figure 3.1. Calculate analytically the stationary state visitation probability and write a Matlab code starting from an arbitrary state that corroborates this result.

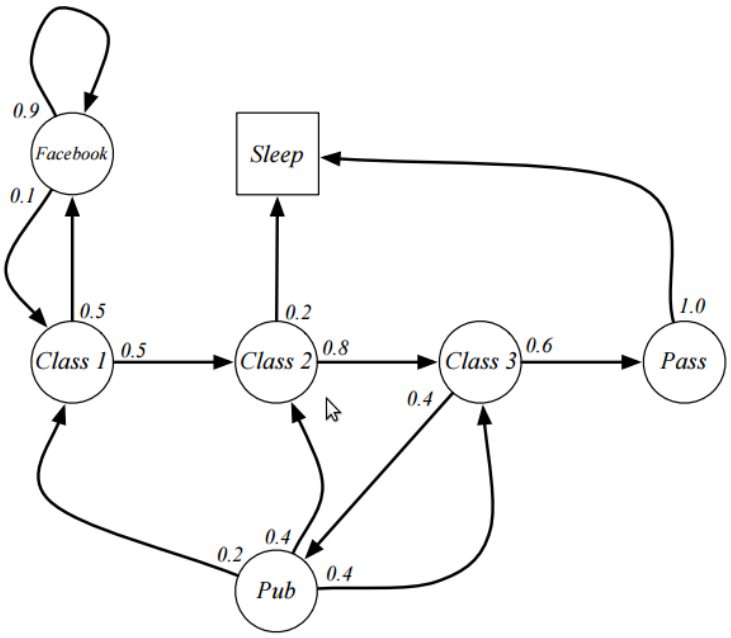


Figure 3.1: Class MP state-diagram

Solution:

$$\mathcal{P} = \begin{array}{c} \\ C1 \\ C2 \\ C3 \\ Pass \\ Pub \\ FB \\ Sleep \end{array} \begin{array}{c} \begin{array}{ccccccc} C1 & C2 & C3 & Pass & Pub & FB & Sleep \end{array} \\ \begin{bmatrix} & 0.5 & & & & 0.5 & \\ & & 0.8 & & & & 0.2 \\ & & & 0.6 & 0.4 & & \\ & & & & & & 1.0 \\ 0.2 & 0.4 & 0.4 & & & & \\ 0.1 & & & & & 0.9 & \\ & & & & & & 1 \end{bmatrix} \end{array}$$

Figure 3.2: Class MP transition matrix

3.2 Markov Decision Process (MDP)

3.2.1 Definition

A Markov decision process (MDP) can be seen as an extension of an MP in which the state transition probabilities depend on an extra random variable, named the action, and where there is associated to every transition another random variable named the reward. More formally, an MDP is defined as a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$ where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions, $\mathcal{P}$ is the transition probability kernel and $\mathcal{R}$ is the reward function. Again, for simplicity, in this course we assume that $\mathcal{S}$ and $\mathcal{A}$ are finite and non-empty.

Different from MPs, the state-transition probability kernel now depends on the action taken by the agent and assigns to each state-action pair $(s, a) \in \mathcal{S} \times \mathcal{A}$ a probability measure over $\mathcal{S}$, which is the probability of transiting to another state $s' \in \mathcal{S}$. Let S_t and A_t denote the state and action at time t, respectively. Since we are assuming finite state and action sets, the number of possible transitions from any state action pair $(s, a) \in \mathcal{S} \times \mathcal{A}$ to any other state $s' \in \mathcal{S}$ is $|\mathcal{S}|^2|\mathcal{A}|$. Hence, similar to MP, we can also express the transition probability kernel in matrix form. In particular, we can express $\mathcal{P}$ for an MDP as an $|\mathcal{S}||\mathcal{A}| \times |\mathcal{S}|$ matrix or as a $|\mathcal{S}| \times |\mathcal{S}| \times |\mathcal{A}|$ tensor, in either case, the elements are given by:

$$\mathcal{P}^a_{ss'} \triangleq \mathbb{P}\left[S_{t+1} = s' | S_t = s, A_t = a\right] \tag{3.13}$$

For the matrix representation of $\mathcal{P}$, each of the rows denotes state-action pairs (s, a), and each column denotes the future state s':

$$\mathcal{P} \triangleq (\mathcal{P}^a_{ss'})_{(s,a)\in\mathcal{S}\times\mathcal{A}, s'\in\mathcal{S}}$$

Let R_{t+1} denote the random *reward* obtained at time $t + 1$. Then, we define the reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \Re$ as the expected value of the random reward for any transition from some state-action pair:

$$\mathcal{R}^a_s \triangleq \mathcal{R}(s, a) \triangleq \mathbb{E}\left[R_{t+1} | S_t = s, A_t = a\right] \tag{3.14}$$

By slightly abusing notation, it is convenient to define the reward vector as the vector of length $|\mathcal{S}||\mathcal{A}|$ that contains the reward function for every possible state-action pair:

$$\mathcal{R} \triangleq (\mathcal{R}_s^a)_{s\in\mathcal{S},a\in\mathcal{A}}$$

As we have already introduced in the earlier chapters, the agent takes actions by following a *policy*. Formally, we define a policy $\pi \in \Pi$, where Π is some set of probability distributions over $\mathcal{A}$, as a distribution over actions given states:

$$\pi(a|s) \triangleq \mathbb{P}(A_t = a|S_t = s) \tag{3.15}$$

Note that a policy fully defines the behavior of an agent. We assume that the policies are defined stationary (i.e., $A_t \sim \pi(\cdot|S_t)$, $\forall t > 0$). Thus, if the agent behavior changes with time (e.g., because it learns how to control the environment), we consider that it is the agent who changes its policy over time, rather than that the policy changes itself.

Definition 3.5. A deterministic policy is a particular case of the mapping $\pi : \mathcal{S} \to \mathcal{A}$, such that by writing $\pi(s) = a$, we mean that $\pi(a|s) = 1$ (i.e., $\mathbb{P}(A_t = a|S_t = s) = 1$).

We also define $\mathcal{P}_{ss'}^{\pi}$ and $\mathcal{R}_s^{\pi}$ as the average transition probability and average reward function, respectively, when the actions are taken from π:

$$\mathcal{P}_{ss'}^{\pi} = \sum_{a\in\mathcal{A}} \pi(a|s)\,\mathcal{P}_{ss'}^{a} \tag{3.16}$$

$$\mathcal{R}_s^{\pi} = \sum_{a\in\mathcal{A}} \pi(a|s)\,\mathcal{R}_s^{a} \tag{3.17}$$

It is convenient to organize all elements (3.16) for all state transition pairs $(s, s') \in \mathcal{S} \times \mathcal{S}$ into a matrix $\mathcal{P}^{\pi}$ of size $|\mathcal{S}| \times |\mathcal{S}|$, and to stack the rewards (3.17) for all $s \in \mathcal{S}$ into a reward vector $\mathcal{R}^{\pi}$ of length $|\mathcal{S}|$:

$$\mathcal{P}^{\pi} \triangleq (\mathcal{P}_{ss'}^{\pi})_{s,s'\in\mathcal{S}} \tag{3.18}$$

$$\mathcal{R}^{\pi} \triangleq (\mathcal{R}_s^{\pi})_{s\in\mathcal{S}} \tag{3.19}$$

In the rest of the course, whenever we have to learn to predict or control the MDP from samples, without knowing $\mathcal{P}$ or $\mathcal{R}$, we will assume the following condition.

Assumption 1. *For any MDP and any stationary policy, the induced Markov chain associated to the state-action-state-reward transitions satisfy Theorem 3.1 so that it has a limiting distribution that is also the unique stationary state-visitation distribution.*

The intuition of Assumption 1 is that by having a stationary state-visitation distribution and being able to visit every state, the agent can build consistent estimates from samples.

Nevertheless, there are many problems that are naturally modelled as finishing at some *terminal* state, where it remains with probability one (for this reason terminal states are also known as *absorbing* states). These MDPs with terminal states are usually known as *episodic*, and the trajectory from the initial state to a terminal state is known as *episode*. It is clear that Assumption 1 is not satisfied for this kind of episodic MDP. Since, in order to learn from samples with the methods that will be proposed in this course, we need the MDP to satisfy Assumption 1, we usually do the following trick: every time the trajectory reaches a terminal state, we reset the episode (both the sample return equals zero and the trajectory restarts at the initial state). This way, we obtain infinite episodes (of finite length each). Thus, Assumption 1 must hold for the MDP in which the transition to the terminal state is transformed into a restart to the initial state. The *Cliff Walking problem* to be analysed later is a good example. Just to mention that there exists other RL methods that do not require Assumption 1, but they will not be covered in this course.

Example 3.1. Let us consider the so named *Cliff Walking problem*, which is usually presented as an episodic task—with a start state and an absorbing goal state—, and the agent can choose one of the four usual actions for moving up, down, right or left, from one state to the next one. Reward is -1 in all transitions with the exception of those into the region marked as the *cliff* and the goal state. Stepping into this region incurs a reward of -100 and send the agent instantly back to the start. And the reward when transiting to the goal state is zero by convention.

3.2.2 Value functions and Bellman equations

The goal in an MDP is to find the policy that maximizes the expected long term return. We formally define the *return* obtained from time t as the discounted cumulative reward:

$$G_t \triangleq R_{t+1} + \gamma R_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3.20}$$

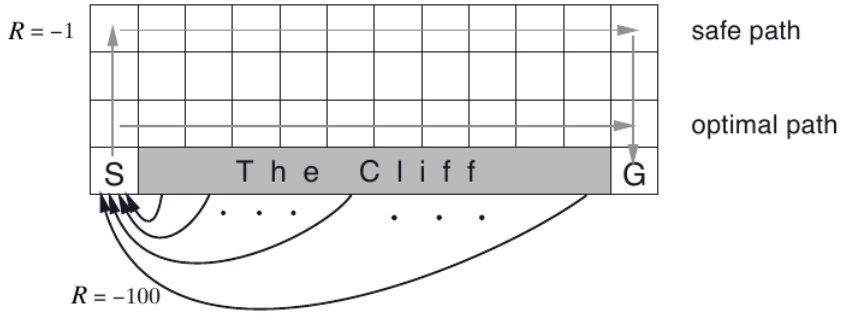


Figure 3.3: The Cliff-walking problem

where $0 < \gamma < 1$ is the so named *discount* factor. There is a common alternative formulation based on the average reward, but in this course we exclusively focus on the discounted return given by (3.20).

In (3.20), the value of the reward after $k+1$ steps is down-weighted by γ^k. This is usually convenient to avoid divergence (since the sum has infinite terms) but it also has some financial meaning in the sense that immediate rewards worth more than delayed rewards. When $\gamma \approx 0$, we say that the return is "myopic"; while when $\gamma \approx 1$, we say that the return is "far-sighted". As a rule of thumb, the theoretical results provided in these notes are valid for the open interval $\gamma \in (0, 1)$, but we must keep in mind that the close interval $\gamma \in [0, 1]$ might be acceptable for some specific problems (e.g., episodic MDPs in which any trajectory gets into a terminal state in a finite number of steps with probability one).

We remark that the return G_t is a random variable that depends on the specific trajectory followed by the agent. We introduce the *state value* function $v^\pi : \mathcal{S} \to \mathbb{R}$ as the expected return over all possible trajectories when the agent starts at some state, and then follows policy π:

$$v^\pi(s) \triangleq \mathbb{E}_{\pi,\mathcal{P}}\left[G_t | S_0 = s, A_{t+k} \sim \pi\right], \quad k = 0, \ldots, \infty \tag{3.21}$$

where $\mathbb{E}_{\pi,\mathcal{P}}[\cdot]$ denotes the expected value over the policy distribution and over the state-transition distribution. We also introduce the *state-action* value function $q^\pi : \mathcal{S} \times \mathcal{A} \to \Re$ as the the expected return over all possible trajectories when the agent starts at some state, takes some arbitrary action, and then follows policy π:

$$q^\pi(s, a) \triangleq \mathbb{E}_{\pi,\mathcal{P}}\left[G_t | S_t = s, A_t = a, A_{t+k} \sim \pi\right], \quad k = 1, \ldots, \infty \tag{3.22}$$

Note that the difference between the right sides of (3.21) and (3.22) is that k starts at 0 in (3.21), while k starts at 1 in (3.22), since $A_t = a$ for the latter.

The value functions satisfy a recursive relation, known as Bellman equation. For the state-value function, we have for $k = 0, \ldots, \infty$:

$$\begin{aligned} v^{\pi}(s) &= \mathbb{E}_{\pi,\mathcal{P}}\left[G_t | S_t = s, A_t \sim \pi\right] \\ &= \mathbb{E}_{\pi,\mathcal{P}}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots | S_t = s, A_{t+k} \sim \pi\right] \\ &= \mathbb{E}_{\pi,\mathcal{P}}\left[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \ldots) | S_t = s, A_{t+k} \sim \pi\right] \\ &= \mathbb{E}_{\pi,\mathcal{P}}\left[R_{t+1} + \gamma G_{t+1} | S_t = s, A_{t+k} \sim \pi\right] \\ &= \mathbb{E}_{\pi,\mathcal{P}}\left[R_{t+1} + \gamma v^{\pi}(S_{t+1}) | S_t = s, A_{t+k} \sim \pi\right] \end{aligned} \tag{3.23}$$

Similarly, for the state-action-value function, we have for $k = 1, \ldots, \infty$:

$$\begin{aligned} q^{\pi}(s,a) &= \mathbb{E}_{\pi,\mathcal{P}}\left[G_t | S_t = s, A_t = a, A_{t+k} \sim \pi\right] \\ &= \mathbb{E}_{\pi,\mathcal{P}}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots | S_t = s, A_t = a, A_{t+k} \sim \pi\right] \\ &= \mathbb{E}_{\pi,\mathcal{P}}\left[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a, A_{t+k} \sim \pi\right] \\ &= \mathbb{E}_{\pi,\mathcal{P}}\left[R_{t+1} + \gamma q^{\pi}(S_{t+1}, A_{t+1}) | S_t = s, A_t = a, A_{t+k} \sim \pi\right] \end{aligned} \tag{3.24}$$

In other words, we can decompose the state and state-action value functions into the immediate reward plus the discounted value of successor state or state-action pair, respectively.

The state and state-action value functions are related as follows:

$$v^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(a|s)\, q^{\pi}(s,a) \tag{3.25}$$

$$q^{\pi}(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v^{\pi}(s') \tag{3.26}$$

where (3.25) is obtained from (3.22) by considering that $A_t \sim \pi$ (i.e., the current action also follows π); and (3.26) is obtained from (3.24) by using (3.14) and the following relation for $k = 1, \ldots, \infty$:

$$\mathbb{E}_{\pi,\mathcal{P}}\left[G_{t+1} | S_t = s, A_t = a, A_{t+k} \sim \pi\right] = \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v^{\pi}(s') \tag{3.27}$$

This is visually expressed with the diagram displayed in Figure 3.4.

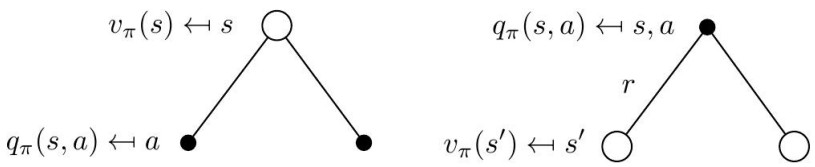


Figure 3.4: Relationships between $v_\pi(s)$ and $q_\pi(s,a)$

In addition, from (3.25) and (3.26), we have:

$$v^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v^{\pi}(s') \right) \tag{3.28}$$

$$q^{\pi}(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s') \, q^{\pi}(s',a') \tag{3.29}$$

which is visually expressed in Figure 3.5.

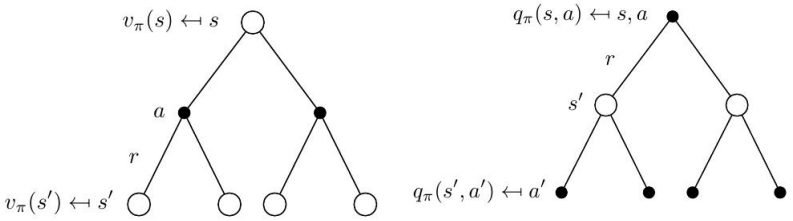


Figure 3.5: Recursive equations for $v_\pi(s)$ and $q_\pi(s,a)$

Interestingly, note that the value function for every state (or state-action pair) form a system of linear equations. Thus, given $\mathcal{R}$ and $\mathcal{P}$ we can obtain v^{π} and q^{π} for every state and every state-action pair, respectively, by solving a system of linear equations. This is more clearly seen by rewriting the equations in vector form. By slightly abusing notation, we can introduce the vectors with the state and state-action value functions for every state and every state-action pairs, respectively:

$$v^{\pi} \triangleq (v^{\pi}(s))_{s \in \mathcal{S}} \tag{3.30}$$

$$q^{\pi} \triangleq (q^{\pi}(s,a))_{s \in \mathcal{S}, a \in \mathcal{A}} \tag{3.31}$$

In addition, we can write the policy $\pi \in \Pi$ as a matrix $\underline{\Pi}$ of size $|\mathcal{S}| \times |\mathcal{S}||\mathcal{A}|$ (note that we use capitolΠ to denote the policy space and italic capitolΠ to denote the policy in matrix form):

$$\Pi = \begin{bmatrix} \mathbb{P}(a_1 \mid s_1) & \dots & \mathbb{P}(a_{|\mathcal{A}|} \mid s_1) \\ \vdots & \vdots & \vdots \\ \mathbb{P}(a_1 \mid s_{|\mathcal{A}|}) & \dots & \mathbb{P}(a_{|\mathcal{A}|} \mid s_{|\mathcal{A}|}) \end{bmatrix} \in \Re^{|\mathcal{S}| \times |\mathcal{S}||\mathcal{A}|} \tag{3.32}$$

By using (3.18) and (3.19), we can express (3.23)–(3.24) for all states and state-action pairs in vector form:

$$v^{\pi} = \mathcal{R}^{\pi} + \gamma \mathcal{P}^{\pi} v^{\pi} \tag{3.33}$$

$$q^{\pi} = \mathcal{R} + \gamma \mathcal{P} \Pi \, q^{\pi} \tag{3.34}$$

where we emphasize that (3.33) includes the average reward vector $\mathcal{R}^\pi$ and transition matrix $\mathcal{P}^\pi$ induced by π, while (3.34) includes the policy independent $\mathcal{R}$ and $\mathcal{P}$. Eqs. (3.33)–(3.34) clearly show that the Bellman equations form systems of linear equations. Moreover, from this compact formulation, it is straightforward to solve for the value functions in closed form:

$$v^\pi = (I - \gamma \mathcal{P}^\pi)^{-1} \mathcal{R}^\pi \tag{3.35}$$

$$q^\pi = (I - \gamma \mathcal{P} \Pi)^{-1} \mathcal{R} \tag{3.36}$$

The following result states the existence and uniqueness of the solution of these systems of equations.

Proposition 3.3. *Under Assumption 1, there exist unique solutions v^π and q^π to (3.35) and (3.36), respectively, and they are the value functions that satisfy (3.28) and (3.29).*

Interestingly, we can also write (3.25)–(3.26) in vector form:

$$v^\pi = \Pi q^\pi \tag{3.37}$$

$$q^\pi = \mathcal{R} + \gamma \mathcal{P} v^\pi \tag{3.38}$$

Also, we can deduce previous Eqs. (3.18)–(3.19) from the policy matrix:

$$\mathcal{P}^\pi = \Pi \mathcal{P}$$

$$\mathcal{R}^\pi = \Pi \mathcal{R}$$

Now, let us see how to build and solve the systems of linear Bellman equations (3.33)–(3.34) for a simple example.

Exercise 3.2. Consider an MDP composed of two states, $\mathcal{S} \triangleq \{x, y\}$, and two actions per state, $\mathcal{A} \triangleq \{u, m\}$. The possible actions at both states are $u =$ unmove, i.e., to stay in the same state, or $m =$ move to the other state. The state-transition probabilities are

$$\begin{aligned} \mathcal{P}^u_{xx} &= 0.8, \quad \mathcal{P}^u_{xy} = 0.2 \\ \mathcal{P}^m_{xx} &= 0.2, \quad \mathcal{P}^m_{xy} = 0.8 \\ \mathcal{P}^u_{yx} &= 0.3, \quad \mathcal{P}^u_{yy} = 0.7 \\ \mathcal{P}^m_{yx} &= 0.9, \quad \mathcal{P}^m_{yy} = 0.1 \end{aligned}$$

Figure (3.6) shows the state transition diagram for all possible state transitions.

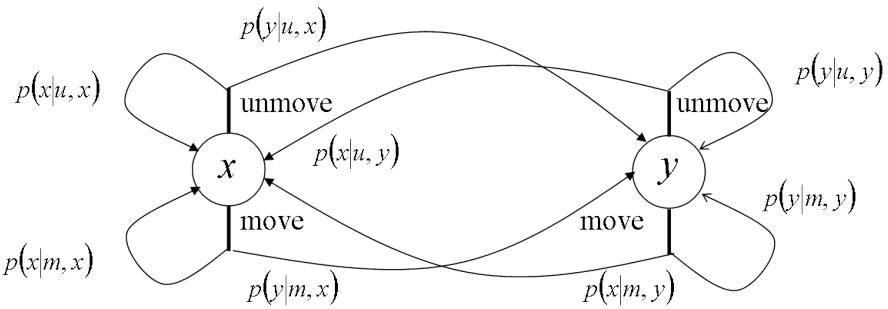


Figure 3.6: Transition graph. Simple example with two states

Let the discount factor be $\gamma = 0.9$ and let the rewards be defined as follows:

$$\begin{aligned} \mathcal{R}_x^u &= -1, \quad \mathcal{R}_x^m = 0.6 \\ \mathcal{R}_y^u &= 0.5, \quad \mathcal{R}_y^m = -0.9 \end{aligned}$$

The goal of this exercise is to calculate the state-value and state-action value functions in vector form. The first part of the exercise consists in solving the system of Bellman equations analytically. The second part of the exercise consists in programming these formulas in Matlab, so that they can be reused for more complex exercises.

a) Write the value functions in vector form: $v^\pi \triangleq (v^\pi(s))_{s\in\mathcal{S}}$ and $q^\pi \triangleq (q^\pi(s,a))_{s\in\mathcal{S},a\in\mathcal{A}}$, which for this example become:

$$v^\pi = \begin{bmatrix} v^\pi(x) \\ v^\pi(y) \end{bmatrix} \in \Re^{|\mathcal{S}|}, \quad q^\pi = \begin{bmatrix} q^\pi(x,u) \\ q^\pi(x,m) \\ q^\pi(y,u) \\ q^\pi(y,m) \end{bmatrix} \in \Re^{|\mathcal{S}||\mathcal{A}|} \tag{3.39}$$

b) Write the expected reward vector, the transition probability matrix and the policy in matrix form:

$$\mathcal{R} = \begin{bmatrix} \mathcal{R}_x^u \\ \mathcal{R}_x^m \\ \mathcal{R}_y^u \\ \mathcal{R}_y^m \end{bmatrix} \in \Re^{|\mathcal{S}||\mathcal{A}|}, \quad \mathcal{P} = \begin{bmatrix} \mathcal{P}_{xx}^u & \mathcal{P}_{xy}^u \\ \mathcal{P}_{xx}^m & \mathcal{P}_{xy}^m \\ \mathcal{P}_{yx}^u & \mathcal{P}_{yy}^u \\ \mathcal{P}_{yx}^u & \mathcal{P}_{yx}^m \end{bmatrix} \in \Re^{|\mathcal{S}||\mathcal{A}|\times|\mathcal{S}|} \tag{3.40}$$

$$\Pi = \begin{bmatrix} \pi(u \mid x) & \pi(m \mid x) & 0 & 0 \\ 0 & 0 & \pi(u \mid y) & \pi(m \mid y) \end{bmatrix} \in \Re^{|\mathcal{S}|\times|\mathcal{S}||\mathcal{A}|} \tag{3.41}$$

c) Write the transition probability matrix and the expected reward vector for policy all possible deterministic policies. Fist, we deduce the formulas for any policy π from (3.16)–(3.17):

$$\mathcal{P}_{ss'}^{\pi} = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{P}_{ss'}^{a}, \quad \rightarrow \quad \mathcal{P}^{\pi} = \Pi \mathcal{P} \tag{3.42}$$

$$\mathcal{R}_{s}^{\pi} = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{R}_{s}^{a}, \quad \rightarrow \quad \mathcal{R}^{\pi} = \Pi \mathcal{R} \tag{3.43}$$

The motivation for this task is that a well known result from optimal control theory states that, under mild conditions, there exists an optimal policy for any MDP that is deterministic (see, Lemma 3.1 below). Hence, if we were interested in finding the optimal policy, we could evaluate the value function for every possible deterministic and take the one that maximizes the value (this will be explained below). The number of possible deterministic polices is given by all possible combinations, of actions and state, such that $|\Pi| = |\mathcal{S}|^{|\mathcal{A}|}$. For this particular example we have $|\Pi| = 2^2 = 4$. Let us enumerate the four possible deterministic policies denoted $\pi_1, \ldots, \pi_4$:

$$\begin{aligned}
&\pi_1(u|x) = 1,\ \pi_1(m|x) = 0, &&\pi_1(u|y) = 1,\ \pi_1(m|y) = 0\\
&\pi_2(u|x) = 0,\ \pi_2(m|x) = 1, &&\pi_2(u|y) = 1,\ \pi_2(m|y) = 0\\
&\pi_3(u|x) = 1,\ \pi_3(m|x) = 0, &&\pi_3(u|y) = 0,\ \pi_3(m|y) = 1\\
&\pi_4(u|x) = 0,\ \pi_4(m|x) = 1, &&\pi_4(u|y) = 0,\ \pi_4(m|y) = 1
\end{aligned}$$

From (3.42)–(3.43), we have the following transition matrices and reward vectors for each of these policies:

$$\mathcal{P}^{\pi_1} = \begin{bmatrix} 0.8 & 0.2 \\ 0.3 & 0.7 \end{bmatrix}, \quad \mathcal{P}^{\pi_2} = \begin{bmatrix} 0.2 & 0.8 \\ 0.3 & 0.7 \end{bmatrix}$$

$$\mathcal{P}^{\pi_3} = \begin{bmatrix} 0.8 & 0.2 \\ 0.9 & 0.1 \end{bmatrix}, \quad \mathcal{P}^{\pi_4} = \begin{bmatrix} 0.2 & 0.8 \\ 0.9 & 0.1 \end{bmatrix}$$

$$\mathcal{R}^{\pi_1} = \begin{bmatrix} -1 \\ 0.5 \end{bmatrix}, \quad \mathcal{R}^{\pi_2} = \begin{bmatrix} 0.6 \\ 0.5 \end{bmatrix}, \quad \mathcal{R}^{\pi_3} = \begin{bmatrix} -1 \\ -0.9 \end{bmatrix}, \quad \mathcal{R}^{\pi_4} = \begin{bmatrix} 0.6 \\ -0.9 \end{bmatrix}$$

d) Obtain the state and state-action value functions for each of the polices. Recall:

$$v^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q(s,a) \rightarrow v^{\pi} = \Pi q^{\pi} \tag{3.44}$$

$$q^{\pi}(s,a) = \mathcal{R}_{s}^{a} + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^{a} v^{\pi}(s') \rightarrow q^{\pi} = \mathcal{R} + \gamma \mathcal{P} v^{\pi} \tag{3.45}$$

we can compute the value function for each of these polices as follows:

$$v^{\pi_1} = \begin{bmatrix} -5.09 \\ -2.36 \end{bmatrix}, \quad v^{\pi_2} = \begin{bmatrix} 5.34 \\ 5.25 \end{bmatrix}, \quad v^{\pi_3} = \begin{bmatrix} -9.84 \\ -9.74 \end{bmatrix}, \quad v^{\pi_4} = \begin{bmatrix} -0.63 \\ -1.54 \end{bmatrix}$$

and identically for the state-action value functions:

$$q^{\pi_1} = \begin{bmatrix} -5.09 \\ -2.02 \\ -2.36 \\ -5.24 \end{bmatrix}, \quad q^{\pi_2} = \begin{bmatrix} 3.79 \\ 5.34 \\ 5.25 \\ 3.90 \end{bmatrix}, \quad q^{\pi_3} = \begin{bmatrix} -9.83 \\ -8.19 \\ -8.29 \\ -9.74 \end{bmatrix}, \quad q^{\pi_4} = \begin{bmatrix} -1.73 \\ -0.63 \\ -0.64 \\ -1.55 \end{bmatrix}$$

e) Finally, derive the value functions in the fixed point vector form:

$$v^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v^{\pi}(s') \right) \rightarrow v^{\pi} = \Pi \mathcal{R} + \gamma \Pi \mathcal{P} v^{\pi}$$
$$q^{\pi}(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s') q^{\pi}(s',a') \rightarrow q^{\pi} = \mathcal{R} + \gamma \mathcal{P} \Pi q^{\pi}$$

And implement the solution in Matlab:

$$v^{\pi} = \Pi \mathcal{R} + \gamma \Pi \mathcal{P} v^{\pi} = \mathcal{R}^{\pi} + \gamma \mathcal{P}^{\pi} v^{\pi} \rightarrow v^{\pi} = (I - \gamma \mathcal{P}^{\pi})^{-1} \mathcal{R}^{\pi}$$
$$q^{\pi} = \mathcal{R} + \gamma \mathcal{P} \Pi q^{\pi} \rightarrow q^{\pi} = (I - \gamma \mathcal{P} \Pi)^{-1} \mathcal{R}$$

Exercise 3.3. Consider the state diagram shown in Figure (3.7) describing a *Random Walk* problem. All episodes start in the center state, C, and proceed either left or right by one state on each step, with equal probability. Episodes terminate either on the extreme left or the extreme right. When an episode terminates on the right, a reward of +1 occurs; all other rewards are zero. For example, a typical episode might consist of the following state-and-reward sequence: C, 0, B, 0, C, 0, D, 0, E, 1. Consider $\gamma = 0.9$. The goal is to compute the state value function for every state applying the fixed point solution to the Bellman equation applied to the state value function.

$$\mathcal{R} = \left[R_{T_1}^l,\ R_{T_1}^r,\ R_A^l,\ R_A^r,\ R_B^l,\ R_B^r,\ R_C^l,\ R_C^r,\ R_D^l,\ R_D^r,\ R_E^l,\ R_E^r,\ R_{T_2}^l,\ R_{T_2}^r \right] \tag{3.46}$$

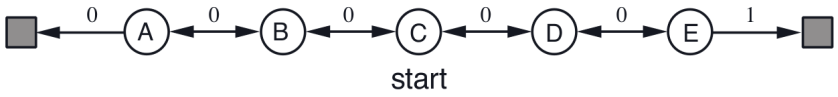


Figure 3.7: Example 3.3.

$$\mathcal{R} = \begin{bmatrix} 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 1, & 0, & 0 \end{bmatrix} \tag{3.47}$$

The transition matrix is given by:

$$\mathcal{P} = \begin{bmatrix} & T_1 & A & B & C & D & E & T_2 \\ T_1\, l & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ T_1\, r & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ A\, l & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ A\, r & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ B\, l & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ B\, r & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ C\, l & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ C\, r & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ D\, l & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ D\, r & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ E\, l & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ E\, r & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ T_2\, l & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ T_2\, r & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.48}$$

The policy matrix is given by:

$$\Pi = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \end{bmatrix} \tag{3.49}$$

Therefore, we have

$$v^{\pi} = \Pi\mathcal{R} + \gamma\Pi\mathcal{P}v^{\pi} = \begin{bmatrix} 0 \\ 0.07 \\ 0.15 \\ 0.26 \\ 0.43 \\ 0.69 \\ 0 \end{bmatrix}$$

So far, we have only considered the *policy evaluation* problem, i.e., we have seen how to compute the value functions for some given policy π. Now, we are going to consider the *control problem* that consists in the agent finding the policy that achieves the best possible performance. Since the performance of a policy is evaluated from its value function, we say that the the agent behaves optimally—it follows the so named optimal policy—when it achieves the highest possible value function—the optimal value function. The optimal policy and optimal value functions are defined as follows.

Definition 3.6. The optimal state-value function v^* is the maximum state-value function over all possible policies:

$$v^*(s) \triangleq \max_{\pi \in \Pi} v^{\pi}(s), \quad \forall s \in \mathcal{S} \tag{3.50}$$

where Π denotes the set of policies. The optimal state-action-value function $q^*(s,a)$ is the maximum state-action value function over all policies:

$$q^*(s,a) \triangleq \max_{\pi \in \Pi} q^{\pi}(s,a), \quad \forall (s,a) \in \mathcal{S} \times \mathcal{A} \tag{3.51}$$

Equivalently, we can express the optimal value functions in vector form:

$$v^* = \max_{\pi \in \Pi} \sum_{s \in \mathcal{S}} v^{\pi}(s) \tag{3.52}$$

$$q^* = \max_{\pi \in \Pi} \sum_{(s,a) \in \mathcal{S} \times \mathcal{A}} q^{\pi}(s,a) \tag{3.53}$$

Intuitively, we can define the optimal policy as one that yields the optimal value function. This idea is more formally stated in the following theorem.

Theorem 3.2. *Defining a partial ordering over policies*

$$\pi \geq \pi' \quad \textit{if} \quad v^{\pi}(s) \geq v^{\pi'}, \forall s \tag{3.54}$$

Then, for any MDP:

1. There exists an optimal policy π^* that is better than or equal to all other policies, i.e., $\exists \pi^* \in \Pi \ : \ \pi^* \geq \pi, \ \forall \pi \in \Pi$.

2. Every optimal policy achieves the optimal state and state-action value functions, i.e., $v^{\pi^*}(s) = v^*(s)$ and $q^{\pi^*}(s,a) = q^*(s,a)$.

It turns out that the existence of a deterministic optimal policy is guaranteed by the following lemma.

Lemma 3.1. *There is always a deterministic optimal policy for any MDP that satisfies Assumption 1.*

Exercise 3.4. Find the optimal policy for exercise 3.2.

a) Calculate $v^\pi = (I - \gamma \mathcal{P}^\pi)^{-1} \mathcal{R}^\pi$ for all deterministic policies and obtain the optimal policy from (3.52) $v^* = \max_{\pi \in \Pi} \sum_{s \in \mathcal{S}} v^\pi(s)$.

If we compute the value function for each of these polices:

$$v_{\pi_1} = \begin{bmatrix} -5.09 \\ -2.36 \end{bmatrix}, \quad v_{\pi_2} = \begin{bmatrix} 5.34 \\ 5.25 \end{bmatrix}, \quad v_{\pi_3} = \begin{bmatrix} -9.84 \\ -9.74 \end{bmatrix}, \quad v_{\pi_4} = \begin{bmatrix} -0.63 \\ -1.54 \end{bmatrix}$$

From these values, we conclude that the optimal policy is π_2, so that

$$\pi^* = \pi_2$$
$$v^* = v^{\pi_2}$$

b) Calculate the state-value function for 1000 random policies and verify that the value functions are below the optimal one found in a).

This approach of finding the optimal policy by computing the value function for every possible policy and then use definitions (3.50)–(3.53) is usually known as *policy search*. Since the number of possible deterministic policies grow exponentially with the size of the state-action sets, it is clear that this direct method may turn unfeasible when the state-action sets are large. Approximations that constraint the search to a restricted policy spaces —like parametric policies— work very well in practice though.

Another approach consists in finding the optimal value function first and then obtaining the optimal policy from the optimal value function. The theory supporting this alternative approach is given by the following Lemma.

Lemma 3.2. *An optimal policy can be found by maximizing the action a over the optimal value functionq*$^*(s,a)$*, i.e.,*

$$\pi^*(a \mid s) = \begin{cases} 1 & \text{if} \quad a \in \arg\max_{a \in \mathcal{A}} q^*(s,a) \\ 0 & \text{otherwise} \end{cases} \tag{3.55}$$

It may happen that more than one action maximize $q^*(s,a)$, in that case we have as many deterministic optimal policies as optimal actions. For instance, let us say that $\mathcal{A} = \{a_1, a_2, a_3, a_4\}$ and suppose there are two actions $a_1^*, a_2^* \in \mathcal{A}$, such that

$$\{a_1^*, a_2^*\} \in \arg\max_{a \in \mathcal{A}} q^*(s,a)$$

Hence, the following policies are optimal:

$$\pi_a^*(a_1^* \mid s) = 1, \pi^*(a_i \mid s) = 0, \ i \in \{2,3,4\}$$
$$\pi_b^*(a_2^* \mid s) = 1, \pi^*(a_i \mid s) = 0, \ i \in \{1,3,4\}$$

Moreover, we have a continuum of stochastic policies that are also optimal, since $\forall \alpha \in [0,1]$,

$$\pi_\alpha^*(a_1^* \mid s) = \alpha, \quad \pi_\alpha^*(a_2^* \mid s) = 1 - \alpha, \quad \pi^*(a_i \mid s) = 0, \ i \in \{3,4\}$$

In the following chapters, we will show different algorithms for finding v^* or q^* and then deriving optimal policy from them. Many of these methods are based on another form of the Bellman equations. But before getting into the details, we introduce the notion of *greediness*.

Definition 3.7. We say that policy π is greedy for some state $s \in \mathcal{S}$ with respect to some value function $v : \mathcal{S} \to \Re$ or $q : \mathcal{S} \times \mathcal{A} \to \Re$, if it maximizes such value function:

$$\begin{aligned} \pi(s) &= \arg\max_{a \in \mathcal{A}} q(s,a) \\ &= \arg\max_{a \in \mathcal{A}} \left[\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v(s') \right] \end{aligned} \tag{3.56}$$

It is easy to show that the optimal policy is greedy with respect to the optimal value function. This result is formally stated in the following lemma, which is illustrated in the example below.

Lemma 3.3. *The optimal policy π^* is greedy for all $s \in \mathcal{S}$ with respect to the optimal value function $v^* : \mathcal{S} \to \Re$ or $q^* : \mathcal{S} \times \mathcal{A} \to \Re$:*

$$\begin{aligned} \pi^*(s) &= \arg\max_{a \in \mathcal{A}} q^*(s,a) \\ &= \arg\max_{a \in \mathcal{A}} \left[\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v^*(s') \right] \end{aligned} \tag{3.57}$$

Example 3.2. In Exercise 3.2, the greedy action when the environment is at state x and we follow policy π_1 is given by

$$\begin{aligned} a^{\text{o}} &= \arg\max_{a \in \mathcal{A}} q^{\pi_1}(x,a) = \max_{a \in \mathcal{A}} \left[\mathcal{R}_x^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{xs'}^a q^{\pi_1}(s') \right] \\ &= \arg\max_{a \in \{u,m\}} \left[\underbrace{-1 + \gamma\left(0.8 v^{\pi_1}(x) + 0.2 v^{\pi_1}(y)\right)}_{a=u}, \underbrace{0.6 + \gamma\left(0.2 v^{\pi_1}(x) + 0.8 v^{\pi_1}(y)\right)}_{a=m} \right] \\ &= \arg\max_{a \in \{u,m\}} \left[\underbrace{-5.09}_{u}, \underbrace{-2.02}_{m} \right] \\ &= m \end{aligned} \tag{3.58}$$

The Bellman equations (3.25)–(3.29), defined above for general policies, can be particularized for deterministic optimal policies, yielding the following relations:

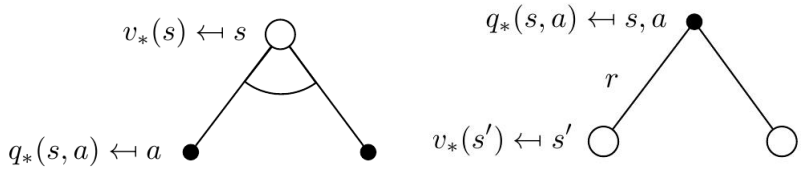
$$v^*(s) = \max_{a \in \mathcal{A}} q^*(s,a) \tag{3.59}$$

$$q^*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v^*(s') \tag{3.60}$$

$$v^*(s) = \max_{a \in \mathcal{A}} \left[\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v^*(s') \right] \tag{3.61}$$

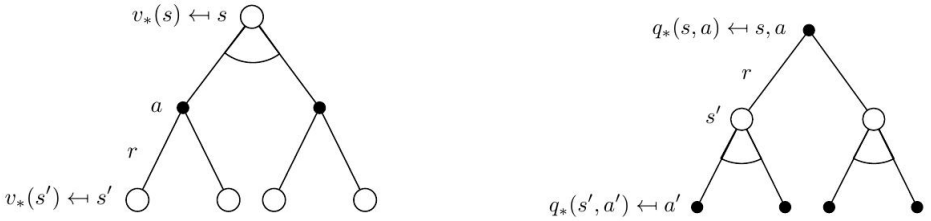
$$q^*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a' \in \mathcal{A}} q^*(s',a') \tag{3.62}$$

Figures 3.8–3.9 illustrate these relations with backup diagrams.



$$v_*(s) = \max_a q_*(s,a) \qquad q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$

Figure 3.8: Relationships between $v^*(s)$ and $q^*(s,a)$



$$v_*(s) = \max_a \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \qquad q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_*(s',a')$$

Figure 3.9: Fixed point equations for $v^*(s)$ and $q^*(s,a)$

Exercise 3.5. For Exercise 3.2, derive the expressions for the optimal Bellman equations (3.59)–(3.62). For convenience we recover the expressions here:

$$v^\pi = \left[\begin{array}{cc} v^\pi(x), & v^\pi(y) \end{array} \right]^T \tag{3.63}$$

$$q^\pi = \left[\begin{array}{cccc} q^\pi(x,u), & q^\pi(x,m), & q^\pi(y,u), & q^\pi(y,m) \end{array} \right]^T \tag{3.64}$$

$$\mathcal{R} = \left[\begin{array}{cccc} \mathcal{R}_x^u, & \mathcal{R}_x^m, & \mathcal{R}_y^u, & \mathcal{R}_y^m \end{array} \right]^T \tag{3.65}$$

$$\mathcal{P} = \left[\begin{array}{cc} \mathcal{P}_{xx}^u & \mathcal{P}_{xy}^u \\ \mathcal{P}_{xx}^m & \mathcal{P}_{xy}^m \\ \mathcal{P}_{yx}^u & \mathcal{P}_{yy}^u \\ \mathcal{P}_{yx}^u & \mathcal{P}_{yx}^m \end{array} \right] \tag{3.66}$$

a) $v^*(s) = \max_{a\in\mathcal{A}}\left[\mathcal{R}_s^a + \gamma\sum_{s'\in\mathcal{S}}\mathcal{P}_{ss'}^a v^*(s')\right]$

$$\begin{aligned} v^*(x) &= \max_{u,m}\left(\mathcal{R}_x^u + \gamma\begin{bmatrix}\mathcal{P}_{xx}^u & \mathcal{P}_{xy}^u\end{bmatrix}\begin{bmatrix}v^*(x)\\ v^*(y)\end{bmatrix}, \mathcal{R}_x^m + \gamma\begin{bmatrix}\mathcal{P}_{xx}^m & \mathcal{P}_{xy}^m\end{bmatrix}\begin{bmatrix}v^*(x)\\ v^*(y)\end{bmatrix}\right) \\ v^*(y) &= \max_{u,m}\left(\mathcal{R}_y^u + \gamma\begin{bmatrix}\mathcal{P}_{yx}^u & \mathcal{P}_{yy}^u\end{bmatrix}\begin{bmatrix}v^*(x)\\ v^*(y)\end{bmatrix}, \mathcal{R}_y^m + \gamma\begin{bmatrix}\mathcal{P}_{yx}^m & \mathcal{P}_{yy}^m\end{bmatrix}\begin{bmatrix}v^*(x)\\ v^*(y)\end{bmatrix}\right) \end{aligned} \tag{3.67}$$

b) $q^*(s,a) = \mathcal{R}_s^a + \gamma\sum_{s'\in\mathcal{S}}\mathcal{P}_{ss'}^a \max_{a'\in\mathcal{A}} q^*(s',a')$

$$\begin{aligned} q^*(x,u) &= \mathcal{R}_x^u + \gamma\begin{bmatrix}\mathcal{P}_{xx}^u & \mathcal{P}_{xy}^u\end{bmatrix}\begin{bmatrix}\max\limits_{u,m}(q^*(x,u), q^*(x,m))\\ \max\limits_{u,m}(q^*(y,u), q^*(y,m))\end{bmatrix} \\ q^*(x,m) &= \mathcal{R}_x^m + \gamma\begin{bmatrix}\mathcal{P}_{xx}^m & \mathcal{P}_{xy}^m\end{bmatrix}\begin{bmatrix}\max\limits_{u,m}(q^*(x,u), q^*(x,m))\\ \max\limits_{u,m}(q^*(y,u), q^*(y,m))\end{bmatrix} \\ q^*(y,u) &= \mathcal{R}_y^u + \gamma\begin{bmatrix}\mathcal{P}_{yx}^u & \mathcal{P}_{yy}^u\end{bmatrix}\begin{bmatrix}\max\limits_{u,m}(q^*(x,u), q^*(y,u))\\ \max\limits_{u,m}(q^*(y,u), q^*(y,m))\end{bmatrix} \\ q^*(y,m) &= \mathcal{R}_y^m + \gamma\begin{bmatrix}\mathcal{P}_{yx}^m & \mathcal{P}_{yy}^m\end{bmatrix}\begin{bmatrix}\max\limits_{u,m}(q^*(x,u), q^*(x,m))\\ \max\limits_{u,m}(q^*(y,u), q^*(y,m))\end{bmatrix} \end{aligned} \tag{3.68}$$

Finally, by defining vector $q^*_{\max} = \begin{bmatrix}\max\limits_{u,m}(q^*(x,u), q^*(x,m))\\ \max\limits_{u,m}(q^*(y,u), q^*(y,m))\end{bmatrix}$, we obtain a very compact expression for the optimal state-action value function:

$$q^* = \mathcal{R} + \gamma\mathcal{P}q^*_{\max} \tag{3.69}$$

Exercise 3.6. Consider a mobile robot has the job of collecting empty cans in an office environment. The environment state is the robot's battery level, which could be in two states: low or high. The robot has to decide whether *i)* actively search for cans, *ii)* be quiet and wait for someone to bring it a can or *iii)* move to its home base to recharge its battery. In order to find cans, the robot has to search them. But searching consumes the robot's battery, whereas waiting does not. The rewards can be zero if battery level is high and no can has been found (likely most of the time), positive when the robot secures an empty can and the battery remains high, or large and negative if

the robot runs out of battery (since, in this case, the robot shuts down and waits to be rescued, causing a big cost).

More formally, the state set is $\mathcal{S} = \{\text{high, low}\}$, the action set is $A = \{\text{search, wait, search, recharge}\}$ and the state transition probabilities are parameterized by parameters α, β, as described in Table 3.10 below. Figure 3.11 displays the state transition diagram.

s	s'	a	$p(s'\|s,a)$	$r(s,a,s')$
high	high	search	α	r_{search}
high	low	search	$1-\alpha$	r_{search}
low	high	search	$1-\beta$	-3
low	low	search	β	r_{search}
high	high	wait	1	r_{wait}
high	low	wait	0	r_{wait}
low	high	wait	0	r_{wait}
low	low	wait	1	r_{wait}
low	high	recharge	1	0
low	low	recharge	0	0.

Figure 3.10: Transition probabilities. The recycling robot problem

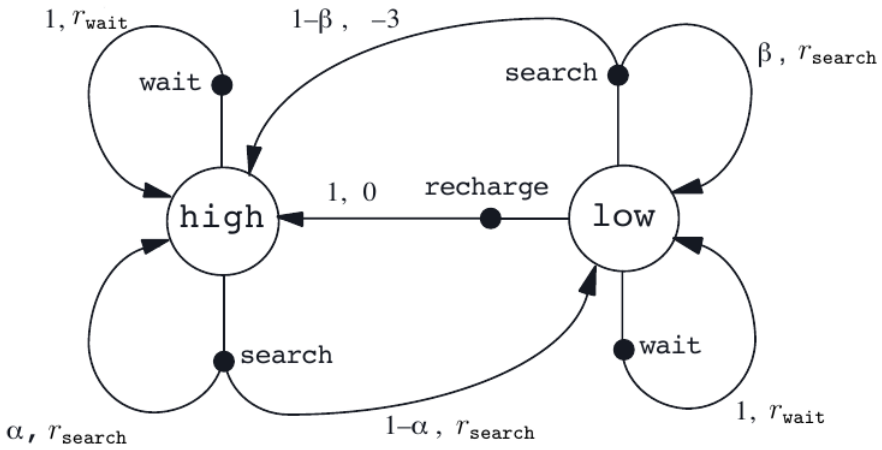


Figure 3.11: Transition graph. The recycling robot problem

Calculate the optimal Bellman equation for the state-value function.
Solution:

$$\begin{aligned} v^*(h) &= \max\left\{ \begin{array}{c} \mathbb{P}(h \mid h,s)(r_s + \gamma v^*(h)) + \mathbb{P}(l \mid h,s)(r_s + \gamma v^*(l)) \\ \mathbb{P}(h \mid h,w)(r_w + \gamma v^*(h)) + (1-\alpha)(r_w + \gamma v^*(l)) \end{array} \right\} \\ &= \max\left\{ \begin{array}{c} \alpha(r_s + \gamma v^*(h)) + \mathbb{P}(l \mid h,s)(r_s + \gamma v^*(l)) \\ 1(r_w + \gamma v^*(h)) + 0(r_w + \gamma v^*(l)) \end{array} \right\} \\ &= \max\left\{ \begin{array}{c} r_s + \gamma(\alpha v^*(h) + (1-\alpha) v^*(l)) \\ r_w + \gamma v^*(h) \end{array} \right\} \end{aligned} \tag{3.70}$$

$$v^*(l) = \max\left\{ \begin{array}{c} \beta r_s - 3(1-\beta) + \gamma((1-\beta) v^*(h) + \beta v^*(l)) \\ r_w + \gamma v^*(l) \\ \gamma v^*(l) \end{array} \right\} \tag{3.71}$$

Part II

Planning and Learning in Small-Scale Problems

Chapter 4

Planning by Dynamic Programming

Dynamic programming (DP) refers to a set of iterative procedures that solve the Bellman equations. The term "programming" is a standard mathematical synonym for optimization, and the term "dynamic" refers to the fact that we are optimizing a function over a time horizon. DP has been applied to several practical problems that have a fundamental property, defined as the *principle of optimality*, which in words of Richard Bellman is explained as follows:

> An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision. (See Bellman, 1957, Chap. III.3.) [Wikipedia]

In other words, the principle of optimality states that the problem can be decomposed into subproblems and the optimal solution of the problem can also be decomposed so that you get the optimal solution of the subproblems. This is exactly what happens in, e.g., the Bellman equation (3.61), where the optimal value function for state s is decomposed into the action that gives the maximal expected reward and the optimal value function for the expected next state. Practical problems that satisfy the principle of optimality include scheduling, shortest path problem, sequence alignment in text matching and genomics, optimal control problems, etc.

4.1 Bellman operators

Let us introduce two mappings that play an important theoretical role in optimal control. For any function $v : \mathcal{S} \to \mathbb{R}$ we consider the right hand side

of the Bellman equations (3.23) and (3.28) as a mapping, known the *Bellman operator* for policy π, and we denote it for $s \in \mathcal{S}$ by

$$\begin{aligned}(T_\pi v)(s) &\triangleq \mathbb{E}_{\pi,\mathcal{P}}\left[R_{t+1} + \gamma v^\pi(S_{t+1}) | S_t = s\right] \\ &= \sum_{a\in\mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v^\pi(s')\right) \end{aligned} \tag{4.1}$$

Note that (4.1) depends on the specific policy that we use for averaging actions and, hence, state transitions. For the specific case of the optimal policy, we define the *optimal Bellman operator*, T, also for any function $v : \mathcal{S} \to \mathbb{R}$, as the right hand side of the optimal Bellman equations (3.61) for $s \in \mathcal{S}$:

$$\begin{aligned}(Tv)(s) &\triangleq \max_{a\in\mathcal{A}} \mathbb{E}_{\mathcal{P}}\left[R_{t+1} + \gamma v(S_{t+1}) | S_t = s\right] \\ &= \max_{a\in\mathcal{A}} \left[\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v(s')\right] \end{aligned} \tag{4.2}$$

By slightly abusing notation, we can also define the same Bellman operators for any state-action value function $q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ from (3.24)–(3.29) and (3.51) and (3.62) for $(s, a) \in \mathcal{S} \times \mathcal{A}$

$$\begin{aligned}(T_\pi q)(s,a) &\triangleq \mathbb{E}_{\pi,\mathcal{P}}\left[R_{t+1} + \gamma q^\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a\right] \\ &= \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a'\in\mathcal{A}} \pi(a'|s') q^\pi(s', a') \end{aligned} \tag{4.3}$$

$$\begin{aligned}(Tq)(s,a) &\triangleq \max_{a\in\mathcal{A}} \mathbb{E}_{\mathcal{P}}\left[R_{t+1} + \gamma q(S_{t+1}, A_{t+1}) | S_t = s, A_t = a\right] \\ &= \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'\in\mathcal{A}} q(s', a') \end{aligned} \tag{4.4}$$

Whether we refer to the Bellman operator for state or state-action value functions should be clear from the context.

Let us express the Bellman operators for some given policy π in vector form:

$$T_\pi v \triangleq \left((T_\pi v)(s)\right)_{s\in\mathcal{S}} = \mathcal{R}^\pi + \gamma \mathcal{P}^\pi v \tag{4.5}$$

$$T_\pi q \triangleq \left((T_\pi q)(s,a)\right)_{(s,a)\in\mathcal{S}\times\mathcal{A}} = \mathcal{R} + \gamma \mathcal{P} \Pi q \tag{4.6}$$

Similarly, let us express the optimal Bellman operator in vector form:

$$Tv \triangleq \left((Tv)(s)\right)_{s\in\mathcal{S}} \tag{4.7}$$

$$Tq \triangleq \left((Tq)(s,a)\right)_{(s,a)\in\mathcal{S}\times\mathcal{A}} \tag{4.8}$$

Then, we can rewrite the Bellman equations very compactly as follows:

$$v^{\pi} = T_{\pi} v^{\pi} \tag{4.9}$$
$$q^{\pi} = T_{\pi} q^{\pi} \tag{4.10}$$
$$v^{*} = T v^{*} \tag{4.11}$$
$$q^{*} = T q^{*} \tag{4.12}$$

These Bellman operators satisfy a very important property known as *Contraction Mapping*. Before proving this, we first introduce the concept of contraction mappings.

Definition. A mapping $F : \mathcal{X} \to \mathcal{X}$ is a contraction mapping, or contraction, if there exists a constant c, with $0 \leq c < 1$, such that

$$\|F(v) - F(v')\| \leq c\|v - v'\| \tag{4.13}$$

for all $v, v' \in \mathcal{X}$, where $\mathcal{X}$ is some space with some metric $\|\cdot\|$.

In order to prove that the Bellman operators are contractions we have to define the infinity norm.

Definition 4.1. The infinity norm (or max-norm), denoted ∞-norm, between to value vectors, $v \in \Re^{|\mathcal{S}|}$ and $v' \in \Re^{|\mathcal{S}|}$, is given by the largest difference between each of their componets:

$$\|v - v'\|_{\infty} = \max_{s \in \mathcal{S}} |v(s) - v'(s)| \tag{4.14}$$

We also need the following assumption on the reward.

Assumption 2. *The expected instantaneous reward obtained after transition from s to s' when taking action a is bounded, i.e., there exists some scalar B so that*

$$|R_t| \leq B, \quad \forall (s, a, s') \in \mathcal{S} \times \mathcal{A} \times \mathcal{S} \tag{4.15}$$

We are ready to prove that Bellman operators are contractions.

Proposition 4.1. *Let $0 < \gamma < 1$ and let Assumption 2 hold. Then, the Bellman operators, T_{π} and T, are contractions:*

$$\|T_{\pi} v - T_{\pi} v'\|_{\infty} \leq \gamma \|v - v'\|_{\infty}, \qquad \forall v, v' \in \Re^{|\mathcal{S}|} \tag{4.16}$$
$$\|T v - T v'\|_{\infty} \leq \gamma \|v - v'\|_{\infty}, \qquad \forall v, v' \in \Re^{|\mathcal{S}|} \tag{4.17}$$

Proof. The proof is similar for T_π and T, so that we only prove for the former. First, note that since $\mathcal{P}^\pi$ is a stochastic matrix, all its elements are less or equal to 1. Then, we have

$$\|\mathcal{P}^\pi v\|_\infty \leq \|v\|_\infty \tag{4.18}$$

Now, we use this relationship in the operator definition:

$$\begin{aligned} \|T_\pi(v) - T_\pi(v')\|_\infty &= \|\mathcal{R}^\pi + \gamma\mathcal{P}^\pi v - (\mathcal{R}^\pi + \gamma\mathcal{P}^\pi v')\|_\infty \\ &= \gamma\|\mathcal{P}^\pi(v - v')\|_\infty \\ &\leq \gamma\|v - v'\|_\infty \end{aligned} \tag{4.19}$$

□

Contractions have very interesting and useful properties. Hence, having shown that the Bellman operators are contractions is a great deal. In particular, we can apply the Contraction Mapping Theorem, or Banach Fixed Point Theorem, as follows.

Theorem 4.1. *The following holds for the Bellman operator for some policy π:*

1. There exists a unique solutions v^π to the fixed point equation $T_\pi v = v$.
2. For any $v_0 \in \Re^{|\mathcal{S}|}$ the fixed point iterates given by $v_{t+1} = T_\pi(v_t)$ converge to v^π as $t \to \infty$.
3. The iterates v_t satisfy the following error bounds:

$$\|v_t - v^\pi\| \leq \frac{\gamma^t}{1-\gamma}\|v_1 - v_0\| \tag{4.20}$$

$$\|v_t - v^\pi\| \leq \frac{\gamma}{1-\gamma}\|v_t - v_{t-1}\| \tag{4.21}$$

Theorem 4.2. *The following holds for the optimal Bellman operator:*

1. There exists a unique solution v^* to the fixed point equation $Tv = v$.
2. For any $v_0 \in \Re^{|\mathcal{S}|}$ the fixed point iterates given by $v_{t+1} = T(v_t)$ converge to v^* as $t \to \infty$.
3. The iterates v_t satisfy the following error bounds:

$$\|v_t - v^*\| \leq \frac{\gamma^t}{1-\gamma}\|v_1 - v_0\| \tag{4.22}$$

$$\|v_t - v^*\| \leq \frac{\gamma}{1-\gamma}\|v_t - v_{t-1}\| \tag{4.23}$$

In words, Theorems 4.1 and 4.2 state that if we recursively apply the (optimal) Bellman operator to any function $v : \mathcal{S} \to \Re$, we will asymptotically obtain the (optimal) state-value function. This suggests iterative methods for finding the state-value functions v^π (for some given policy π) or for finding the optimal state-value function v^*. Similar conclusion can be obtained for the state-action value functions, q^π and q^*. In the following section we use this idea to introduce two related algorithms, named Value Iteration (VI) and Policy Iteration (PI), together known as Dynamic Programming (DP) methods.

4.2 Dynamic Programming methods

When dealing with MDPs, we consider two different problems:

- **Prediction**: consists in finding the state or state-action value function for a given policy. This is usually known as policy evaluation because the value function tells us how good such policy is (in terms of expected cumulative reward).
- **Control**: consists in finding the optimal policy, i.e., the policy that yields the optimal value function.

DP methods are mainly derived from the convergence properties of the Bellman operators introduced in Theorem 4.1.

4.2.1 Prediction

For the *prediction* problem, we consider the following iterates derived from Theorem 4.1:

$$v_{t+1} = T_\pi(v_t) = \mathcal{R}^\pi + \gamma \mathcal{P}^\pi v_t \tag{4.24}$$

$$q_{t+1} = T_\pi(q_t) = \mathcal{R} + \gamma \mathcal{P} \Pi q_t \tag{4.25}$$

These recursions yield the so named *policy evaluation* (PE) algorithm, which asymptotically returns the value functions, v^π or q^π, for some given policy π. The pseudocode for PE is given by Algorithms 4.1 and 4.2 for the state and state-action value functions, respectively. In practice, for the prediction problem, the state value function, v^π, is all what we need to evaluate how good a policy is and q^π is not usually computed. For the control problem the situation is different and there exists practical algorithmic variants for both the state and state-action value functions. Furthermore, as we will see

Algorithm 4.1 Policy Evaluation for the state-value function.

Input: π, the policy to be evaluated.
Output: v^π, the state value function for policy π.

Initialize $v(s)$ arbitrarily (e.g., $v(s) = 0$), for all $s \in \mathcal{S}$
repeat
  $\Delta \leftarrow 0$
  for each $s \in \mathcal{S}$ **do**
    $v_{\text{old}} \leftarrow v(s)$
    $v(s) \leftarrow \sum_{a\in\mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v(s')\right)$
    $\Delta \leftarrow \max\left[\Delta, |v_{\text{old}} - v(s)|\right]$
until $\Delta < \epsilon$
return v

Algorithm 4.2 Policy Evaluation for the state-action value function.

Input: π, the policy to be evaluated.
Output: q^π, the state-action value function for policy π.

Initialize $q(s,a)$ arbitrarily (e.g., $q(s,a) = 0$), for all $(s,a) \in \mathcal{S} \times \mathcal{A}$
repeat
  $\Delta \leftarrow 0$
  for each $(s,a) \in \{\mathcal{S} \times \mathcal{A}\}$ **do**
    $q_{\text{old}} \leftarrow q(s,a)$
    $q(s,a) \leftarrow \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a\in\mathcal{A}} \pi(a|s') q(s',a)$
    $\Delta \leftarrow \max\left[\Delta, |q_{\text{old}} - q(s,a)|\right]$
until $\Delta < \epsilon$
return q

in the following chapters, the state-action value function is usually preferred for learning the optimal policies.

Figure 4.1 illustrates the process of updating $v_{t+1}(s)$ from $v_t(s)$.

Exercise 4.1. Implement PE in Matlab and compute the value function for each of the four deterministic policies and problem parameters given in Exercise 3.2. Check that the result obtained through PE is consistent with the theoretical result obtained in previous chapter. Finally, check that the error bounds (4.20)–(4.23).

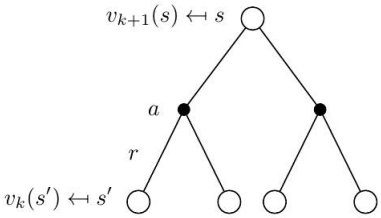


Figure 4.1: Scheme used for updating the value in PE.

4.2.2 Control

The goal of the control problem is to obtain the optimal policy (i.e., the policy with largest value function). In order to improve some given policy π to obtain another policy π', such that $v^{\pi'} \geq v^{\pi}$, the agent could consider whether it should change the policy at a certain state to $a \neq \pi(s)$. One way to check whether the resulting policy after such change has improved the value function is to evaluate the result of selecting a at s and follow $\pi(s)$ thereafter, which is equivalent to compute the state-action value function, or simply q-function:

$$q^{\pi}(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v^{\pi}(s') \tag{4.26}$$

One intuitive way to find a policy that improves $q^{\pi}(s,a)$ is the greedy policy, $\pi'(s)$, given by (3.56), such that

$$\pi'(s) = \arg\max_{a \in \mathcal{A}} q^{\pi}(s,a) \tag{4.27}$$

Theorem *4.3*, known as *policy improvement theorem* formalizes these ideas.

Theorem 4.3. *Let $\pi'(s)$ and $\pi(s)$ be any pair of deterministic policies that satisfy the following inequality for all $s \in \mathcal{S}$:*

$$q^{\pi}(s, \pi'(s)) \geq v^{\pi}(s), \quad \forall s \in \mathcal{S} \tag{4.28}$$

Then, we have that

$$v^{\pi'}(s) \geq v^{\pi}(s) \tag{4.29}$$

In other words, Theorem *4.3* establishes that by replacing $\pi(s)$ with the greedy policy $\pi'(s)$ given by (4.27), the resulting value function $v^{\pi'}$ is better or equal than the original value v^{π}. The algorithm consisting in iterating these two stages of policy evaluation and policy improvement is known as *policy*

iteration (PI) and is typically represented as a sequence of monotonically improving (I) policies of evaluated (E) value functions:

$$\pi_0(s) \xrightarrow{E} v^{\pi_0}(s) \xrightarrow{I} \pi_1(s) \xrightarrow{E} v^{\pi_1}(s) \xrightarrow{E} \cdots \xrightarrow{I} \pi^*(s) \xrightarrow{E} v^*(s) \tag{4.30}$$

$$\pi_0(s) \xrightarrow{E} q^{\pi_0}(s,a) \xrightarrow{I} \pi_1(s) \xrightarrow{E} q^{\pi_1}(s,a) \xrightarrow{E} \cdots \xrightarrow{I} \pi^*(s) \xrightarrow{E} q^*(s,a) \tag{4.31}$$

An alternative approach is to directly apply the *optimal Bellman operator* to the value function in a recursive manner, as derived from Theorem (4.2). Since the optimal Bellman operator is a contraction, this procedure, known as *value iteration* (VI), also converges to the optimal value function. Then, we can get the optimal policy as the one that is greedy with respect to the optimal value function for every state.

Interestingly, we can see a relationship between VI and PI as follows. As we have discussed, at every step, PI performs policy evaluation, meaning that we have to wait until convergence to the value function of the current policy. Nevertheless, instead of waiting until convergence, we could tolerate some error and truncate the policy evaluation process at some point, so that we then perform the policy improvement step over the truncated approximation of the value function. Now, consider the extreme case where we only apply one iteration of the policy evaluation loop and then take the greedy policy. This extreme case is equivalent to applying the optimal Bellman operator. In other words, we can see VI as an extreme case of PI, where the policy evaluation step is approximated with just one single iteration. As an intermediate case, we can consider more than one iteration of the policy evaluation step before improving the policy. This approach is known as *generalized policy iteration* (GPI) and has PI and VI as particular extreme cases. Figure (4.2) shows schematically this process.

In the next subsections we show the steps of PI and VI in more detail. These DP algorithms are the base of the learning algorithms that we will study in posterior next chapters. Therefore, they should be fully understood. We will derive both algorithms for both state and state-action value functions. However, from the point of view or learning, the state-action variant is usually more useful (with the exception of policy search algorithms, where the state value function variant is preferred).

Policy iteration

Policy iterations evaluate policies by constructing their value functions, and use these value functions to find new improved policies. The algorithm starts

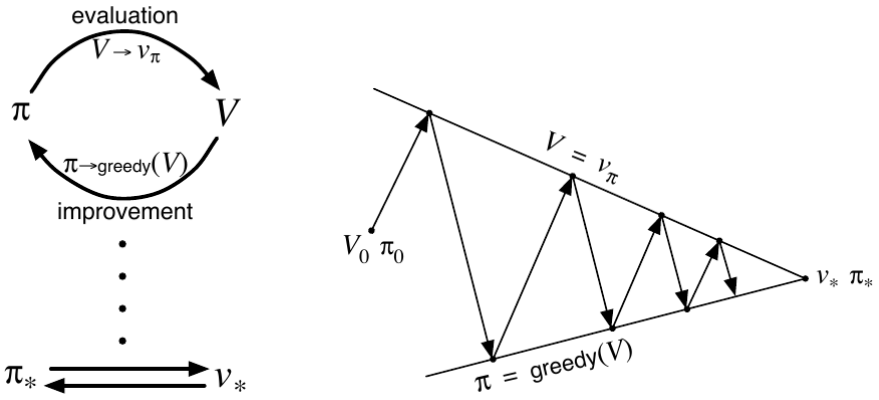


Figure 4.2: Generalized policy iteration algorithm.

with an arbitrary policy π_0. At every iteration t the value function of the current policy, v^{π_t} or q^{π_t}, is determined by using the PE algorithm, given by Algorithm 4.1 or Algorithm 4.2, which recursively applies (4.24) or (4.25). Once PE is complete, we improve the policy by taking the greedy policy given by (4.27):

$$\pi_{t+1}(s) \in \arg\max_{a \in \mathcal{A}} q^{\pi_t}(s,a), \quad \forall s \in \mathcal{S} \tag{4.32}$$

The pseudocode for PI for the state and state-action value functions is given by Algorithms (4.3) and (4.4), respectively.

Exercise 4.2. Implement PI in Matlab and compute the optimal policy and optimal value function for the problem described in Exercise 3.2. Compare the results obtained through PI with the theoretical result obtained in Exercise (3.5) and with the results obtained by using PE in Exercise (4.1).

Value iteration

Value iteration starts from some initial value function, v_0 or q_0, and then uses the following recursions on the optimal Bellman operator, derived from Theorem (4.2):

$$v_{t+1} = Tv_t \tag{4.33}$$

$$q_{t+1} = Tq_t \tag{4.34}$$

Algorithm 4.3 Policy Iteration for state value functions.

Output: π^*, the optimal policy.

Initialize $v(s)$ and $\pi(s)$ arbitrarily, for all $s \in \mathcal{S}$
while θ is false **do** ▷ Policy iteration main loop
  repeat ▷ Policy evaluation loop
    $\Delta \leftarrow 0$
    for each $s \in \mathcal{S}$ **do**
      $v_{\text{old}} \leftarrow v(s)$
      $v(s) \leftarrow \mathcal{R}_s^{\pi(s)} + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^{\pi(s)} v(s')$
      $\Delta \leftarrow \max\left[\Delta, |v_{\text{old}} - v(s)|\right]$
  until $\Delta < \epsilon$ (a small positive number)
  $\theta \leftarrow$ true
  for each $s \in \mathcal{S}$ **do** ▷ Policy improvement loop
    $a \leftarrow \pi(s)$
    $\pi(s) \leftarrow \arg\max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v(s')\right)$
    if $a \neq \pi(s)$ **then** $\theta \leftarrow$ false (policy is unstable)
return π

which are detailed for every state and state-action pair, respectively, as follows:

$$v_{t+1}(s) = \max_{a \in \mathcal{A}} \left[\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_t(s') \right]$$
$$q_{t+1}(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a' \in \mathcal{A}} q_t(s', a')$$

Due to the contraction property of the optimal Bellman operator stated by Theorem (4.2), we can guarantee the convergence of this algorithm to the optimal value functions, $v^*(s)$ or $q^*(s,a)$. We recall from Lemma (3.3) that any policy that is is greedy with respect to the optimal value function will be optimal. We rewrite Lemma (3.3) with the Bellman operator notation in Prop. 4.2.

Proposition 4.2. *A stationary policy $\pi^* : \mathcal{S} \to \mathcal{A}$ is optimal if and only if $\pi^*(s)$ attains the optimal value function (3.61) for each $s \in \mathcal{S}$. This is compactly expressed as follows*

$$T_{\pi^*} v^* = T v^* \tag{4.35}$$

Equivalently, we can say that π^* is greedy with respect to the optimal value

Algorithm 4.4 Policy Iteration for state-action value functions.

Output: π^*, the optimal policy.

Initialize $q(s,a)$ arbitrarily (e.g., $q(s,a)=0$), for all $s \in \mathcal{S}, a \in \mathcal{A}$
while θ is false **do** ▷ Policy iteration main loop
  repeat ▷ Policy evaluation loop
    $\Delta \leftarrow 0$
    for each $(s,a) \in \{\mathcal{S} \times \mathcal{A}\}$ **do**
      $q_{\text{old}} \leftarrow q(s,a)$
      $q(s,a) \leftarrow \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a \in \mathcal{A}} \pi(a|s') q(s',a)$
      $\Delta \leftarrow \max\left[\Delta, |q_{\text{old}} - q(s,a)|\right]$
  until $\Delta < \epsilon$ (a small positive number)
  $\theta \leftarrow$ true
  for each $s \in \mathcal{S}$ **do** ▷ Policy improvement loop
    $a \leftarrow \pi(s)$
    $\pi(s) \leftarrow \arg\max_{a' \in \mathcal{A}} q(s,a')$
    if $a \neq \pi(s)$ **then** $\theta \leftarrow$ false (policy is unstable)
return π

function:

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} v^*(s) = \arg\max_{a \in \mathcal{A}} q^*(s,a), \quad \forall s \in \mathcal{S} \tag{4.36}$$

Thus, once we have found v^*, we can easily obtain the optimal policy by computing the greedy policy with respect to v^*. Pseudocode of VI is shown in Algorithm .

Exercise 4.3. Implement VI in Matlab and compute the optimal policy and optimal value function for the problem described in Exercise 3.2. Compare the results obtained through VI with the theoretical result obtained in Exercise (3.5). Compare also the convergence curve and execution time of VI and PI.

Relationship and comparison of PI and VI

We have introduced PI as a two steps procedure: *i)* apply the Bellman operator for the current best policy in a recursive manner until convergence to the value function for such policy; then, *ii)* we take the greedy policy for

Algorithm 4.5 Value Iteration for state- value functions.

Output: π^*, the optimal policy.

Initialize $v(s)$ arbitrarily (e.g., $v(s) = 0$), for all $s \in \mathcal{S}$
repeat
  $\Delta \leftarrow 0$
  for each $s \in \mathcal{S}$ **do**
    $v_{\text{old}} \leftarrow v(s)$
    $v(s) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v(s') \right)$
    $\Delta \leftarrow \max \left[\Delta, |v_{\text{old}} - v(s)| \right]$
until $\Delta < \epsilon$
for each $s \in \mathcal{S}$ **do**
  $\pi(s) \leftarrow \arg\max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v(s') \right)$
return π

Algorithm 4.6 Value Iteration for state-action value functions.

Output: π^*, the optimal policy.

Initialize $q(s, a)$ arbitrarily (e.g., $q(s, a) = 0$), for all $s \in \mathcal{S}, a \in \mathcal{A}$
repeat
  $\Delta \leftarrow 0$
  for each $(s, a) \in \{\mathcal{S} \times \mathcal{A}\}$ **do**
    $q_{\text{old}} \leftarrow q(s, a)$
    $q(s, a) \leftarrow \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a \in \mathcal{A}} q(s', a)$
    $\Delta \leftarrow \max \left[\Delta, |q_{\text{old}} - q(s, a)| \right]$
until $\Delta < \epsilon$
for each $s \in \mathcal{S}$ **do**
  $\pi(s) \leftarrow \arg\max_{a \in \mathcal{A}} q(s, a)$
return π

the computed value function. On the other hand, we have presented VI as the result of directly applying the optimal Bellman operator to the value function in a recursive manner, so that it converges to the optimal value. Then, we simply obtain the optimal policy as greedy policy with respect to the optimal value function. In addition, we have discussed another useful interpretation that shows that PI and VI are two extreme cases of GPI.

There are two key issues to compare the algorithmic complexity of VI and PI from:

- Double loop vs. single loop: PI requires (surprisingly) few iterations of the outer loop. Indeed, for finite-state MDP's, there exists a finite number of deterministic policies. Hence, PI must converge to the opti-

mal policy in a finite number of steps. Nevertheless, each of these steps requires to solve the Bellman equation (i.e., policy evaluation problem), which could be accomplished by, e.g., solving the linear system of equations (using numerical linear algebra) or by finding its fixed point, i.e., iterating the Bellman operator in a second inner loop. Typically, for large state-action sets, the complexity of finding the fixed point is much smaller than the complexity of solving the system of equations. But, in any case, policy evaluation is costly. On the other hand, VI usually requires several iteration of its single loop before convergence, but each iteration is much faster since it requires no further inner loop. From this point of view, VI usually requires less computational resources.

- Maximization over the action set: If the action set is large or even continuous, maximization is very costly. PI performs the maximization over the action set at each iteration of the outer loop, which, again, requires very few iterations. On the other hand, VI performs the maximization over the action set at each iteration of the inner loop, which usually requires many iterations to converge. From this maximization point of view, PI should require less computational resources.

4.3 Case study 1. Simplified cleaning robot problem. Practical implementation.

Consider a cleaning robot that has to move to collect a used can but it also has to recharge its batteries. The state space is discrete and contains 6 distinct states, denoted by integers $\mathcal{S} = \{0, 1, 2, 3, 4, 5\}$, that indicate the position of the robot in a grid. The robot can move to the left ($a = -1$) or to the right ($a = 1$); the discrete action space is therefore $\mathcal{A} = \{1, -1\}$. States 0 and 5 are terminal (absorbing) states, meaning that once the robot reaches any of these terminal states, it can no longer leave, regarding of the action. The discounting factor is $\gamma = 0.9$.

Due to stochastic behavior of the environment (e.g., slippery floor) the robot moves in the desired direction with probability 0.8, remains in the same position with probability 0.15 and moves in the opposite direction with probability 0.05.

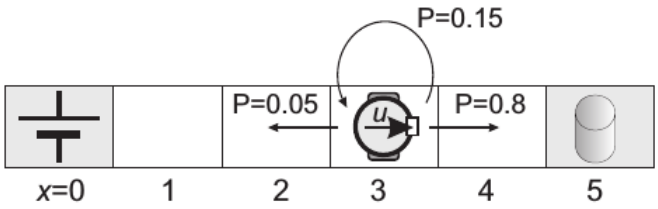


Figure 4.3: The cleaning robot problem with uncertainty

The reward function is given in terms of the state-to-state-given action transition, denoted $R^a_{ss'}$:

$$R^a_{ss'} = \begin{cases} 5 & \text{if } s \neq 5 \quad \text{and } s' = 5 \\ 1 & \text{if } s \neq 0 \quad \text{and } s' = 0 \\ 0 & \text{Otherwise} \end{cases} \tag{4.37}$$

This is different from the expected state-action reward, $\mathcal{R}^a_s$, that we have seen so far. This should not cause any trouble, since

$$\mathcal{R}^a_s = \sum_{s' \in \mathcal{S}} R^a_{ss'}$$

a) Calculate (analytically) the state value function for the uniform random policy (same probability of taking both actions at any state).

b) Use your Matlab implementations to find the optimal value function and the associated optimal policy with PI and VI.

4.4 Case study 2. The Grid-World MDP problem. Practical implementation

Consider a simple grid-world problem, where the environment is a rectangular grid. The state of the environment is given by the location of a robot in the grid. Hence, the cells of the grid correspond to the possible states of the environment. At each cell, four actions are possible: *north, south, east and west*, which deterministically (i.e., with probability 1) cause the agent to move one cell in the respective direction of the grid. Actions that would take the robot off the grid leave its location unchanged, but also results in a negative (undesired) reward of -1. Other actions result in a reward of 0, except those that move the agent out of the special states A and B. From

state A, all four actions yield a positive (desired) reward of +10 and take the agent to A'. From state B, all four actions yield a positive reward of +5 and take the agent to B'.

a) Suppose the agent selects all four actions with equal probability in all states (uniform random policy). Calculate analytically the value function for the random policy when $\gamma = 0.9$.

Solution:

Figure 4.4: The grid-world problem. Policy evaluation

b) Use your Matlab implementations of PI and VI to compute the optimal value function and the associated optimal policy.

Solution:

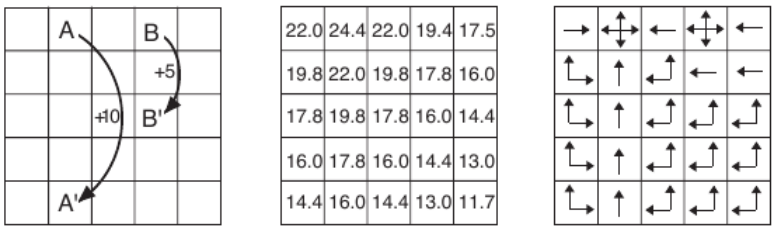


Figure 4.5: The grid-world problem. Optimal Policy. When there are multiple arrows in a cell, any of the corresponding actions is optimal.

Chapter 5

Model-free methods

In this chapter we consider problems where the state-transition probability distribution and/or the reward distribution are unknown. Hence, the agent has to learn these distributions from interaction with the environment. In other words, we move from planning with DP methods to learning with RL algorithms. This setting is usually known as *model-free*, in the sense that the agent learns without having any prior model of the environment, which is in contrast to model-based methods that assumed perfect knowledge of the environment, like the DP methods discussed in the previous chapter.

Similar to planning, we will distinguish two subproblems, namely prediction, which consists in estimating the value function for some given policy; and *control*, which consists in finding the optimal policy. Again, the difference now is that we will solve these problems for an unknown MDP.

One of the key ideas that we are going to introduce is *bootstrapping*, which could be defined as estimating the value function at some state from another estimate of the same value function at other states. The idea is to solve the Bellman equation using estimates of the value function. This idea turns to be very effective and will lead us to *Temporal Difference (TD)* and *Q-learning*, which are very useful RL algorithm for learning to solve the prediction and control problems, respectively. Other interesting ideas that will be covered include off-policy learning—for learning from one behavior policy that is different than the one that we aim to predict—and eligibility traces, which will be useful for setting a trade-off between the bias and variance of our estimates of the value function.

5.1 Prediction

The prediction problem aims to estimate the value function of some given policy for some MDP. For the planning problem, we have seen that policy evaluation is able to predict the value by taking the expected value of the future rewards. This is representation of that task in Figure 5.1, where the red area indicates that in order to predict the value at S_t, we average the reward over all possible state transitions S_{t+1}.

$$V(S_t) \leftarrow \mathbb{E}_\pi \left[R_{t+1} + \gamma V(S_{t+1}) \right]$$

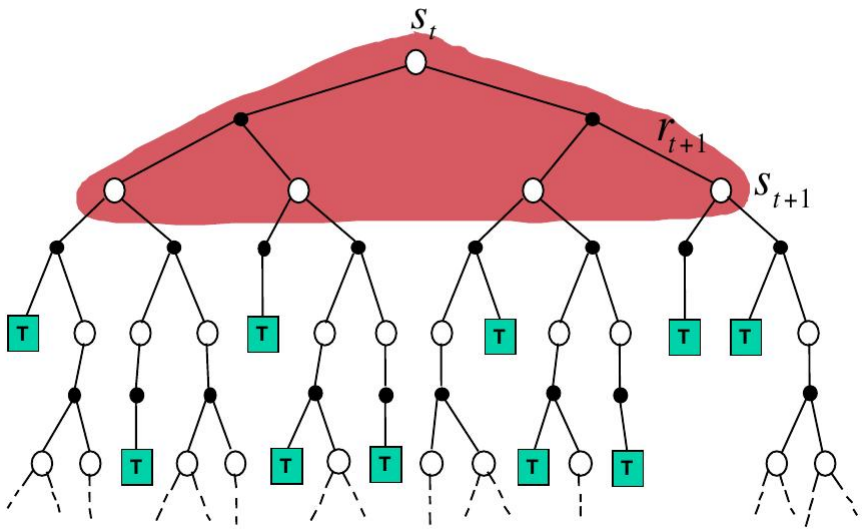


Figure 5.1: Dynamic Programming backup

When we have no access to the state-transition and reward distributions, we cannot compute the exact expected reward. Fortunately, we can use Monte Carlo (MC) methods to estimate the expected reward. This is the main idea underlying model-free RL algorithms. First, we present MC learning for prediction, then we will introduce the basic TD algorithm.

5.1.1 Monte-Carlo (MC) prediction

Recall from (3.20) and (3.21), that the value function for some policy π is the expected return (i.e., the cumulative reward) over all possible trajectories

induced by π. For convenience, we write the definition of return here again:

$$G^{\pi}(S_t) \triangleq R_{t+1} + \gamma R_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{5.1}$$

where we have introduced a more specific notation: $G^{\pi}(S_t) \triangleq G_t$.

One natural approach for estimating v^{π} consists in interacting with environment to obtain a sequence of samples of the form:

$$\begin{aligned}\{S_0 = s, A_0 = a_0, R_1 = r_1, S_1 = s', A_1 = a_1, R_2 = r_2,\\ S_2 = s, A_2 = a_2, R_3 = r_3, \ldots\}\end{aligned} \tag{5.2}$$

where $A_t \sim \pi$ and the state-transitions and rewards follow unknown distributions. This way, the agent can gather actual reward samples, $\{r_{t+1}, r_{t+2}, r_{t+3}, \ldots\}$and compute the actual return for that reward sequence as

$$g^{\pi}(S_t) = r_1 + r_2 + r_3 + \ldots \tag{5.3}$$

In other words, $g^{\pi}(S_t)$ denotes a realization of $G^{\pi}(S_t)$ and sequence (5.2) is one sample drawn from the joint distribution

$$p(S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots | \pi, \mathcal{P}) \tag{5.4}$$

Hence, an unbiased estimate of the expected return can be obtained by drawing several sample sequences like (5.2) and then computing and averaging their returns. More formally, let $g_l^{\pi}(s)$ denote a realization of $G^{\pi}(s)$ that includes the return for some particular reward sequence l that starts at initial state $S_t = s$. Suppose that we draw L sequences, all of them starting at s and following policy π. Then, we can estimate the value function $v^{\pi}(s)$ as follows:

$$v(s) \triangleq \frac{1}{L} \sum_{l=1}^{T} g_l^{\pi}(s) \tag{5.5}$$

This idea of using the *sample mean* of the return instead of the *expected return* is known as MC policy evaluation and is illustrated in Figure 5.2, where we can see that a realization of the return is computed from the current state and until the end of the episode (we are assuming episodic MDPs, with terminal states, in the picture).

$$V(S_t) \leftarrow V(S_t) + \alpha\left(G_t - V(S_t)\right)$$

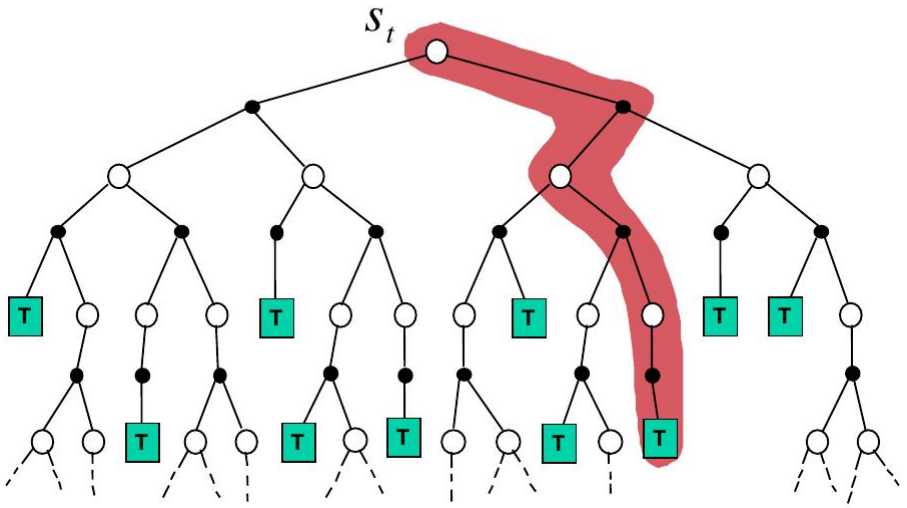


Figure 5.2: Monte-Carlo backup

There are two common implementations for MC policy evaluation, namely *first-visit* MC and *every-visit* MC. The every-visit MC method estimates the value function of some state as the average of the returns that have followed visits to it. Suppose we have the following sequence obtained by following π:

$$\{S_0 = s, A_0 = a_0, R_1 = r_1, S_1 = s', A_1 = a_1, R_2 = r_2, \\ S_2 = s, A_2 = a_2, R_3 = r_3\} \tag{5.6}$$

In this sequence, we visit state s twice ($S_0 = s$ and $S_2 = s$). Hence, the every-visit MC estimate of the value function for state s obtained from this sequence is given by:

$$g_1^\pi(s) = r_1 + r_2 + r_3 \tag{5.7}$$

$$g_2^\pi(s) = r_3 \tag{5.8}$$

$$v(s) = \frac{1}{2}\left(g_1^\pi(s) + g_2^\pi(s)\right) = \frac{1}{2}\left(r_1 + r_2 + 2r_3\right) \tag{5.9}$$

This method could be also applied to MDPs with no terminal states, granted that the sequence is so long enough that the discount factor is small when we truncate the sequence to a finite value.

On the other hand, the first-visit MC method can only be applied to MDPs with terminal states. The idea of first-visit MC is to average the

returns following the first time that the state was visited in each episode. For instance, for sequence (5.6), the first-visit MC estimate is given by:

$$g_1^{\pi}(s) = r_1 + r_2 + r_3 \tag{5.10}$$
$$v(s) = g_1^{\pi}(s) = r_1 + r_2 + r_3 \tag{5.11}$$

Both every-visit and first-visit MC converge to the true expected values as the number of visits to each state approaches infinity. This is a well known result derived from the law of large numbers.

Sometimes, it is convenient to compute the average of the available returns (either every or first visit) in an iterative manner. For instance, this is convenient for updating the value estimate as soon as we gather a new return, instead of having to wait until all returns have been gathered. Let $N(s)$ denote a counter that indicates the number of returns available for state s. Then, we can update the value estimate whenever some new return $g_l^{\pi}(s)$, for $l = 0, \ldots, \infty$, is available as follows:

$$v_{l+1}(s) = v_l(s) + \tfrac{1}{N(s)}\left(g_l^{\pi}(s) - v_l(s)\right) \tag{5.12}$$

where $v_{l+1}(s)$ denotes the estimate available when $g_l^{\pi}(s)$ arrives, and where $v_0(s)$ can be initialized to any value. Indeed, recursion (5.12) is an stochastic approximation where we can replace $N(s)$ with a more general sequence of step-sizes, $\{\alpha_l\}_{l=0}^{\infty}$, as follows:

$$v_{l+1}(s) \;\; = v_l(s) + \alpha_{l+1}\left(g_l(s) - v_l(s)\right) \tag{5.13}$$

This more general recursion also converges to the true value function $v^{\pi}(s)$ as long as the step-size sequence satisfies the following properties:

$$\sum_{l=0}^{\infty} \alpha_l = \infty \quad \text{and} \quad \sum_{l=0}^{\infty} \alpha_l^2 < \infty \tag{5.14}$$

A particular sequence of steps that satisfy these conditions is given by

$$\alpha_l = \frac{\beta}{l}, \quad \beta > 0 \tag{5.15}$$

The pseudocode for every-visit MC for predicting state and state-action value functions, for episodic MDPs, with iterative updates is given by Algorithms 5.1 and 5.2, respectively.

The main disadvantage of MC policy evaluation is that the joint distribution (5.4) is defined over the Cartesian product $\{\mathcal{S} \times \mathcal{A} \times \Re \times \mathcal{S} \times \mathcal{A} \times \Re \times \ldots\}$,

Algorithm 5.1 Every-visit MC policy evaluation for state-value functions and for episodic MDPs.

Input: π, the policy to be evaluated.
Output: v, the estimate of the value function v^π.

Initialize $v(s)$ arbitrarily (e.g., $v(s) = 0$), for all $s \in \mathcal{S}$
Initialize $N(s) = 0$, for all $s \in \mathcal{S}$
repeat
    $\Delta \leftarrow 0$
    Generate an episode using π: $\{S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T\}$
    $g \leftarrow 0$
    for $s = T-1, T-2, \ldots$ downto 0 **do**
        $g \leftarrow \gamma g + R_{t+1}$
        $N(s) \leftarrow N(s) + 1$
        $v_{\text{old}} \leftarrow v(s)$
        $v(s) \leftarrow v(s) + \frac{1}{N(s)}(g - v(s))$
        $\Delta \leftarrow \max[\Delta, |v_{\text{old}} - v(s)|]$
until $\Delta < \epsilon$
return v

which can be arbitrarily large. Hence, the variance of the estimate can also be arbitrarily large, and many samples are required in order to reduce such variance.

Exercise 5.1. For the Case Study 1 "random walk" considered in Sec. 4.3, find the value function when following the random policy using Monte-Carlo methods.

Off-policy MC prediction via importance sampling

Sometimes, the agent aims to predict the response of the environment to a hypothetical behavior, the target policy, that is different from the actual \behavior policies it is following. This problem of predicting the response to a target policy different from the behavior policy is commonly referred as off-policy learning. Off-policy learning has shown useful in several scenarios:

- when the target policy is risky and some actions could be costly (e.g., choosing the advertisement to show a user visiting a website or even dangerous (e.g., determining the medical treatment for a patient, so we

Algorithm 5.2 Every-visit MC policy evaluation for state-action value functions and for episodic MDPs.

Input: π, the policy to be evaluated.
Output: q, the estimate o fthe state-action value function q^π.

Initialize $q(s,a)$ arbitrarily (e.g., $q(s,a)=0$), for all $(s,a) \in \mathcal{S} \times \mathcal{A}$
Initialize $N(s,a)=0$, for all $(s,a) \in \{\mathcal{S} \times \mathcal{A}\}$
repeat
    $\Delta \leftarrow 0$
    Generate an episode using π: $\{S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T\}$
    $g \leftarrow 0$
    for $s = T-1, T-2, \ldots$ downto 0 **do**
        $g \leftarrow \gamma g + R_{t+1}$
        $N(s,a) \leftarrow N(s,a) + 1$
        $q_{\text{old}} \leftarrow q(s,a)$
        $q(s,a) \leftarrow q(s,a) + \frac{1}{N(s,a)}(g - q(s,a))$
        $\Delta \leftarrow \max[\Delta, |q_{\text{old}} - q(s,a)|]$
until $\Delta < \epsilon$
return q

can estimate the response to the target policy from more conservative behavior policies;

- if the agents have to learn from historical data (e.g., from human demonstration), where the learned target policies are different from the policy used for obtaining the data;
- in hierarchically structured environments, where it is convenient to predict the response of the environment for multiple policies (e.g., skills for achieving sub-goals at different time-spans) from a single data stream.
- when having many different predictions in parallel from a single stream of data is useful for extracting a feature representation of the environment;

Nevertheless, although clearly desirable for those situations, the variance of predictions derived from off-policy learning techniques can be unbounded, turning them unfeasible for certain applications. Learning off-policy predictions with bounded variance guarantees is an active area of research nowadays. In this section we just present some preliminary ideas.

The standard technique for off-policy learning is *importance sampling*, a general technique for estimating expected values under one distribution given

samples from another. The idea is to weight the samples according to the relative probability of their occurrence under the target and behavior policies. This weight is indeed called the *importance sampling ratio*, or *importance weight*. Suppose that the agent aims to estimate $v^{\pi}(s)$ or $q^{\pi}(s,a)$, but it only has access to a bag of samples obtained when following another policy $\mu(a \mid s)$, such that $\mu(a \mid s) \neq \pi(a \mid s)$. We require that every action that can be taken by π must be also taken, at least occasionally, by μ, i.e., we require that

$$\pi(a|s) > 0 \quad \text{implies} \quad \mu(a|s) \tag{5.16}$$

Let $\Omega(s,T)$ denote the set of all possible trajectories of $T+1$ state-action pairs starting from state $S_0 = s$, i.e.:

$$\Omega(s,T) \triangleq \{\{S_0, A_0, S_1, A_1, \ldots, S_T, A_T\} | S_0 = s\} \tag{5.17}$$

and let ω_T^{π} denote any such trajectory obtained by following target policy π:

$$\omega_T^{\pi} \triangleq \{S_0, A_0, S_1, A_1, \ldots, S_T, A_T\} \tag{5.18}$$

The probability distribution of ω_T^{π} is:

$$\mathbb{P}(\omega_T^{\pi}|S_0 = s) = \prod_{l=0}^{T} \pi(A_l \mid S_l) \mathcal{P}_{S_l S_{l+1}}^{A_l} \tag{5.19}$$

Then, the expected value of the return starting at state s can be expressed as follows:

$$\begin{aligned} v^{\pi}(s) &= \sum_{t=0}^{\infty} \sum_{w \in \Omega(s,\pi,\infty)} \mathbb{P}(\omega_t^{\pi}|S_0 = s) \gamma^t \mathcal{R}_{S_t}^{A_t} \\ &= \sum_{t=0}^{\infty} \sum_{w \in \Omega(s,\pi,\infty)} \prod_{l=0}^{t} \pi(A_l \mid S_l) \mathcal{P}_{S_l S_{l+1}}^{A_l} \gamma^t \mathcal{R}_{S_t}^{A_t} \end{aligned} \tag{5.20}$$

where we recall from (3.14) that $\mathcal{R}_s^a \triangleq \mathcal{R}(s,a) \triangleq \mathbb{E}[R_{t+1}|S_t = s, A_t = a]$.

Now, suppose that we have have obtained the sequence by following the behavior policy μ. In this case, we can obtain the expected return with respect to the behavior policy:

$$\begin{aligned} v^{\mu}(s) &= \sum_{t=0}^{\infty} \sum_{w \in \Omega(s,\pi,\infty)} \mathbb{P}(\omega_t^{\mu}|S_0 = s) \gamma^t \mathcal{R}_{S_t}^{A_t} \\ &= \sum_{t=0}^{\infty} \sum_{w \in \Omega(s,\pi,\infty)} \prod_{l=0}^{t} \mu(A_l \mid S_l) \mathcal{P}_{S_l S_{l+1}}^{A_l} \gamma^t \mathcal{R}_{S_t}^{A_t} \end{aligned} \tag{5.21}$$

The question is how can we compute v^{π} from data generated by following μ? In order to answer this question, consider the relative probability of the trajectory under the target and behavior policies, which is known as the importance sampling ratio:

$$\zeta_T = \frac{\prod_{l=0}^{t} \pi\left(A_l \mid S_l\right) \mathcal{P}_{S_l S_{l+1}}^{A_l}}{\prod_{l=0}^{t} \mu\left(A_l \mid S_l\right) \mathcal{P}_{S_l S_{l+1}}^{A_l}} = \frac{\prod_{l=0}^{t} \pi\left(A_l \mid S_l\right)}{\prod_{l=0}^{t} \mu\left(A_l \mid S_l\right)} \tag{5.22}$$

where, although the trajectory probabilities depend on the unknown transition probabilities, $\mathcal{P}_{S_l S_{l+1}}^{A_l}$, these terms cancel off and the importance sampling ratio ends up depending on the two policies alone, and not on the MDP. Introduce the importance-sampling corrected return:

$$\begin{aligned} G^{\pi/\mu}\left(S_t\right) &\triangleq \frac{\pi\left(A_t \mid S_t\right)}{\mu\left(A_t \mid S_t\right)} R_{t+1} + \frac{\pi\left(A_t \mid S_t\right) \pi\left(A_{t+1} \mid S_{t+1}\right)}{\mu\left(A_t \mid S_t\right) \mu\left(A_{t+1} \mid S_{t+1}\right)} \gamma R_{t+2} + \ldots \\ &= \sum_{k=0}^{\infty} \frac{\prod_{l=0}^{t} \pi\left(A_l \mid S_l\right)}{\prod_{l=0}^{t} \mu\left(A_l \mid S_l\right)} \gamma^k R_{t+k+1} \\ &= \sum_{k=0}^{\infty} \zeta_k \gamma^k R_{t+k+1} \end{aligned} \tag{5.23}$$

Suppose that we are interested the expected return when starting at any initial state S_0. Hence, we have the following relation:

$$\begin{aligned} \mathbb{E}_{\mu,\mathcal{P}}\left[G^{\pi/\mu}\left(S_0\right)\right] &= \sum_{t=0}^{\infty} \sum_{w \in \Omega(s,\pi,\infty)} \prod_{l=0}^{t} \mu\left(A_l \mid S_l\right) \mathcal{P}_{S_l S_{l+1}}^{A_l} \frac{\prod_{l=0}^{t} \pi\left(A_l \mid S_l\right)}{\prod_{l=0}^{t} \mu\left(A_l \mid S_l\right)} \gamma^t \mathcal{R}_{S_t}^{A_t} \\ &= \sum_{t=0}^{\infty} \sum_{w \in \Omega(s,\pi,\infty)} \prod_{l=0}^{t} \pi\left(A_l \mid S_l\right) \mathcal{P}_{S_l S_{l+1}}^{A_l} \gamma^t \mathcal{R}_{S_t}^{A_t} \\ &= v^{\pi}\left(S_0\right) \end{aligned} \tag{5.24}$$

We remark that the if the terms happen to satisfy $\mu\left(A_t \mid S_t\right) < \pi\left(A_t \mid S_t\right)$, then the variance of the estimate becomes high. In the limit, when T tends to infinity, the variance could become unbounded (i.e., infinity) too. We remark that the estimate remains unbiased though.

Let $g_l^{\pi/\mu}(s)$ denote the l-th realization of $G_l^{\pi/\mu}(s)$. There are many ways to implement off-policy strategies. For instance, for the MC policy evaluation with iterative updates(5.12), we can simply replace $g_l^{\pi}(s)$ with $g_l^{\pi/\mu}(s)$. Hence, we obtain the following off-policy recursion for any state $s \in \mathcal{S}$:

$$v_{l+1}(s) = v_l(s) + \tfrac{1}{N(s)} \left(g_l^{\pi/\mu}(s) - v_l(s)\right) \tag{5.25}$$

We know from (5.24) that (5.25) converges to the true value $v^{\pi}(s)$.

Exercise 5.2. For the Case Study 2 "grid-world environment" considered in Sec. 4.4, find the value function when following the random policy, π, using MC methods. Then, calculate the value function v^{μ} assuming that the model is known (i.e., using DP policy evaluation) for policy $\mu(\text{up}) = \mu(\text{down}) = 1/3$ and $\mu(\text{right}) = \mu(\text{left}) = 1/6$. Finally, apply off-policy MC to learn the value function v^{μ} from the data generated when following policy π.

Figure 5.3: The grid-world problem. Policy evaluation for the random policy

---

5.1.2 Temporal Difference (TD) prediction

Similar to MC, TD methods also learn from interaction with the environment. The main difference with MC is that MC uses the actual return to update the value estimate, while TD estimates also the return as the sum of the instantaneous reward and the estimate of the value function in the future reward:

$$\text{MC}: \quad v(S_t) \leftarrow v(S_t) + \alpha_t \left(G^{\pi}(S_t) - v(S_t)\right) \tag{5.26}$$

$$\text{TD}: \quad v(S_t) \leftarrow v(S_t) + \alpha_t \left(R_{t+1} + \gamma v(S_{t+1}) - v(S_t)\right) \tag{5.27}$$

where α_t denotes some step-size of a sequence satisfying (5.14). In other words, note that the return can be rewritten as:

$$G^{\pi}(S_t) = R_{t+1} + \gamma v^{\pi}(S_{t+1}) \tag{5.28}$$

TD estimates v^{π} in (5.28) by approximating $G^{\pi}(S_t)$ with another, $\widehat{G}^{\pi}(S_t)$, given by:

$$\widehat{G}^{\pi}(S_t) \triangleq R_{t+1} + \gamma v(S_{t+1}) \approx G^{\pi}(S_t) \tag{5.29}$$

In other words, TD replaces the actual v^π with its current estimate v. This idea of updating an estimate with another estimate is known as *bootstrapping* in the RL literature and is illustrated in Figure 5.4.

Typically, we use the following terminology:

- The bootstrapping return, $\widehat{G}^\pi(S_t)$, is known as *TD target.*
- The so named *TD error* is defined as

$$\delta_t \triangleq R_{t+1} + \gamma v(S_{t+1}) - v(S_t) \tag{5.30}$$

$$V(S_t) \leftarrow V(S_t) + \alpha\left(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right)$$

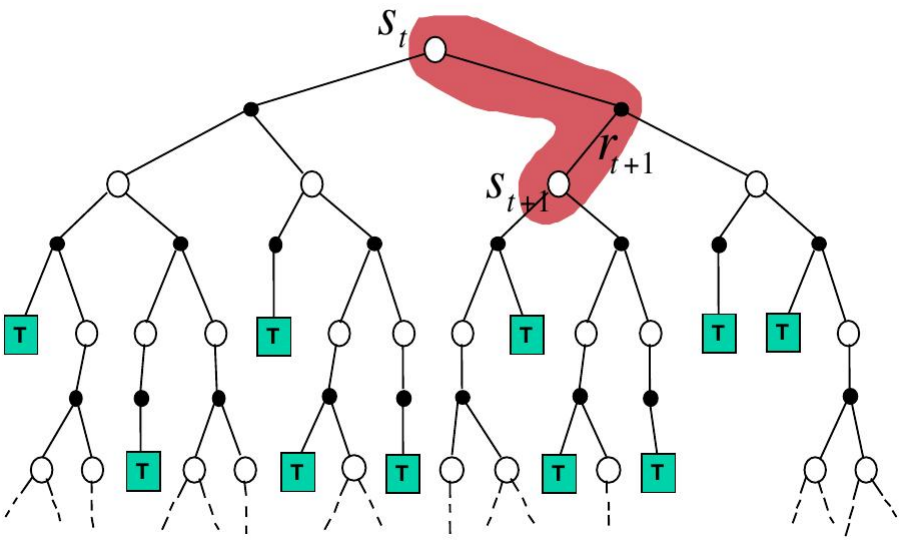


Figure 5.4: Time Difference backup

We can obtain a TD recursion for state-action value functions following the same reasoning as for state value functions. First, let the state action return be defined as:

$$G^\pi(S_t, A_t) = R_{t+1} + \gamma q^\pi(S_{t+1}, A_{t+1}) \tag{5.31}$$

where $A_{t+1} \sim \pi$. TD approximates $G^\pi(S_t, A_t)$ with another term given by:

$$\widehat{G}^\pi(S_t, A_t) \triangleq R_{t+1} + \gamma q(S_{t+1}, A_{t+1}), \quad A_{t+1} \sim \pi \tag{5.32}$$

such that

$$\widehat{G}^\pi(S_t, A_t) \approx G^\pi(S_t, A_t)$$

Thus, we obtain:

$$q\left(S_t, A_t\right) \leftarrow q\left(S_t, , A_t\right) + \alpha_t \left(R_{t+1} + \gamma q\left(S_{t+1}, A_{t+1}\right) - q\left(S_t, , A_t\right)\right) \quad (5.33)$$

where, again, α_t is some step-size of a sequence satisfying (5.14).

The pseudocode for every-visit MC for predicting state and state-action value functions, for episodic MDPs, with iterative updates is given by Algorithms 5.3 and 5.4, respectively.

Algorithm 5.3 TD for state-value functions.

Input: π, the policy to be evaluated.
Output: v, the estimate of the value function v^{π}.

Initialize $v(s)$ arbitrarily (e.g., $v(s) = 0$), for all $s \in \mathcal{S}$
repeat(for each episode)
    Initialize s
    repeat(for each step in the episode)
        Choose action $a \sim \pi(\cdot|s)$
        Take action a and observe r, s'
        $v(s) \leftarrow v(s) + \alpha(r + \gamma v(s') - v(s))$
        $s \leftarrow s'$
    until s is terminal
until we cannot run more episodes
return v

It is important to remark that TD predictions are usually of much less variance than MC predictions, at the cost of introducing some bias. The reason for the reduced variance is that TD only approximates the expected value of the current reward, given the current state-action pair, with distribution over the real line; while we remind that MC approximates the expected return, with distribution defined over the Cartesian product of all the rewards in the sequence. In addition, TD usually converges faster than MC.

Exercise 5.3. For the Case Study 1 considered in Sec. 4.3, find the value function when following the random policy using MC and TD methods for different values of the step-size parameter β, used for both decreasing ($\alpha_l = \beta/l$) and constant ($\alpha_l = \beta$) step-sizes. You have to reproduce the results and those presented in Figure 4.3.

Algorithm 5.4 TD for state-action value functions.

Input: π, the policy to be evaluated.
Output: q, the estimate o fthe state-action value function q^{π}.

Initialize $q(s,a)$ arbitrarily (e.g., $q(s,a)=0$), for all $(s,a) \in \mathcal{S} \times \mathcal{A}$
repeat(for each episode)
    Initialize s, a
    repeat(for each step in the episode)
        Take action a and observe r, s'
        Choose action $a' \sim \pi(\cdot|s)$
        $q(s,a) \leftarrow q(s,a) + \alpha(r + \gamma q(s',a') - q(s,a))$
        $s \leftarrow s'$
        $a \leftarrow a'$
    until s is terminal
until we cannot run more episodes
return q

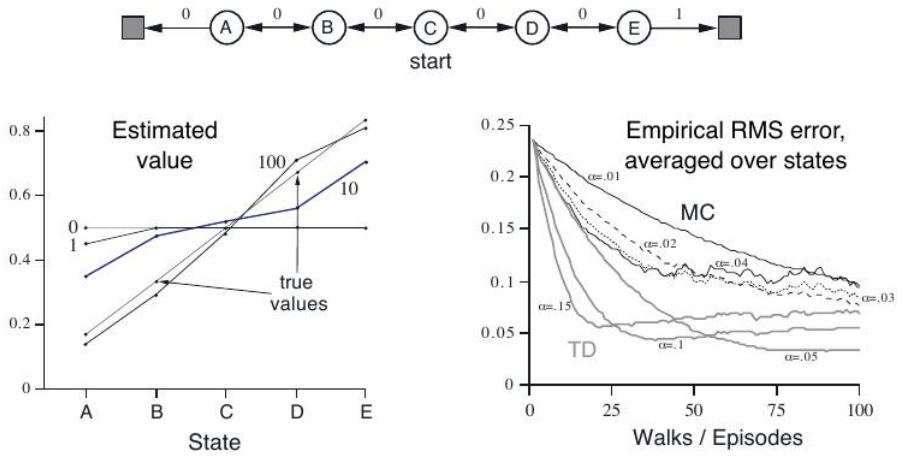


Figure 5.5: The Random-Walk problem. Comparison of MC / TD

Importance sampling for Off-policy TD

In a similar way as we described for MC methods, it is possible to estimate the value function for policy π from data generated from the behavior policy μ. Recall from (5.27) and (5.29) that TD approximates the return with the bootstrapping return, which only includes one reward sample, rather than the whole reward sequence. Hence, in order to learn off-policy, we have

to correct the bootstrapping return with the importance sampling weight. Suppose that we obtain sample R_{t+1} after being at state S_t and taking action $A_{t+1} \sim \mu(\cdot|S_t)$. The expected bootstrapping return for μ is then given by:

$$\mathbb{E}_{\mu,\mathcal{P}}\left[\widehat{G}^{\mu}(S_t)\right] = \mu(A_t \mid S_t)\,\mathcal{P}^{A_t}_{S_tS_{t+1}}\left(\mathcal{R}^{A_t}_{S_t} + \gamma v(S_{t+1})\right) \tag{5.34}$$

Introduce the importance-weighted bootstrapping return:

$$\widehat{G}^{\pi/\mu}(S_t) \triangleq \frac{\pi(A_t \mid S_t)}{\mu(A_t \mid S_t)}\left(R_{t+1} + \gamma v(S_{t+1})\right) \tag{5.35}$$

Then, we have

$$\begin{aligned}\mathbb{E}_{\mu,\mathcal{P}}\left[\widehat{G}^{\pi/\mu}(S_t)\right] &= \mu(A_t \mid S_t)\,\mathcal{P}^{A_t}_{S_tS_{t+1}}\frac{\pi(A_t \mid S_t)}{\mu(A_t \mid S_t)}\left(\mathcal{R}^{A_t}_{S_t} + \gamma v(S_{t+1})\right)\\ &= \pi(A_t \mid S_t)\,\mathcal{P}^{A_t}_{S_tS_{t+1}}\left(\mathcal{R}^{A_t}_{S_t} + \gamma v(S_{t+1})\right)\\ &= \mathbb{E}_{\pi,\mathcal{P}}\left[\widehat{G}^{\pi}(S_t)\right]\end{aligned} \tag{5.36}$$

In words, the expected target of the importance-weighted-bootstrapping return for TD targets generated from μ equals the expected value of the bootstrapping-return with respect to π. Hence, we can use the following off-policy TD recursions for estimating the state value function of π:

$$v(S_t) \leftarrow v(S_t) + \alpha_t\left(\frac{\pi(A_t \mid S_t)}{\mu(A_t \mid S_t)}\left(R_{t+1} + \gamma v(S_{t+1})\right) - v(S_t)\right) \tag{5.37}$$

where α_t is some step-size of a sequence satisfying (5.14).

Off-policy TD has only one importance weight (as opposed to off-policy MC that requires a product of infinite importance weights). Hence, the variance increment due to importance sampling is bounded.

Interestingly, off-policy TD learning for action values $q(s,a)$ is free from importance weights. The reason is that the first action of the sequence is already given by $A_t = a$, rather than drawn from any (either target or behavior) policy. From (5.32), we can see that the only random variable in the bootstrapping return is A_{t+1}, the other term is an estimate $q(S_{t+1}, A_{t+1})$. Then, we have:

we have:

$$\mathbb{E}_{\mu,\mathcal{P}}\left[\widehat{G}^{\mu}(S_t, A_t)\right] = \mathcal{R}^{A_t}_{S_t} + \gamma q(S_{t+1}, A_{t+1}), \quad A_{t+1} \sim \mu \tag{5.38}$$

In order to compute the expected state-action bootstrapping return with respect to π, we only have to ensure that $A_{t+1} \sim \pi$, with no importance weighting:

$$\mathbb{E}_{\pi,\mathcal{P}}\left[\widehat{G}^{\mu}\left(S_t, A_t\right)\right] = \mathcal{R}_{S_t}^{A_t} + \gamma q\left(S_{t+1}, A_{t+1}\right), \quad A_{t+1} \sim \pi$$

Exercise 5.4. For the Case Study 2 "grid world" problem considered in Sec. 4.4, implement TD for state-action value functions and compare the results with the DP solution that assumed knowledge of the model.

5.2 TD(λ) algorithms

We have seen that MC predictions provide an unbiased estimate with high variance, since they approximate a high dimensional distribution—corresponding to the whole sequence—with samples. On the other hand, we have seen that TD predictions are biased, since they only have to approximate the instantaneous reward with samples, but they are biased, since the return is approximated with the TD target. In this section, we show a family of algorithms, denoted TD(λ) that, depending on the value of $\lambda \in [0, 1]$, can trade-off the variance of MC and the bias of TD. Indeed, we will see that TD(λ) includes MC and TD as particular cases, being TD(0) = TD and TD(1)=MC. In other words, TD and MC are the two ends of the spectrum of algorithms defined by TD(λ). Also, we can see λ as a parameter that indicates the amount of bootstrapping of the algorithm.

We introduce TD(λ) from two points of view: *forward* and *backward*. The *forward view* is more theoretical and sets the bridge between MC and TD. Nevertheless, the forward view is not directly implementable. The *backward* view presents *eligibility traces* as the key technique for implementing TD(λ) (it can be applied to any other bootstrapping algorithm though, like SARSA or Q-learning), and consists in assigning credit to which states or state-action pairs are responsible for the TD error.

5.2.1 n-step returns

The standard TD algorithm is sometimes called *one-step* TD method because the bootstrapping return, $\widehat{G}^{\pi}\left(S_t\right)$ given by 5.29, uses only the reward of just

one state-transition (R_{t+1}) and the rest of the return is approximated with the current value estimate ($\gamma v(S_{t+1})$). This can be easily generalized to approximating the return with the sum of the reward for the first n steps ($R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{n-1} R_{t+n}$) and then bootstrapping with the current estimate with much less weight ($\gamma^n v(S_{t+n})$), yielding the so called *n-step* TD methods. Let $G^{\pi}_{(n)}(S_t)$ denote the n-step return given by:

$$G^{\pi}_{(n)}(S_t) = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \gamma^{n-1} R_{t+n} + \gamma^n v^{\pi}(S_{t+n}) \quad (5.39)$$

And let $\widehat{G}^{\pi}_{(n)}(S_t)$ denote the bootstrapping approximation of $G^{\pi}_{(n)}(S_t)$:

$$\widehat{G}^{\pi}_{(n)}(S_t) = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \gamma^{n-1} R_{t+n} + \gamma^n v(S_{t+n}) \quad (5.40)$$

where we only have replaced the actual value v^{π} with the estimate v. Note that the $\widehat{G}^{\pi}(S_t)$ used in (5.27) for the TD algorithm is actually $\widehat{G}^{\pi}_{(1)}(S_t)$ with the new n-step notation. Hence, the idea of n-step TD methods is simply to use $\widehat{G}^{\pi}_{(n)}(S_t)$ as TD error:

$$n-\text{step TD}: \quad v(S_t) \leftarrow v(S_t) + \alpha \left(\sum_{i=1}^{n} R_{t+i} + \gamma^n v(S_{t+n}) - v(S_t) \right) \quad (5.41)$$

It is clear that this new family of methods have TD in one end ($n = 1$) and MC in the other ($n = T$). And the benefit is that we can tradeoff bias and variance with the value of $n = 1, \ldots, T$. In particular, when $n = 1$, we have lowest variance but highest bias; and highest variance and zero bias when $n = T$. Figure (5.6) illustrates this trade-off between bias and variance.

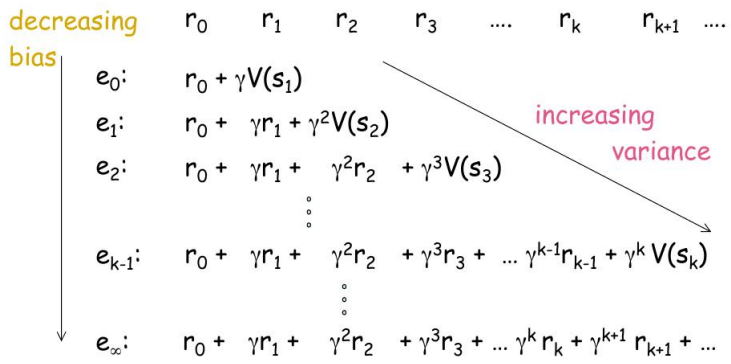


Figure 5.6: Bias vs. variance trade-off for n-steps returns.

5.2.2 Forward view of TD(λ)

The problem with the n-step TD methods discussed above is that they require the reward n steps ahead of the current state transition. This is not implementable in a standard online setting because such future rewards are unknown. One way to overcome this issue with TD(λ) and the eligibility traces. Before discussing eligibility traces, we introduce another more general return.

Instead of simply using the n-step return, we could consider any weighted average of n-step bootstrapping returns for multiple values of n. In particular, consider the following way of averaging n-step returns:

$$\widehat{G}_{\lambda}^{\pi}\left(S_t\right) = (1-\lambda)\sum_{n=1}^{T}\lambda^{n-1}\widehat{G}_{(n)}^{\pi}\left(S_t\right) \tag{5.42}$$

where $0 \leq \lambda \leq 1$ is a parameter. Figure illustrates the weight assigned to different $n-$step returns.

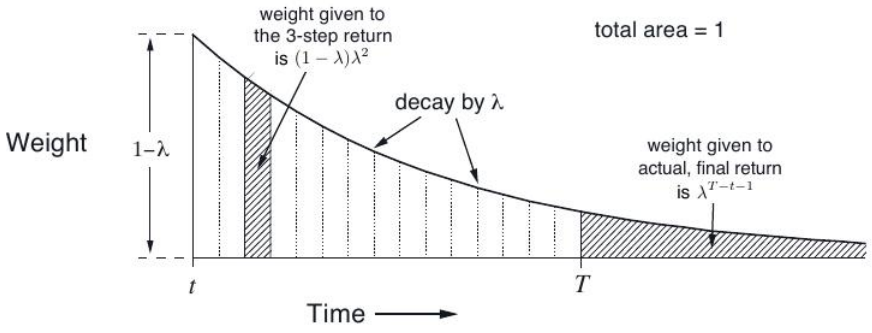


Figure 5.7: Weighting given in the $\lambda-$return to each of the $n-$steps returns

The idea of TD(λ) consists in using $G_{\lambda}^{\pi}\left(S_t\right)$ as bootstrapping return in (5.27):

$$\text{forward view TD}(\lambda): \qquad v\left(S_t\right) \leftarrow v\left(S_t\right) + \alpha_t\left(\widehat{G}_{\lambda}^{\pi}\left(S_t\right) - v\left(S_t\right)\right) \tag{5.43}$$

Interestingly, it can be shown that TD(λ) is equivalent to TD when $\lambda = 0$ and to MC when $\lambda = 1$. However, we still have the same problem of n-step TD, since $\widehat{G}_{\lambda}^{\pi}$ requires future rewards that are not available. This is the reason (5.43) is known as forward view. In the following section we introduce an implementable backward view, for which the update of the value estimate at time $t+1$ depends only on rewards that are known at time $t+1$. Actually,

the real motivation for the λ-way of averaging $n-$step returns is that there it can be implemented with the backward view that we present next.

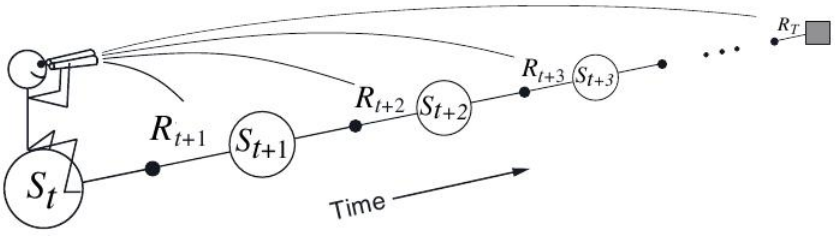


Figure 5.8: The forward or theoretical view of TD(λ) and n-step returns.

Exercise 5.5. Prove that that TD(λ) is equivalent to TD, when $\lambda = 0$; and it is equivalent to MC, when $\lambda = 1$.

5.2.3 Backward view of TD(λ)

The backward view presented here provides a *causal*, incremental mechanism for approximating the forward view online (i.e., at every iteration), and for achieving it exactly in the offline case.

The idea consists in introducing an additional memory variable associated with each state, known as *eligibility trace*. The eligibility trace for state s at time t is a nonnegative random variable denoted $E_t(s) \in \mathbb{R}^+$ that assigns temporal credit to state s. In particular, if state s has been visited often and recently, then $E_t(s)$ will be high, meaning that we assign much credit to s for the actual return that we may obtain from now on. On the other hand, if s has been rarely, long time ago, or not visited at all in the current sequence, then we assign very little credit to s for the rewards that we may obtain in the future. There are different forms of implementing this temporal-credit idea. For instance, *accumulating traces* are updated incrementally as follows:

$$E_t(s) \triangleq \begin{cases} \gamma\lambda E_{t-1}(s) & \text{if } s \neq S_t \\ \gamma\lambda E_{t-1}(s) + 1 & \text{if } s = S_t \end{cases}, \quad \forall s \in \mathcal{S} \tag{5.44}$$

where γ is the discount rate and $0 \leq \lambda \leq 1$ is the parameter introduced in the previous section. Note that the accumulating trace decays by $\gamma\lambda$ for all

non-visited states and increases only for the currently visited state. Thus, we refer to λ as the *trace-decay* parameter henceforth. Accumulating traces take the name because they accumulate each time the state is visited, then fades away gradually when the state is not visited, as illustrated bellow.

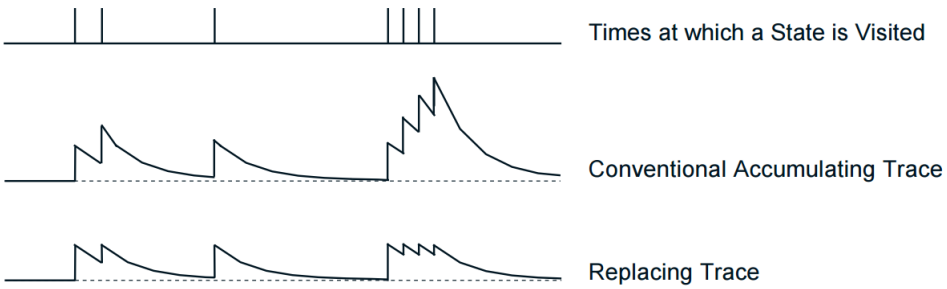


Figure 5.9: Temporal credit assignment for accumulating and replacing eligibility traces.

An alternative to accumulating traces are the so named replacing traces:

$$E_t(s) \triangleq \begin{cases} \gamma\lambda E_{t-1}(s) & \text{if } s \neq S_t \\ 1 & \text{if } s = S_t \end{cases}, \quad \forall s \in \mathcal{S} \tag{5.45}$$

Replacing traces do not increase with repeated visits to a state but are reset to a constant value each time the state is visited. The trace due to a revisit does not increase the existing trace, it simply replaces it. Numerical experiments have shown that replacing traces can result in faster and more reliable learning than accumulating traces.

Eligibility traces keep a simple record of which states have been visited "recently", where recently is defined in terms of $\gamma\lambda$. The traces indicate the degree to which each state should be influenced by the current response of the environment for the learning process:

$$\text{backward view TD}(\lambda): \qquad v(S_t) \leftarrow v(S_t) + \alpha_t \delta_t E_t(s) \tag{5.46}$$

where δ_t is the TD error defined by (5.30).

It has been shown that the forward view (with λ-return) given by (5.43) and the backward view (with eligibility traces) given by (5.46)are equivalent under some circumstances.

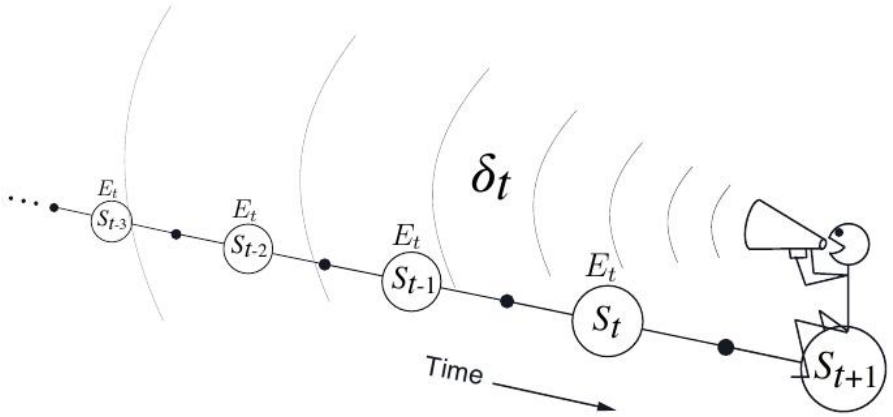


Figure 5.10: The backward or mechanism view

5.3 Control.

In this section we extend previous MC and TD methods from making prediction to solving the control problem (i.e., finding the optimal value function) by interaction with an unknown MDP, without any prior knowledge of the environment. The idea is to build approximate DP methods where we replace the value function with a high-variance, unbiased MC estimate or with a low-variance, biased bootstrapping TD estimate. We start with the illustrative extension of MC to control, then we present two TD based control algorithms, named Q-learning and SARSA.

5.3.1 Monte-Carlo control

In order to extend MC for solving the control problem, we can develop a policy iteration algorithm where, at every iteration, the greedy policy is taken with respect to the current value estimate. Recall the policy iteration scheme from (4.30)–(4.31), which we repeat here for convenience for the state-action value function:

$$\pi_0(s) \overset{E}{\to} q^{\pi_0}(s,a) \overset{I}{\to} \pi_1(s) \overset{E}{\to} q^{\pi_1}(s,a) \overset{E}{\to} \cdots \overset{I}{\to} \pi^*(s) \overset{E}{\to} q^*(s,a) \quad (5.47)$$

The idea consists in performing the policy evaluation steps approximately, by using MC methods. The policy improvement step is then obtained by taking the greedy policy with respect to the MC estimate of the value function.

The main technical issue is that in order to improve the policy from state-action value estimates, we need to have value estimates of all possible state-action pairs. In other words, if we want to find the action that maximizes $q(s,a)$, we need estimates of the state-action value for all possible state-action pairs: $q\left(s,a_1\right), q\left(s,a_2\right), \ldots, q\left(s,a_{|\mathcal{A}|}\right)$. Thus, MC methods require to force some form of *exploration* in order to visit state-action pairs that, otherwise, would not be visited. Actually, we need to establish a exploration vs. exploitation tradeoff in order to visit state-action pairs with higher value estimate more often, while allowing exploration of state-action pairs with lower value estimate. Standard forms of ensuring exploration in MC methods are using an exploratory behavior policy (e.g., ϵ-greedy) or evaluating the return for every possible state-action pair (the latter is known as *exploring-starts*).

Numerical experiments have shown that by guaranteeing that every state-action pair is visited (infinitely) often, then MC control can asymptotically converge to the optimal policy. However, the main handicap of this MC control scheme is that MC estimates are usually high variance and, therefore, are rarely used in practice.

5.3.2 TD control

In the last subsection, we showed the need of exploring all state-action pairs in order to have accurate value estimates and, hence, be able to properly choose the actions that yield the highest value estimates. The need for establishing an exploration vs. exploitation tradeoff is of main importance in RL. The key idea to keep in mind is that we are estimating the distributions of the return conditioned on the state-action pairs from samples. Therefore, as we have discussed in Chapter 2, we want to exploit the actions that have resulted highest value estimates so far (i.e., we can think on the estimate of the value function as the sample mean of the distribution of the return) but, at the same time, we need to explore other actions just in case they could provide even higher rewards.

Designing behavior policies that provide an optimal exploration vs. exploitation tradeoff is an active area of research, but will not be covered in this introductory course. Instead, we present two *heuristic* behavior policies that have shown to provide decent results in practice, they are the ϵ-greedy and the Boltzmann policies. The ϵ-greedy has been already discussed, but we remind it here for convenience:

$$\pi_\epsilon\left(a|s\right) \triangleq \begin{cases} 1-\epsilon+\epsilon/|\mathcal{A}|, & a \in \arg\max_{a' \in \mathcal{A}} \; q\left(s,a'\right) \\ \epsilon/|\mathcal{A}|, & a \in \mathcal{A} \end{cases} \tag{5.48}$$

where $\epsilon \in (0,1)$ is the exploration parameter, such that the higher ϵ, the more exploration and less exploitation. The Boltzmann policy assigns probabilities based on their actual merit as indicated by their value estimates:

$$\pi_{\text{Boltzmann}}(a|s) \triangleq \frac{e^{q(s,a)/\tau}}{\sum_{a' \in \mathcal{A}} e^{q(s,a')/\tau}} \quad , \quad \forall a \in \mathcal{A} \tag{5.49}$$

where the temperature $\tau > 0$ controls the amount of exploration, such that when $\tau \to 0$, we obtain the greedy action, and when $\tau \to \infty$, the action is drawn uniformly from $\mathcal{A}$.

We proceed to introduce two very well known RL algorithms, namely SARSA and Q-learning, which are TD approximations of PI and VI planning algorithms, respectively.

SARSA

SARSA is an approximation of the PI algorithm (see Section (4.2.2)). Since TD learns from sample tuples of the form $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$, the name SARSA comes from joining the initial of every element in the data tuples employed by the algorithm: state, action, reward, (next) state, (next) action. SARSA is based on three key ideas.

The first idea consists in replacing q^π with its bootstrapping estimate for the current policy, π_ϵ, so that the state-action value estimate update becomes:

$$q(s_t, a_t) \leftarrow q(s_t, a_t) + \alpha_t \left(R_{t+1} + \gamma q(s_{t+1}, a_{t+1}) - q(s_t, a_t)\right) \tag{5.50}$$

where $a_{t+1} \sim \pi_\epsilon(\cdot|s_{t+1})$ is the action taken from the current policy, and α_t is the step-size of some sequence satisfying (5.14). Figure (5.11) illustrates the backups for TD iterates (5.50).

The second idea is to perform the policy improvement step by taking the ϵ-greedy policy with respect to the current value estimate, q. Note that the TD target is the bootstrapping return with respect to the ϵ-greedy policy, $\widehat{G}^{\pi_\epsilon}(S_t)$. Hence, since the behavior policy used to obtain the data tuples and the target policy are the same (i.e., the current ϵ-greedy denoted π_ϵ), we say that SARSA is an on-policy learning algorithm.

The third idea is to perform the policy improvement step without having waited for the convergence of the bootstrapping value estimate (5.50). Instead, the most common implementation of SARSA consists in perform one single TD iterate (5.50) and then perform policy improvement. This idea of considering that just one TD iterate is good enough for performing policy iteration is sometimes called *fully optimistic*.

The scheme of SARSA iterations is illustrated in Figure (5.12). And the pseudocode for SARSA is given by Algorithm (5.5).

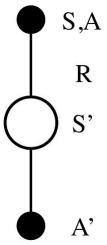


Figure 5.11: Backup diagram for SARSA algorithm

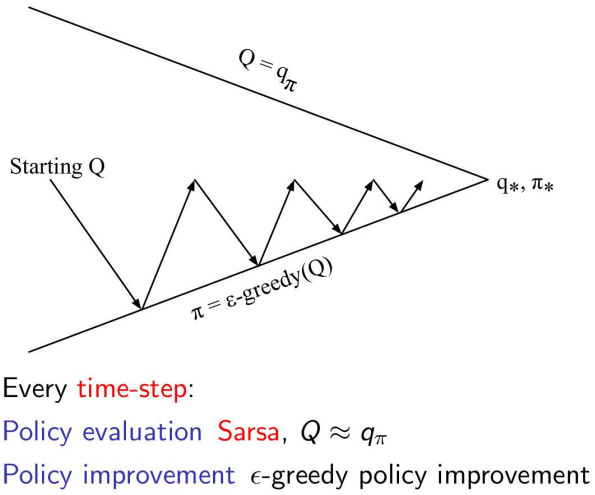


Figure 5.12: SARSA algorithm for on policy control

Q-learning

The Q-learning algorithm is an approximation to VI algorithm (see Algorithm 4.6) that is based on two simple ideas.

The first idea consists in replacing q^π with its bootstrapping estimate in the optimal Bellman operator, so that the state-action value estimate update becomes:

$$q(s_t, a_t) \leftarrow q(s_t, a_t) + \alpha_t \left(R_{t+1} + \gamma \max_{a \in \mathcal{A}} q(s_{t+1}, a) - q(s_t, a_t) \right) \tag{5.51}$$

Algorithm 5.5 SARSA with ϵ-greedy target policy for episodic MDP.

Input: Parameter ϵ and step-size sequence α.
Output: π, the approximate optimal policy $\pi^\star$.

Initialize $q(s,a)$ arbitrarily for all $(s,a) \in \mathcal{S} \times \mathcal{A}$
Initialize $q(s,\cdot) = 0$ for all terminal states
repeat(for each episode)
    Initialize s
    Choose action $a \sim \pi_\epsilon(\cdot|s)$ using ϵ-greedy policy (5.48) from current q
    repeat(for each step in the episode)
        Take action a and observe r, s'
        Choose action $a' \sim \pi_\epsilon(\cdot|s')$ using (5.48) from current q
        $q(s,a) \leftarrow q(s,a) + \alpha(r + \gamma q(s',a') - q(s,a))$
        $s \leftarrow s'$, $a \leftarrow a'$
    until s is terminal
until we cannot run more episodes
for all $s \in \mathcal{S}$ **do**
    $\pi(s) \leftarrow \arg\max_{a \in \mathcal{A}} \; q(s,a)$
return π

where, as usual, α_t is the step-size of some sequence satisfying (5.14). This is different from SARSA in that here we are approximating the *optimal* Bellman operator, rather than the Bellman operator for the current policy. From a TD perspective, note that the term $R_{t+1} + \gamma \max_{a \in \mathcal{A}} \; q\left(s_{t+1}, a\right)$ is the bootstrapping approximation of the return obtained with the greedy policy. This is a key point of Q-learning: the value function is estimated for the greedy policy. In order to see this key point more clearly, let π_{greedy} denote the greedy policy, such that

$$\pi_{\text{greedy}}\left(a|s\right) = \begin{cases} 1, & a \in \arg\max_{a' \in \mathcal{A}} \; q\left(s,a'\right) \\ 0, & \text{otherwise} \end{cases} \tag{5.52}$$

Then, according with (5.31), the return obtained with the greedy policy is given by:

$$G^{\pi_{\text{greedy}}}\left(s_t\right) = R_{t+1} + \gamma \max_{a \in \mathcal{A}} \; q^{\pi_{\text{greedy}}}\left(s_{t+1}, a\right) \tag{5.53}$$

Thus, Q-learning uses the following approximation:

$$\widehat{G}^{\pi_{\text{greedy}}}\left(s_t\right) = R_{t+1} + \gamma \max_{a \in \mathcal{A}} \; q\left(s_{t+1}, a\right) \approx G^{\pi_{\text{greedy}}}\left(s_t\right) \tag{5.54}$$

Figure 5.13 illustrates the backup diagram for iteration (5.51).

The second idea consists in using an *exploratory behavior* policy such as ϵ-greedy given by (5.48) to ensure that all state-action pairs are evaluated. Since the target policy is greedy, it is different from the ϵ-greedy behavior policy. Therefore, we say that Q-learning is an *off-policy* algorithm.

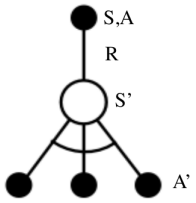


Figure 5.13: Backup diagram for the Q-learning algorithm.

The pseudocode for Q-learning is shown in Algorithm.

Algorithm 5.6 Q-learning with ϵ-greedy behavior policy for episodic MDP.

Input: Parameter ϵ and step-size sequence α.
Output: π, the approximate optimal policy $\pi^\star$.

Initialize $q(s,a)$ arbitrarily for all $(s,a) \in \mathcal{S} \times \mathcal{A}$
Initialize $q(s,\cdot) = 0$ for all terminal states
repeat(for each episode)
    Initialize s
    repeat(for each step in the episode)
        Choose action $a \sim \pi_\epsilon(\cdot|s)$ using (5.48) from current q
        Take action a and observe r, s'
        $q(s,a) \leftarrow q(s,a) + \alpha(r + \gamma \max_{a' \in \mathcal{A}} q(s',a') - q(s,a))$
        $s \leftarrow s'$
    until s is terminal
until we cannot run more episodes
for all $s \in \mathcal{S}$ **do**
    $\pi(s) \leftarrow \arg\max_{a \in \mathcal{A}} q(s,a)$
return π

Exercise 5.6. For the Case Study 2 "grid world" problem considered in Sec. 4.4, implement both SARSA and Q-learning and compare their output with the optimal policy given by the VI/PI algorithms that assumed knowledge of the model.

Exercise 5.7. The Cliff-walking problem. Let us consider a particular grid-world problem, named the *cliff-walking problem,* that is appropriate for comparing SARSA and Q-learning. The cliff-waling problem is an undiscounted (i.e., $\gamma = 1$) episodic task, with one single starting state and one single terminal state, denoted S and G, respectively, in Figure (5.14). The action set consists of the usual moving directions for grid-worlds problems: $\mathcal{A} = \{\text{up}, \text{down}, \text{right}, \text{left}\}$. The reward is -1 for all state transitions, except those into the region marked as *cliff*, such that stepping into this region is penalized with a highly negative reward of -100. In addition, transiting to the goal terminal state provides zero reward. When the agent gets to the goal state, the episode ends, and we can start a new episode.

Compare the solutions provided by SARSA and Q-learning for ε-greedy action selection with $\varepsilon = 0.1$. Represent the reward per episode after 500 episodes.

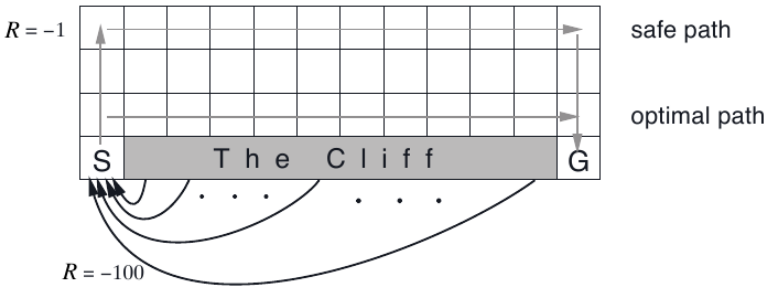


Figure 5.14: The cliff-walking problem

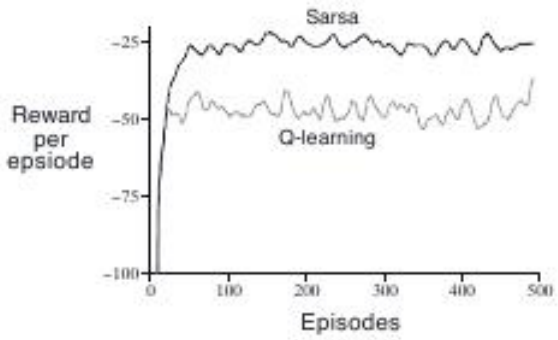


Figure 5.15: Solution to the Cliff-walking problem

Part III

Reinforcement Learning in Large and continuous spaces

Chapter 6

Linear approximation

6.1 Motivation

In previous chapters we have assumed a very convenient model of moderate size discretized spaces of actions and states. In such scenario, value functions can be represented by *lookup* tables where every state s has an entry $v(s)$ or every state-action pair s, a has an entry $q(s, a)$. This assumption simplifies implementation issues but it is not very realistic in many real problems. Consider for instance the Backgammon game with 10^{20} states or the Computer Go with 10^{170} states. Also, consider the case where we intend to control a helicopter with continuous state space. Is RL capable of providing a satisfactory solution to these problems?. The answer is *yes* but requires some sort of approximation since exact solutions can not be found in general.

The algorithms of previous chapters can no longer be applied in their original form. Instead, we will introduce approximate versions of a number of model-free algorithms. In particular, we will consider that the rather than having access to the environment state, the agent has access to a set of features of the state. In this chapter, we only consider linear approximations, where the value function for some state is approximated as a linear combination of the feature vector for such state. Nonlinear approximations with neural networks will be delayed until the next chapter.

We remark that there exists non-parametric approximations (e.g., kernel methods) that can be very competitive, but they will not be covered in this course.

Let θ and ω denote the parameter vectors for the state and state-action value functions approximations, respectively, so that we have:

$$v^{\pi}(s) \approx v^{\pi}(s, \omega) \tag{6.1}$$

$$q^{\pi}(s, a) \approx q^{\pi}(s, a, \theta) \tag{6.2}$$

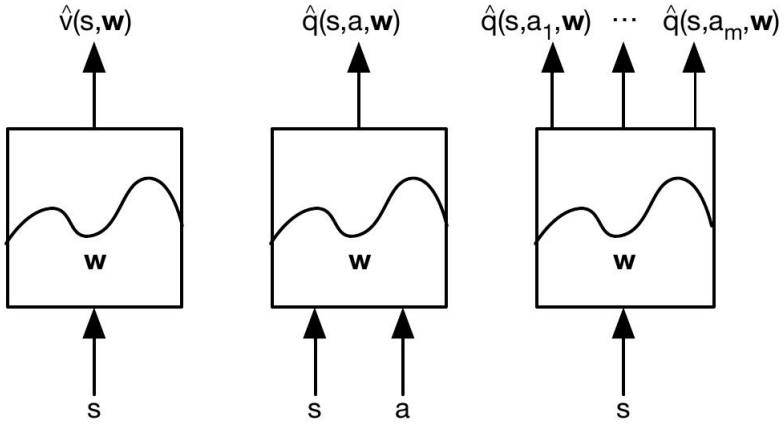


Figure 6.1: Type of Value function approximations

Now, the problem of making a prediction (i.e., estimating the value vector v^π) becomes equivalent to seeking a parameter vector $\omega^\star$ or $\theta^\star$ that is optimal in a certain sense. It is important to realize that apart of the parametric approximation, we will also have to consider the model-free approximation when necessary.

Let $\bar{\phi} : \mathcal{S} \to \Re^N$ be some mapping from states to features, such that $\phi(s)$ gives the *feature vector* of length N that represents state s. The components of such feature vector are given by a set of basis functions, $\phi_1, \ldots, \phi_N : \mathcal{S} \to \Re$:

$$\bar{\phi}(s) = \begin{bmatrix} \bar{\phi}_1(s) \\ \vdots \\ \bar{\phi}_N(s) \end{bmatrix} \tag{6.3}$$

Among many parametrizations, a linear approximation of the form

$$v^\pi(s, \omega) = \bar{\phi}(s)^\top \omega \tag{6.4}$$

has been extensively studied in the literature and it is promising mainly because it leads to solutions with low computational demands. Moreover, it is expected that if one chooses the mapping of features carefully, such that that they are able to capture the structure of the state transitions and rewards, then the linear approximation model will generally provide good results.

Let us assume a finite set of actions for now: $\mathcal{A} \triangleq \{a_1, \cdots, a_A\}$. Then, for the state-action value function approximations, the basis functions are denoted $\phi_1, \phi_2, \cdots, \phi_M : \mathcal{S} \times \mathcal{A} \to \Re$, which are aggregated in a state-action feature vector:

$$\phi(s,a) = \begin{bmatrix} \phi_1(s,a) \\ \vdots \\ \phi_M(s,a) \end{bmatrix} \tag{6.5}$$

It is standard to build $\phi(s,a)$ from $\bar{\phi}(s)$ by using N features for each possible action, so that $M = NA$; setting the N features corresponding to action a equal to $\bar{\phi}(s)$; and the other $(N-1)A$ features, corresponding with the non-selected actions, equal to zero:

$$\phi(s,a_m)^\top = \left[\underbrace{0,\ldots,0}_{a_1},\ldots,\underbrace{\bar{\phi}_1(s),\ldots,\bar{\phi}_N(s)}_{a_m},\ldots,\underbrace{0,\ldots,0}_{a_A}\right]^\top \tag{6.6}$$

Hence, the $q(s,a)$ function can be linearly approximated with parameter $\theta \in \Re^M$ as

$$q(s,a_m,\theta) = \phi(s,a_m)^\top \theta \tag{6.7}$$

6.2 Families of basis functions

The basis functions used for representing the space can be arbitrarily complex, even when the parametric approximation can as simple as a linear approximation. We present the most common basis function schemes, namely polynomial basis, radial basis functions (RBF), the Fourier basis and state aggregation. We remark that choosing the right set of basis functions set can be critical for achieving good approximations. In addition, the approaches considered here can require significant design effort and/or problem insight in order to obtain a satisfactory result.

There are a number of methods for automated discovery of the right basis functions for each problem. However, we will not cover such methods in this course. Moreover, we advance that one of the for the popularity of neural networks is that they are able to somehow implicitly learn good features from raw data, at the same time that learn the parameter vector. We will study neural network approximations in the following chapter.

6.2.1 State aggregation or discretization

For state aggregation, the state space is partitioned into N disjoint subsets. Let $\mathcal{S}_n$ be the n-th subset in this partition, for $n = 1, \cdots, N$. For a given ac-

tion, the approximation assigns the same $v(s)$ values for all states in $\mathcal{S}_n$. This corresponds to a feature vector with binary valued (0 or 1) state dependent :

$$\bar{\phi}_n(s) = \begin{cases} 1 & \text{if } s \in \mathcal{S}_n \\ 0 & \text{otherwise} \end{cases} \tag{6.8}$$

In this case, the state-action features (6.6) can be written compactly as follows:

$$\phi_n(s, a_j) = \begin{cases} 1 & \text{if } s \in \mathcal{S}_n \text{ and } a = a_j \\ 0 & \text{otherwise} \end{cases} \tag{6.9}$$

The main problem with this approach is that the number of aggregated states grows exponential with the dimension, S, of the input state set. Moreover, the variation between two consecutive states (especially for coarse aggregations) could be very abrupt.

6.2.2 The polynomial basis

Consider that the state-space is continuous, so that each state is described by a vector of S state variables: $s = (s_i)_{i=1}^{S}$. The order j polynomial basis is given by:

$$\bar{\phi}_n(s) = \prod_{i=1}^{j} s_i^{c_{n,i}} \tag{6.10}$$

where $c_{n,i}$ is an integer between 0 and j.

For instance, consider a state set that is a subset of a 2-dimensional vector space, such that $\mathcal{S} \subset \Re^2$. Then, the state variable is given as a vector of two components: $s = [s_1, s_2]^\top$, and a second order polynomial basis (i.e., $j = 2$) results in a feature vector of length $M = 7$:

$$\bar{\phi}(s) = \left[1, s_1, s_2, s_1 s_2, s_1^2 s_2, s_1 s_2^2, s_1^2 s_2^2\right]^\top \tag{6.11}$$

Note that the basis functions $\phi_4 \ldots, \phi_7$ are function of both state-variables and, hence, they model the interaction between those variables. In general, the idea of mapping the original input space into a higher dimensional feature space is to make the possibly nonlinear structure of the state-transitions and reward functions more easily described by the linear approximation. This also applies to the the rest of basis functions that we proceed to present.

6.2.3 The Gaussian Radial Basis Functions (RBFs)

Again, consider that the state-space is continuous, where the state vector is composed of S state variables. A radial basis function (RBF) is a real-valued function whose value depends only on the distance between the state variable and some reference point c_n. For simplicity, we consider Gaussian Radial Basis Functions, which are defined as follows:

$$\bar{\phi}_n(s) = \frac{1}{\sqrt{(2\pi)^S |B_n|}} e^{-\frac{1}{2}(s-c_n)^\top B_n^{-1}(s-c_n)} \tag{6.12}$$

where c_n and B_n are the mean and covariance of the n-th basis function, respectively. The main handicap of RBFs is that they have to be distributed evenly in the state space, requiring a number of basis functions that is exponential with the number of state variables S (i.e., the dimension of the original continuous state set). Figure 6.2 illustrates this idea.

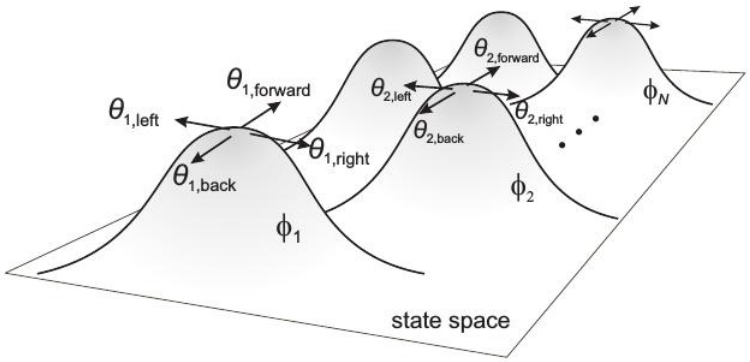


Figure 6.2: Function approximation using RBFs

6.2.4 The Fourier Basis (FB)

In order to explain this approach let us start assuming that the state variable is continuous with one single component, so that $S = 1$ and $\mathcal{S} \subset \Re$. . It is well known that Fourier series are a set of coefficients that represent periodic functions using exponential functions (cosines / sines in real functions) as a basis. The idea is to use just the cosine terms as follows:

$$\bar{\phi}_n(s) = \cos(n\pi s) \tag{6.13}$$

Now, the extension to S-dimensional state sets, where $\mathcal{S} \subset \Re^S$, can be done as follows:

$$\bar{\phi}_n(s) = \cos\left(\pi c_n^\top s\right) \tag{6.14}$$

where the coefficient vector $c_n = [c_{n,1}, \ldots c_{n,S}]$ is composed of integer values $0 \leq c_{n,j} \leq N-1$, for $1 \leq j \leq S$. The basis set is obtained by systematically varying these coefficients.

6.3 Projected Bellman Equation

Let Φ be the matrix of size $S \times N$ formed by stacking the transposed feature vectors $\bar{\phi}(s)^\top$ on top of each other or, equivalently, having the basis function vectors $\bar{\phi}_n$ as columns:

$$\overline{\Phi} \triangleq \begin{bmatrix} \bar{\phi}(1)^\top \\ \vdots \\ \bar{\phi}(S)^\top \end{bmatrix} = \begin{bmatrix} \bar{\phi}_1(1) & \cdots & \bar{\phi}_N(1) \\ \vdots & \cdots & \vdots \\ \bar{\phi}_1(S) & \cdots & \bar{\phi}_N(S) \end{bmatrix} \tag{6.15}$$

Then, the linear approximation (6.4) can be expressed in compact vector form as:

$$v^\pi(s, \omega) = \overline{\Phi}\omega \tag{6.16}$$

In order to solve for ω, we can replace $v^\pi$with its approximation (6.16) in (3.33), so that we obtain a system of linear equations known as approximate Bellman equation:

$$\overline{\Phi}\omega = \mathcal{R}^\pi + \gamma \mathcal{P}^\pi \overline{\Phi}\omega \tag{6.17}$$

We follow the same procedure for state-action value functions. Let Φdenote the $SA \times M$ matrix formed by the transposed feature vectors $\phi(s,a)^\top$, which can be obtained from (6.6):

$$\Phi \triangleq \begin{bmatrix} \phi(1,1)^\top \\ \vdots \\ \phi(1,A)^\top \\ \vdots \\ \phi(S,1)^\top \\ \vdots \\ \phi(S,A)^\top \end{bmatrix} = \begin{bmatrix} \phi_1(1,1) & \cdots & \phi_M(1,1) \\ \vdots & \cdots & \vdots \\ \phi_1(1,A)^\top & \cdots & \phi_M(1,1) \\ \vdots & \cdots & \vdots \\ \phi_1(S,1)^\top & \cdots & \phi_M(S,1)^\top \\ \vdots & \cdots & \vdots \\ \phi_1(S,A)^\top & \cdots & \phi_M(S,A)^\top \end{bmatrix} \tag{6.18}$$

Similar to the previous case, we can obtain an approximate Bellman equation for state-action features from (3.34):

$$\Phi\theta = \mathcal{R} + \gamma \mathcal{P} \Pi \Phi\theta \tag{6.19}$$

We assume that the features constitute a linearly independent set of basis functions, which effectively represent the states. Thus, we assume that $\overline{\Phi}$ and Φ are *full rank* by construction. However, the fixed point equations(6.17) and (6.19) may not have a solution ω or θ in general because the right-hand side need not lie in the range space of $\overline{\Phi}$ or Φ, respectively. To address this issue, we are going to project the right side of (6.17)–(6.19) (i.e., the Bellman operator) onto the range space of the corresponding feature matrix. First of all, we introduce the weighted norm.

Definition 6.1. The weighted Euclidean norm of a vector $x \in \Re^{SA}$ is given by

$$\|x\|_D = \sqrt{x^\top D x} = \sqrt{\sum_{m=1}^{M} d(m)x^2}$$

where D is a diagonal matrix with positive entries $d(1), \ldots, d(SA)$.

Let $\overline{\mathbb{X}}$ and $\mathbb{X}$ denote the range space of the feature matrices $\overline{\Phi}$ and Φ, respectively. Since $\overline{\mathbb{X}}$ and $\mathbb{X}$ are subspaces of $\Re^N$ and $\Re^M$, the projection operator with respect to some metric norm $\|\cdot\|_D$ is defined as:

$$\overline{\Xi} v \triangleq \arg\min_{x' \in \overline{\mathbb{X}}} \|v - x'\|_D^2 \tag{6.20}$$

$$\Xi q \triangleq \arg\min_{x' \in \mathbb{X}} \|q - x'\|_D^2 \tag{6.21}$$

where $\overline{\Xi}$ and Ξ are projection operators onto $\overline{\mathbb{X}}$ and $\mathbb{X}$, respectively; and where we assume that $\overline{D}$ and D are diagonal matrices containing the limiting stationary state and state-action visitation probability distributions associated to the state and state-action Markov chains, respectively (i.e., $\overline{D} = \text{diag}\,[d^\pi]$ where d^π is given by (3.12)). The reason for this particular choice of $\overline{D}$ and D is that it will allow us to obtain model-free sample-based approximations of the value function. The matrices $\overline{\Xi}$ and Ξ are then given by:

$$\overline{\Xi} = \overline{\Phi}\left(\overline{\Phi}^\top \overline{D}\overline{\Phi}\right)^{-1} \overline{\Phi}^\top \overline{D} \tag{6.22}$$

$$\Xi = \Phi\left(\Phi^\top D \Phi\right)^{-1} \Phi^\top D \tag{6.23}$$

Hence, we define the *projected Bellman equation* (PBE) for state and state-action value functions as follows:

$$\overline{\Phi}\omega = \overline{\Xi}\left(\mathcal{R}^\pi + \gamma \mathcal{P}^\pi \overline{\Phi}\omega\right) \tag{6.24}$$

$$\Phi\theta = \Xi\left(\mathcal{R} + \gamma \mathcal{P} \Pi \Phi\theta\right) \tag{6.25}$$

Note that these equations can also be expressed more compactly in terms of the Bellman operator:

$$\overline{\Phi}\omega = \overline{\Xi}T\left(\overline{\Phi}\omega\right) \tag{6.26}$$

$$\Phi\theta = \Xi T\left(\Phi\theta\right) \tag{6.27}$$

This form is useful since it reveals the contraction nature of the projected Bellman operator. On one hand, the projection onto a convex set (like the subspace spanned by the columns of Φ or $\overline{\Phi}$) is always non-expansive. On the other hand, the Bellman operator is contraction. Hence, their composition is also a contraction.

6.3.1 Prediction

Model-based prediction

The prediction problem consists in evaluating the approximate policy for a given policy. Since the feature matrix is given by the environment and part of the problem description, the policy evaluation consists in finding the optimal parameter that solves the PBE. We focus on the state-action value formulation (6.27), since it will be useful for a related model-free control method.

By expanding (6.23) in (6.27), we have:

$$\Phi\theta = \Phi\left(\Phi^\top D\Phi\right)^{-1}\Phi^\top D\left(\mathcal{R} + \gamma\mathcal{P}\Pi\Phi\theta\right) \tag{6.28}$$

Left multiplying both sides by $\left(\Phi^\top D\Phi\right)^{-1}\Phi^\top D$ we have:

$$\left(\Phi^\top D\Phi\right)^{-1}\Phi^\top D\Phi\theta = \left(\Phi^\top D\Phi\right)^{-1}\Phi^\top D\Phi\left(\Phi^\top D\Phi\right)^{-1}\Phi^\top D\left(\mathcal{R} + \gamma\mathcal{P}\Pi\Phi\theta\right) \tag{6.29}$$

By removing the terms that cancel out and by left multiplying both sides by $\Phi^\top D\Phi$, we get:

$$\Phi^\top D\Phi\theta = \Phi^\top D\mathcal{R} + \gamma\Phi^\top D\mathcal{P}\Pi\Phi\theta \tag{6.30}$$

Rearranging terms, we have:

$$\left(\Phi^\top D\Phi - \gamma\Phi^\top D\mathcal{P}\Pi\Phi\right)\theta = \Phi^\top D\mathcal{R} \tag{6.31}$$

Thus, we can obtain the optimal parameter as:

$$\theta^* = \left(\Phi^\top D\Phi - \gamma\Phi^\top D\mathcal{P}\Pi\Phi\right)^{-1}\Phi^\top D\mathcal{R} \tag{6.32}$$

Model free prediction

In order to derive model-free methods that approximate (6.32) from samples, we find convenient to introduce the following shorthands:

$$\Gamma \triangleq \Phi^\top D \Phi \tag{6.33}$$

$$\Lambda \triangleq \Phi^\top D \mathcal{P}\Pi \Phi \tag{6.34}$$

$$z \triangleq \Phi^\top D \mathcal{R} \tag{6.35}$$

so that we can rewrite (6.32) as follows:

$$\theta^* = (\Gamma - \gamma\Lambda)^{-1} z \tag{6.36}$$

The idea is to approximate the terms (6.33)—(6.35) from samples and use them to solve (6.36) with the sample-approximated terms. We consider two popular methods for doing this: Least-Squares Temporal-Difference on state-action value functions (LSTD) and Least-Squares Policy Evaluation (LSPE).

Algorithm 6.1 LSTD for policy evaluation of state-action value functions

Input: π, the policy to be evaluated. Basis functions ϕ
Output: θ, parameter estimate for linear approximation of q^π.

Initialize $\Gamma = 0_{M\times M}$, $\Lambda = 0_{M\times M}$, $z = 0_M$
repeat(for each episode)
  Initialize s, a and observe $\phi(s,a)$
  repeat(for each step in the episode)
    Take action a and observe $r, \phi(s')$
    Choose action $a' \sim \pi(\cdot|s)$ and build $\phi(s',a')$
    $\Gamma \leftarrow \Gamma + \phi(s,a)\phi(s,a)^\top$
    $\Lambda \leftarrow \Lambda + \phi(s,a)\phi(s',a')^\top$
    $z \leftarrow z + \phi(s,a)r$
    $s \leftarrow s'$
    $a \leftarrow a'$
  until s is terminal
until we cannot run more episodes
Compute: $\theta = (\Gamma - \gamma\Lambda)^{-1} z$
return $\hat{q}_\theta = \Phi\theta$

LSTD is described in Algorithm 6.1 and it performs the algorithm in two main steps: First, it estimate the terms. Second, it computes **(6.36)**. Note

that the term $1/t$ is not really necessary (but may improve numerical stability if the sums grows too large). There are recursive implementations of LSTD that avoid having to compute the matrix inversion explicitly.

6.3.2 Control

We follow a policy iteration (PI) approach. As we have seen in previous chapters, the PI scheme iteratively performs two clearly separated steps: *i)* Evaluation of the current policy; *ii)* policy improvement. Here, we present an approximate policy iteration algorithm for model-free control with linear function approximation named least-squares-policy-iteration (LSPI).

LSPI uses LSTD to perform the policy evaluation step, so that we obtain θ^* using (6.36), and then improves the policy by taking the greedy action that maximizes the linearly approximated state-action value function:

$$\pi(s) \in \arg\max_{a \in \mathcal{A}} \phi(s,a)^\top \theta^*, \quad \forall s \in \mathcal{S} \tag{6.37}$$

Recall that we are assuming finite action sets, so that maximizing over the action set can be done relatively efficiently for small to mid (e.g., tens of thousands) number of possible actions, but for very large or continuous action spaces, explicit maximization over all actions in $\mathcal{A}$ may be impractical. In such cases, some sort of global optimization over the actions set may be required to determine the best action.

We consider two possible settings, namely *offline* and *online*. In the *offline* setting, we are given a set of samples of the form of state-action transitions and rewards:

$$\{(\phi(s_t, a_t), r_{t+1}, \phi(s_{t+1}, a_{t+1}))\}_{t=1}^{T} \tag{6.38}$$

These samples can be obtained from a single trajectory, with one policy, or from multiple trajectories corresponding to multiple behavior policies. The main assumption is that the set of samples adequately covers the state-action space, so that for every possible state we have sampled all possible actions several times. This way, when we are evaluating some current policy π for state s_{t+1}, we have samples of the form $\{(\phi(s_t, a_t), r_{t+1}, \phi(s_{t+1}, \pi(s_{t+1})))\}_{t=1}^{T}$ that can be included in our computations. In other words, when we include the transition from s_t to s_{t+1}, we will use the sample $\phi(s_{t+1}, a_{t+1})$ that satisfies $a_{t+1} = \pi(s_{t+1})$. You can see that this is actually an off-policy learning setting and, therefore, there will be some bias caused by the fact that the stationary state-visitation probability of the sampling distribution may be different from the stationary distribution associated to the policy under

Algorithm 6.2 Offline LSPI

Input: Set of samples $\{(\phi(s_t, a_t), r_{t+1}, \phi(s_{t+1}, a_{t+1}))\}_{t=1}^{T}$. Parameter ϵ.
Output: π, the approximate optimal policy.

```
Initialize Γ = 0_{M×M}, Λ = 0_{M×M}, z = 0_M
Initialize π' randomly such that π'(s) > 0, ∀s ∈ S and Σ_{s∈S} π(s) = 1
repeat(for each policy)
    π ← π' (evaluation of π')
    for t=1,...,T do (for each sample)
        Γ ← Γ + φ(s_t, a_t)φ(s_t, a_t)^⊤
        Λ ← Λ + φ(s_t, a_t)φ(s_{t+1}, π(s_{t+1}))^⊤
        z ← z + φ(s_t, a_t)r_{t+1}
    Compute: θ = (Γ − γΛ)^{−1} z
    for all s ∈ S do (policy improvement)
        π'(s) ← arg max_{a∈A} φ(s, a)^⊤θ
until ||π − π'|| < ε
return π
```

evaluation π. This could be corrected with importance sampling though. Algorithm 6.2 shows the details of the offline LSPI algorithm.

Now, we consider *online* LSPI, where the name comes from the fact that the sample set is updated between iterations. The idea is to draw samples from the target that we are evaluating. This is pretty similar to SARSA (Algorithm (5.5)). When the behavior sampling policy used to obtain the data tuples and the target policy are the same, LSPI becomes an on-policy learning algorithm (like SARSA). In order to establish a exploration vs. exploitation tradeoff, we can consider ϵ-greedy, Boltzmann or any other policy:

$$\pi_\epsilon(a|s) \triangleq \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}| & \text{if } a = \arg\max_{a' \in \mathcal{A}} \phi(s, a')^\top \theta \\ \epsilon/|\mathcal{A}| & \text{otherwise} \end{cases} \tag{6.39}$$

To ensure that online LSPI learns quickly, policy improvements can be performed once every few transitions, denoted K, before having an accurate evaluation of the current policy. This can be achieved, e.g., by taking the modulus of the division of t, the counter of evaluated samples, over K. In practice, it is often convenient to set K to a low value (i.e., $K = 1$ or $K = 10$). However, this will likely make that matrix $(\Gamma - \gamma\Lambda)$ singular. In such case, it is convenient to replace the inverse with the pseudoinverse. In addition, we remark that reset the value of Γ, Λ and z at the end of the episode makes only sense if we had enough samples for a good approximation of the variables.

Otherwise, it is better not to reset and accumulate many samples, even at the risk that the estimates could be biased due to the fact that the policy evaluation will include a state visitation probability that accumulates over all the learning episodes, rather than only since the last policy improvement step.

Algorithm 6.3 Online LSPI

Input: Parameter ϵ.
Output: π, the approximate optimal policy.

Initialize $\Gamma = 0_{M\times M}$, $\Lambda = 0_{M\times M}$, $z = 0_M$, $t = 0$
Initialize θ randomly
repeat(for each episode)
    Initialize s, a and build $\phi(s, a)$
    repeat(for each step in the episode)
        Take action $a \sim \pi_\epsilon(\cdot|s)$ using (6.39) and observe r, s'
        Choose action $a' \sim \pi_\epsilon(\cdot|s')$ using (6.39) and build $\phi(s', a')$
        $\Gamma \leftarrow \Gamma + \phi(s, a)\phi(s, a)^\top$
        $\Lambda \leftarrow \Lambda + \phi(s, a)\phi(s', a')^\top$
        $z \leftarrow z + \phi(s, a)r$
        $s \leftarrow s'$, $a \leftarrow a'$
        $t \leftarrow t + 1$
        if $((\mathrm{mod}(t, K) == 0)$ **then** (implicit policy improvement)
            Compute: $\theta = (\Gamma - \gamma\Lambda)^{-1} z$
            Reset $\Gamma = 0_{M\times M}$, $\Lambda = 0_{M\times M}$, $z = 0_M$, $t = 0$
    until s is terminal
until we cannot run more episodes
Compute: $\theta = (\Gamma - \gamma\Lambda)^{-1} z$
for all $s \in \mathcal{S}$ **do**
    $\pi(s) \leftarrow \arg\max_{a\in\mathcal{A}} \phi(s, a)^\top\theta$
return π

Exercise 6.1. LSPI for solving Chain Walk problem. Chain Walk consists of a chain with 4 states (numbered from 1 to 4) and is shown in Figure 6.3. There are two actions available, "left" (L) and "right" (R). The actions succeed with probability 0.9, changing the state in the intended direction, and fail with probability 0.1, changing the state in the opposite direction; the two boundaries of the chain are dead-ends. The reward vector over states is $(0, 1, 1, 0)$ and the discount factor is set to $\gamma = 0.9$. It is clear that the optimal policy is RRLL.

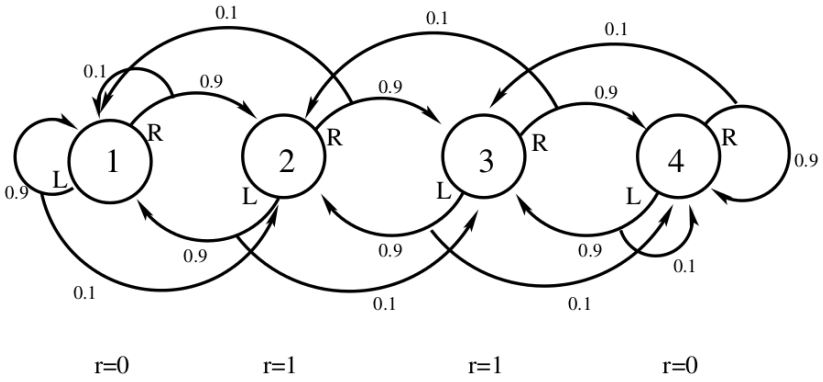


Figure 6.3: Chain Walk problem.

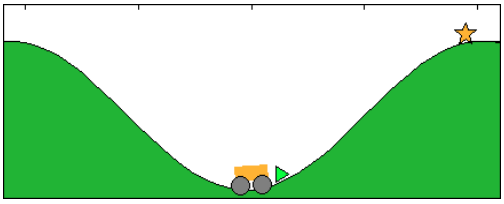


Figure 6.4: Mountain Car problem.

Hint: We suggest to use a set of three polynomial features of the form $\overline{\phi} = [1, s, s^2]$ where s is the state number.

Exercise 6.2. Online LSPI for solving the Mountain Car problem. Mountain Car, a standard testing domain in reinforcement learning, is a problem in which an under-powered car must drive up a steep hill. Since gravity is stronger than the car's engine, even at full throttle, the car cannot simply accelerate up the steep slope. The car is situated in a valley and must learn to leverage potential energy by driving up the opposite hill before the car is able to make it to the goal at the top of the rightmost hill. The domain has been used as a test bed in various reinforcement learning papers. The mountain car problem, although fairly simple, is commonly applied because it requires a reinforcement learning agent to learn on two continuous variables: position and velocity. For any given state (position and velocity) of the car, the agent is given the possibility of driving left, driving right, or not using the engine at all. The agent receives a negative reward at every time step when the goal is not reached. The agent has no information about the goal until an initial success. The task is to implement online-LSPI given by Algorithm 6.3 to obtain the optimal policy that solves this Mountain Car problem.

The problem parameters are:

- The discount factor is set to γ =1.
- The state set is the Cartesian product $\mathcal{S} = [-0.07, 0.07] \times [-1.2, 0.6]$, such that the position component is $x_{\text{pos}} \in [-1.2, 0.6]$, and the velocity component is $x_{\text{vel}} \in [-0.07, 0.07]$. We remark that those intervals are in the real line, so that the state is described as a vector of two real values, $x \triangleq [x_{\text{vel}}, x_{\text{pos}}]^\top \in \mathcal{S} \subset \Re^2$.
- The initial state is $x_{\text{pos},0} = -0.5$, $x_{\text{vel},0} = 0.0$, i.e., the initial state vector is $x_0 = [-0.5, 0.0]$.
- The set of terminal states is given by the condition $x_{\text{pos}} = 0.6$, i.e., the episode finishes when the agent reaches that the right end of the postion interval, independently on the velocity.
- The action set is $\mathcal{A} = [-1, 0, 1]$, where left=-1, no-engine=0 and right=1.
- The reward is -1 at every time step. Hence, the goal of the agent becomes minimizing the number of steps (i.e., negative reward) before reaching the goal.
- The state transition equations for some action $a \in \mathcal{A}$ are given by:

$$x_{\text{vel}} \leftarrow x_{\text{vel}} + a \cdot 0.001 - 0.0025 \cos(3x_{\text{pos}})$$
$$x_{\text{pos}} \leftarrow x_{\text{pos}} + x_{\text{vel}}$$

Hint: We suggest using the RBF features for the state components.

6.4 Approximate Policy Search (APS). Actor-Critic algorithms.

As we have seen, the dominant approach in RL is based on the calculation of any value function and in a second step, determine the optimum policy. However, this initiative has some important limitations: first, it is oriented towards finding deterministic policies, whereas the optimal policy is often stochastic. Nota: no cuadra con lo que dice Bertsekas: hay siempre una política determinista: nAlso, it has been noticed that small changes in the

estimated value of an action can cause it to be or not to be selected and therefore it has been observed lack of convergence in some even simple examples of MDPs with function approximations.

Rather than approximating a value function and using that to compute a deterministic policy, in the technique that we are going to describe now, we approximate a stochastic policy directly using an independent function approximator with its own parameters ϑ.

Typically, gradient updates are performed to find parameters to lead to (locally) maximal returns. Some policy gradient methods estimate the gradient without using a value function. Other methods compute an approximate value function of the current policy and use it to form the gradient estimate. These are called *actor-critic* methods, where the actor is the approximate policy and the critic is the approximate value function.

In this chapter we will be focused on *Actor-Critic* because they are the most popular in RL. Actor critic methods are similar to policy iteration, which also improves the policy on the basis of its value function. The main difference is that in policy iteration, the improved policy is greedy in the value function, i.e., it fully maximizes the value function over the action variables. In contrast, actor-critic methods employ gradient rules to update the policy in a direction that increases the received return. However, the gradient estimate is constructed using the value function.

6.4.1 Fundamentals

In the previous section we approximated the value or the action-value function using parameters θ, ω:

$$v^{\pi}(s) \approx \hat{v}^{\pi}(s, \omega) = \bar{\phi}^{T}(s)\,\omega \tag{6.40}$$

$$q^{\pi}(s, a) \approx \hat{q}^{\pi}(s, a, \theta) \approx \phi^{T}(s, a)\,\theta \tag{6.41}$$

whereas a policy was generated directly from the value function for instance using an $\epsilon-$greedy approach. In this section we will directly parametrize the policy:

$$\pi(s, a, \vartheta) = \mathbb{P}[a \mid s, \vartheta] = \pi_{\vartheta}(s, a) \tag{6.42}$$

or π_{ϑ} for short. The main advantages of the policy-based RL are:

1. Better convergence properties

2. Effective in high-dimensional or continuous action spaces

3. Can learn stochastic policies

while the main disadvantages are:

1. Typically converge to a local rather than global minimum

2. Evaluating a policy is usually inefficient and achieves high variance

The first question to answer is how we are going to measure the quality of a policy. Although there are different alternatives we are going to use an average of the value function:

$$J(\vartheta) = \sum_s d^{\pi_\vartheta}(s)\, v^{\pi_\vartheta}(s) \tag{6.43}$$

where d^{π_ϑ} is the stationary distribution of the Markov chain for π_ϑ.

It is clear that policy-based RL is an optimization problem finding ϑ that maximizes $J(\vartheta)$. Some approaches do not use gradient as genetic algorithms but usually we can obtain greater efficiency using gradient techniques (gradient descent, conjugate gradient, quasi-Newton...) although in this course we will be focused just on gradient descent. So, if $J(\vartheta)$ is the policy objective function, policy gradient algorithms search for a local maximum in $J(\vartheta)$ by ascending the gradient of the policy with respect to parameters ϑ: $\triangle\vartheta = \alpha\nabla_\vartheta J(\vartheta)$ where α is the step-size parameter.

Definition 6.2. Assuming policy π_ϑ is differentiable whenever it is non-zero and that we know the gradient $\nabla_\vartheta\pi_\theta(s,a)$, likelihood ratios exploit the following identity:

$$\nabla_\vartheta\pi_\vartheta(s,a) = \pi_\vartheta(s,a)\frac{\nabla_\vartheta\pi_\vartheta(s,a)}{\pi_\vartheta(s,a)} = \pi_\vartheta(s,a)\nabla_\vartheta\ln(\pi_\vartheta(s,a)) \tag{6.44}$$

where $\nabla_\vartheta\ln(\pi_\theta(s,a))$ is the score function.

Let us introduce the two most used policy parametric models.

Definition 6.3. Softmax policy. Weight actions use linear combination of features $\phi^T(s,a)\vartheta$ whereas the probability of action is proportional to the exponential weight.

$$\pi_\vartheta(s,a) = \frac{e^{\phi^T(s,a)\vartheta}}{\sum_{a'} e^{\phi^T(s,a')\vartheta}} \tag{6.45}$$

Also, for continuous action spaces, a Gaussian policy becomes very natural.

Definition 6.4. Gaussian policy. The mean is a linear combination of state features $\mu(s) = \bar{\phi}^T(s)\vartheta$ (variance typically is fixed) so the probability of action a is:

$$a \sim \mathcal{N}\left(\mu(s), \sigma^2\right) \tag{6.46}$$

We will find the score functions for both strategies

Definition 6.5. For the *softmax policy* the *score function* is

$$\nabla_\vartheta \ln\left(\pi_\vartheta(s,a)\right) = \phi(s,a) - \frac{\sum_{a'} \phi(s,a') e^{\phi^T(s,a')\vartheta}}{\sum_{a'} e^{\phi^T(s,a')\vartheta}} \tag{6.47}$$

Nota: no estoy seguro de esta expresión.
and for the Gaussian policy we have

Definition 6.6. For the *Gaussian policy* the *score function* is

$$\nabla_\vartheta \ln\left(\pi_\vartheta(s,a)\right) = \frac{(a-\mu(s))\,\bar{\phi}(s)}{\sigma^2} \tag{6.48}$$

The most important result of policy gradient techniques is the following theorem.

Theorem 6.1. *Policy Gradient Theorem. For any differentiable policy $\pi_\vartheta(s,a)$ policy objective function $J(\vartheta)$, the policy gradient is:*

$$\nabla_\vartheta J(\vartheta) = \mathbb{E}_{\pi_\vartheta}\left[\nabla_\vartheta \ln\left(\pi_\vartheta(s,a)\right) q^{\pi_\vartheta}(s,a)\right] \tag{6.49}$$

whose proof is omitted but can be found in the book by Sutton&Barto.

6.4.2 Actor - Critic algorithms

Actor - critic methods cover a set of algorithms that share a common aspect: there is one element called critic whose intention is to evaluate the current policy prescribed by the actor. In principle, this evaluation can be done by any evaluation method as those described in this course as TD or LSTD. On the other hand, the critic approximates and updates the value function using samples. The value function is then used to update the actor's policy parameters in the direction of performance improvement. However, the policy is not directly inferred from the value function but updated in the policy gradient direction using a small step size.

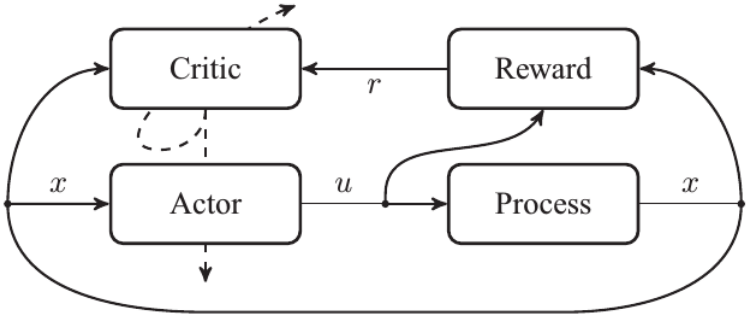


Figure 6.5: Actor - Critic methods

Basic approach

The most basic approach of actor-Critic algorithms maintain two sets of parameters:

1. Critic updates action-value functions parameters θ. For estimating the action-value function we can use whatever approximation method presented in this chapter. Typically we will consider the estimate of the action-state-value function

$$q^{\pi_\vartheta}(s,a) \approx \hat{q}(s,a,\theta) \tag{6.50}$$

2. Actor updates policy parameters ϑ, in the direction suggested by the critic.

and approximate the gradient by its instantaneous value:

$$\nabla_\vartheta J(\vartheta) \approx \mathbb{E}_{\pi_\vartheta}\left[\nabla_\vartheta \ln\left(\pi_\vartheta(s,a)\right) \hat{q}(s,a,\theta)\right] \tag{6.51}$$

$$\triangle\vartheta = \alpha \nabla_\vartheta \ln\left(\pi_\theta(s,a)\right) \hat{q}(s,a,\theta) \tag{6.52}$$

Next table shows the pseudocode of the algorithm where the critic updates θ by linear TD(0) and the actor updates ϑ by policy gradient.

It has been noticed the symbol $\approx$ remarking the fact that $q^{\pi_\vartheta}(s,a)$ is not known and is replaced by $\hat{q}(s,a,\theta)$. Unfortunately, this replacement introduces bias that might make the policy gradient algorithm fail looking for the right solution.

Algorithm 6.4 Actor Critic

Input: Step sizes α, β.
Output: ϑ, parameter for approximate optimal policy.

Initialize θ and ϑ randomly
repeat(for each episode)
    Initialize s and $a \sim \pi_\vartheta(\cdot|s)$ and build $\phi(s,a)$
    repeat(for each step in the episode)
        Take action a and observe r, s'
        Choose action $a' \sim \pi_\vartheta(\cdot|s')$ and build $\phi(s',a')$
        $\delta \leftarrow r + (\gamma\phi(s',a') - \phi(s,a))^\top \theta$
        $\vartheta \leftarrow \vartheta + \alpha\nabla_\vartheta \ln(\pi_\vartheta(a|s))\,\phi(s,a)^\top\theta$
        $\theta \leftarrow \theta + \beta\delta\phi(s,a)$
        $s \leftarrow s'$, $a \leftarrow a'$
    until s is terminal
until we cannot run more episodes
return ϑ

Unbiased actor-critic algorithm

Although in principle this algorithm is biased, if we choose the value function approximation carefully, we can avoid introducing any bias so we can still follow the exact policy gradient. This result is properly formulated in the next theorem.

Theorem 6.2. *Compatible Function Approximation Theorem. If the following two conditions are satisfied:*

1. The Value function approximator is compatible to the policy:

$$\nabla_\theta \hat{q}(s,a,\theta) = \nabla_\vartheta \ln(\pi_\vartheta(s,a)) \tag{6.53}$$

2. Value function parameter θ minimizes the mean square error

$$\varepsilon = \mathbb{E}_{\pi_\vartheta}\left[\left(q^{\pi_\vartheta}(x,u) - \hat{q}(x,u,\theta)\right)^2\right] \tag{6.54}$$

Then the policy gradient is exact

$$\nabla_\vartheta J(\vartheta) = \mathbb{E}_{\pi_\vartheta}\left[\nabla_\vartheta \ln(\pi_\vartheta(s,a))\,\hat{q}(s,a,\theta)\right] \tag{6.55}$$

Proof. if θ is chosen to minimize the mean-squared error, the gradient of ϵ

with respect θ must be zero,

$$\begin{array}{ccc} \nabla_\theta \varepsilon & = & 0 \\ \mathbb{E}_{\pi_\vartheta}\left[\left(q^{\pi_\vartheta}(x,u)-\hat{q}(x,u,\theta)\right)\nabla_\theta Q(s,a,\theta)\right] & = & 0 \\ \mathbb{E}_{\pi_\vartheta}\left[\left(q^{\pi_\vartheta}(x,u)-\hat{q}(x,u,\theta)\right)\nabla_\vartheta \ln\left(\pi_\vartheta(s,a)\right)\right] & = & 0 \\ \mathbb{E}_{\pi_\vartheta}\left[q^{\pi_\vartheta}(x,u)\nabla_\vartheta \ln\left(\pi_\vartheta(s,a)\right)\right] & = & \mathbb{E}_{\pi_\vartheta}\left[\hat{q}(x,u,\theta)\nabla_\vartheta \ln\left(\pi_\vartheta(s,a)\right)\right] \end{array} \tag{6.56}$$

so $\hat{q}(x,u,\theta)$ can be substituted directly into the policy gradient,

$$\nabla_\vartheta J(\vartheta) = \mathbb{E}_{\pi_\vartheta}\left[\nabla_\vartheta \ln\left(\pi_\vartheta(s,a)\right)\hat{q}(s,a,\theta)\right] \tag{6.57}$$

□

It is important to remark that condition $\nabla_\theta \hat{q}(s,a,\theta) = \nabla_\vartheta \ln\left(\pi_\vartheta(s,a)\right)$ basically implies that the features of the $Q-$function are imposed by the policy structure, that is: $\hat{q}^\pi(s,a,\theta) \approx \psi^T(s,a)\theta$ so $\psi(s,a) = \nabla_\vartheta \ln\left(\pi_\vartheta(s,a)\right)$. Unfortunately, we still have an issue to mitigate. It was observed that, although unbiased, the policy gradient has a very large variance. In next section we will introduce a certain modification to mitigate this characteristic.

Unbiased and reduced variance actor-critic algorithm

It is known that we can reduce the variance of an estimator if we subtract a baseline function function $b(s)$ if we make sure that we do not change the expectation. In our particular setting we have

$$\begin{array}{rcl} \nabla_\vartheta J(\vartheta) = \mathbb{E}_{\pi_\vartheta}\left[\nabla_\vartheta \ln\left(\pi_\vartheta(s,a)\right)b(s)\right] & = & \sum\limits_{s\in\mathcal{S}} d^{\pi_\vartheta}(s)\sum\limits_a \nabla_\vartheta \pi_\vartheta(s,a)\,b(s) \\ & = & \sum\limits_{s\in\mathcal{S}} d^{\pi_\vartheta}(s)\,b(s)\,\nabla_\vartheta \sum\limits_a \pi_\vartheta(s,a) \\ & = & 0 \end{array} \tag{6.58}$$

because $\sum\limits_a \pi_\vartheta(s,a) = 1\ \forall\vartheta$. It was found that a good baseline is the state value function $b(s) = \hat{v}(s,\omega)$ depending on parameter ω which is also simple to be estimated from samples.

Definition 6.7. The Advantage function is $A(s,a) = \hat{q}(s,a,\theta) - \hat{v}(s,\omega)$

So, we can rewrite the policy gradient using the Advantage function definition as follows,

$$\nabla_\vartheta J(\vartheta) = \mathbb{E}_{\pi_\vartheta}\left[\nabla_\vartheta \ln\left(\pi_\vartheta(s,a)\right)A(s,a)\right] \tag{6.59}$$

Role of the critic

The goal of the critic is to find an approximate solution to the Bellman equation for the policy provided by the actor. For the true value function $v^{\pi_\vartheta}(s)$ the TD error is $\delta^{\pi_\vartheta} = r + \gamma v^{\pi_\vartheta}(s') - v^{\pi_\vartheta}(s)$ that is an unbiased estimate of the advantage function as presented in the following *proposition.*

Proposition 6.1. *The TD error δ^{π_ϑ} is an unbiased estimate of the advantage function*

Proof. $\mathbb{E}_{\pi_\vartheta}\left[\delta^{\pi_\vartheta} \mid s, a\right] = \mathbb{E}_{\pi_\vartheta}\left[r + \gamma v^{\pi_\vartheta}(s') \mid s, a\right] - v^{\pi_\vartheta}(s)$ where the first term is in fact the state-action value function. So, we have:

$$\mathbb{E}_{\pi_\vartheta}\left[\delta^{\pi_\vartheta} \mid s, a\right] = q^{\pi_\vartheta}(s, a) - v^{\pi_\vartheta}(s) = A^{\pi_\vartheta}(s, a) \tag{6.60}$$

□

So, we can use the TD error to compute the policy gradient

$$\nabla_\vartheta J(\vartheta) = \mathbb{E}_{\pi_\vartheta}\left[\nabla_\vartheta \ln\left(\pi_\vartheta(s, a)\right) \delta^{\pi_\vartheta}\right] \tag{6.61}$$

whereas in practice we can use an approximate TD error $\delta_\omega = r + \gamma v_\omega^{\pi_\vartheta}(s') - v_\omega^{\pi_\vartheta}(s)$. Critic can estimate value functions $v_\omega^{\pi_\vartheta}(s)$ from many targets in a similar way as we reviewed in previous sections but now for the state value where $\hat{v}(s, \omega) = \bar{\phi}^T(s)\,\omega$.

1. For MC, the target is the return v_t

$$\triangle\omega = \alpha\left(v_t - \hat{v}(s, \omega)\right)\bar{\phi}(s) \tag{6.62}$$

2. For TD(0), the target is the TD target $r + \gamma v(s')$

$$\triangle\omega = \alpha\left(r + \gamma\hat{v}(s', \omega) - \hat{v}(s, \omega)\right)\bar{\phi}(s) \tag{6.63}$$

3. For backward view TD(λ),the equivalent update is:

$$\begin{array}{rcc} \delta_t & = & r_{t+1} + \gamma\hat{v}(s_{t+1}, \omega) - \hat{v}(s_t, \omega) \\ e_t & = & \gamma\lambda e_{t-1} + \bar{\phi}(s_t) \\ \triangle\omega_t & = & \alpha\delta_t e_t \end{array} \tag{6.64}$$

This approach is very interesting because we just need to calculate a single set of parameters ω.

Role of the actor

The role of the actor is to derive an updating rule for the coefficient representing the policy ϑ with the innovation intending to minimize the cost-to-go function:

$$\vartheta_{k+1} = \vartheta_k + \beta_k \nabla_\vartheta J\left(\pi_{\vartheta_k}\right) \tag{6.65}$$

where $\nabla_\vartheta J\left(\vartheta\right) = \mathbb{E}_{\pi_\vartheta}\left[\nabla_\vartheta \ln\left(\pi_\vartheta\left(s,a\right)\right) A(s,a,\theta)\right]$. This expectation is approximated using the one step TD error:

$$\triangle\vartheta = \alpha\left(r + \gamma\hat{v}\left(s_{t+1},\omega\right) - \hat{v}\left(s_t,\omega\right)\right)\nabla_\vartheta \ln\left(\pi_\vartheta\left(s_t,a_t\right)\right) \tag{6.66}$$

where we need to know the expressions for the score function for the policy models that we have used.

Actor-Critic Algorithm

Therefore, the *Actor-Critic* algorithm is a set of two stochastic equations as follows:

$$\begin{array}{c} \omega_{t+1} = \omega_t + \alpha_t\delta_t\bar{\phi}\left(s_t\right) \\ \vartheta_{t+1} = \vartheta_t + \beta_t\left(r + \gamma\hat{v}\left(s_{t+1},\omega\right) - \hat{v}\left(s_t,\omega\right)\right)\nabla_\vartheta \ln\left(\pi_\vartheta\left(s_t,a_t\right)\right) \end{array} \tag{6.67}$$

One important detail can be observed: in the second equation, a good estimate of $r + \gamma\hat{v}\left(s_{t+1},\omega\right) - \hat{v}\left(s_t,\omega\right)$ is required in order to make a proper update of the policy parameters. This is typically solved with two time scales: it can be controlled by the step-size or just updating ω_t much more often than ϑ_t.

Other implementations of the actor update

Although the previous recursion follows directly the theoretical developments, different authors have provided alternative contributions. We would like to emphasize:

$$\begin{array}{c} \omega_{t+1} = \omega_t + \alpha_t\delta_t\bar{\phi}\left(s_t\right) \\ \vartheta_{t+1} = \vartheta_t + \beta_t\omega_{t+1} \end{array} \tag{6.68}$$

which is very simple but in fact works very well because it can be shown that it corresponds to the *Natural Gradient* of the *Cost-to-go* function. The Natural Gradient refers to a different update rule incorporating knowledge about the curvature of the space into the gradient. Typically, this knowledge is extracted from the *Fisher Information Matrix* (FIM)

$$\nabla_{\vartheta}^{nat}\pi_{\vartheta}(s,a) = G_{\vartheta}^{-1}\nabla_{\vartheta}\pi_{\vartheta}(s,a) \tag{6.69}$$

where $G_{\vartheta} = \mathbb{E}_{\pi_{\vartheta}}\left[\nabla_{\vartheta}\ln(\pi_{\vartheta}(s,a))\nabla_{\vartheta}\ln(\pi_{\vartheta}(s,a))^{T}\right]$ is the FIM. Fortunately in this particular case, it has the simple expression in eq. (6.68) meaning that we have to update the actor parameters in the direction of critic parameters.

Theorem 6.3. $\nabla_{\vartheta}^{nat}\pi_{\vartheta}(s,a) = G_{\vartheta}^{-1}\nabla_{\vartheta}\pi_{\vartheta}(s,a) = \omega$

Proof. Using the compatible function approximation $\nabla_{\omega}A(s,a) = \nabla_{\vartheta}\ln(\pi_{\vartheta}(s,a))$ the natural policy gradient simplifies,

$$\begin{array}{rcc} \nabla_{\vartheta}J(\vartheta) & = & \mathbb{E}_{\pi_{\vartheta}}\left[\nabla_{\vartheta}\ln(\pi_{\vartheta}(s,a))A(s,a)\right] \\ & = & \mathbb{E}_{\pi_{\vartheta}}\left[\nabla_{\vartheta}\ln(\pi_{\vartheta}(s,a))\nabla_{\vartheta}^{T}\ln(\pi_{\vartheta}(s,a))\omega\right] \\ & = & G_{\vartheta}\omega \\ \nabla_{\vartheta}^{nat}J(\vartheta) & = & \omega \end{array} \tag{6.70}$$

□

Nota: revisar y aclarar la demostración.

Chapter 7

Non-linear approximations

7.1 Neural networks for approximating the value function

Although theoretically appealing, the main drawback of the linear function approximation approach presented in the previous chapter is that we have to provide rich enough features. There have been reported some experiments with hand-crafted features where linear feature approximations have led to excellent results for particular problems. The problem with hand-crafted features is that they do not usually translate to other problems, and sometimes even to variations of the same problem domain. The cornerstone of any RL is to work well in any domain and with minimum human engineering (no step-size tuning, no feature hand-crafting, etc.). And although there are some methods for feature automatic-discovery for linear approximations, their success has been limited and this is an active research.

An alternative approach consists in allowing more general approximation architectures. This way, instead of having to figure out a set of basis functions that transform the input data in a set of features that work well under linear approximations, we can learn the transformation that better suits any given input data representation.

Neural networks (NN) are well known universal approximation machines, able to learn to approximate any smooth nonlinear function with arbitrary degree of accuracy. The main limitation for NN is that they have been typically difficult to train. In the last decade, several advances have turned the process of learning NN simpler and more effective, becoming mainstream for multiple learning problems. Perhaps, the key idea for this success has been using *very large training datasets* for training NN architecutres *multiple hiden layers*. This new trend in NN is commonly known as *deep learning*

(DL) where deep refers precisely to having multiple hiden layers (as opposed to previously more common *shallow* architectures with a single hiden layer).

The combination of DL with RL is usually known as *deep reinforcement learning* (DRL) and consists in using a deep NN for approximating either the value function, or the policy, or both. In this chapter we present two algorithms, named *Neural Fitted Q-iteration* (NFQ) and *Deep Q-learning networks* (DQN). The former, is more or less similar to LSTD in that it performs regression on the state-action value function, though here we use NN to perform non-linear regression. The latter was one of the main breakthroughs of AI in the last years and consists in performing non-linear regression in an online Q-learning manner, where the value function is approximated with a deep neural network.

However, RL presents several challenges from a deep learning perspective. Firstly, most successful deep learning applications have required large amounts of hand-labeled training data. But RL algorithms, on the other hand, must be able to learn from a scalar reward signal that is frequently sparse, noisy and delayed. The delay between actions and resulting rewards, which can be thousands of timesteps long, seems particularly daunting when compared to the direct association between inputs and targets found in supervised learning. Another issue is that most deep learning algorithms assume the data samples to be independent, while in RL one typically encounters sequences of highly correlated states. Furthermore, in RL the data distribution changes as the algorithm learns new behaviors, which can be problematic for deep learning methods that assume a fixed underlying distribution.

In this chapter, we consider that the environment's internal state, $s_t \in \mathcal{S}$, may or may not be observed by the agent. When the state is not directly observed, we assume that the agent can only observe some feature representation of the state, denoted $x_t \in \Re^N$. These input data is usually preprocessed (normalization and mean subtraction) for improving the learning capabilities of the NN, obtaining some features of the form $\phi(x_t)$.

7.1.1 Neural Fitted Q-iteration (NFQ)

Fitted Q-iteration performs starts with a set of samples in the form of state-action transitions and rewards $\{(s_t, a_t, s_{t+1}, r_{t+1})\}_{t=0}^{T}$, where s_t and s_{t+1} may have to be replaced by $\phi(x_t)$ and $\phi(x_{t+1})$. At every iteration $j = 1, \ldots, \text{maxIter}$, Neural Fitted Q-iteration (NFQ) uses a neural network to approximate the Bellman equation. Similar to the linearly approximated Bellman equation given by (6.27), the nonlinear approximation is given by:

$$\text{Predict}\left((s_t, a_t), \theta\right) = T\left(\text{Predict}\left((s_t, a_t), \theta\right)\right) \tag{7.1}$$

where $T(\cdot)$ denotes the Bellman operator (as usual) and $\text{Predict}((s_t, a_t), \theta)$ implies a feedforward pass to give the ouptut of the neural network for input data (s_t, a_t) and some parameter vector θ. Thus, NFQ aims to learn the parameters that minimize the approximate Bellman error:

$$\theta^* = \arg\min_{\theta} \mathbb{E} \left\| \text{Predict}((s,a),\theta) - T(\text{Predict}((s,a),\theta)) \right\|^2$$

In practice, we have to minimize the empirical risk over a set of samples. In particular, for every MDP sample $(s_t, a_t, s_{t+1}, r_{t+1})$, we build a new sample of the form:

$$(\text{Predict}((s_t, a_t), \theta_j), B_t) \tag{7.2}$$

where θ_j is the current set of parameters (let us say at iteration j) and $B_{t,j}$ is the approximate Bellman operator for the transtion, estimated with the current set of parameters:

$$B_{t,j} = \begin{cases} r_{t+1} & \text{for terminal } s_{t+1} \\ r_{t+1} + \gamma \max_{a' \in \mathcal{A}} \text{Predict}((s_{t+1}, a'), \theta_j) & \text{for non-terminal } s_{t+1} \end{cases} \tag{7.3}$$

Note that in order to maximize over the action set, we have to perform the prediction for each possible $a \in \mathcal{A}$, which implies doing a feedforward pass for all actions.

Once we have the set of T samples for the regression problem

$$\mathbb{S}_j \triangleq \{(\text{Predict}((s_t, a_t), \theta_j), B_{t,j})\}_{t=1}^{T} \tag{7.4}$$

we perform some form of stochastic gradient descent (SGD) with back-propagation (in particular, R-prop has shown good results but more modern methods could be considered) to optimize the empirical approximate Bellman error, which is given by:

$$\theta^* = \arg\max_{\theta} \frac{1}{T} \sum_{t=1}^{T} (\text{Predict}((s_t, a_t), \theta) - B_t)^2 \tag{7.5}$$

Note that we have included iteration subscript j in the dataset $\mathbb{S}_j$. The reason is that we have to update the dataset every time that we have obtained a new parameter vector.

It is common to perform the optimization after each episode, though other conditions can be equally valid (e.g., the condition $\text{mod}(t, K) = 0$ used for LSPI). Once we have obtained a new set of parámeters, say θ_{j+1}, we

implicitly have a new (hopefully improved) policy as the greedy policy that can be obtained from the new approximate value function:

$$\pi_j(s) = \max_{a' \in \mathcal{A}} \text{Predict}\left((s_{t+1}, a'), \theta_j\right) \tag{7.6}$$

Therefore, similar to previous algorithms, we require some exploration vs. exploitation tradeoff. In particular, we can update the behavior policy with an ϵ-greedy policy of the form:

$$\pi_\epsilon(a|s) \triangleq \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}| & \text{if } a = \arg\max_{a' \in \mathcal{A}} \text{Predict}\left((s, a'), \theta\right) \\ \epsilon/|\mathcal{A}| & \text{otherwise} \end{cases} \tag{7.7}$$

so that we can draw samples of better trajectories to better estimate the approximate value function of the target. This is especially important if the initial sampling policy was very different from the optimal. In such case, apart of updating the dataset with the new parameter θ_{j+1} at every iteration, it also grows with new samples.

The details of NFQ are shown in Figure 7.1, where the routine $\text{SGD}(\mathbb{S})$ means some form of stochastic gradient descent (e.g,. R-prop). Note that the optimization with SGD is performed after all feedforward passes have been done. Thus, we can use the neural network for updating the dataset $\mathbb{S}_j$ with the current θ_j; then, we optimize the network to obtain θ_{j+1}.

Algorithm 7.1 Neural Fitted Q-iteration (NFQ)

Input: Parameter ϵ.
Output: π, the approximate optimal policy.

Initialize θ randomly
Initialize $t = 0$ and $\mathbb{S} = \{\}$ empty
for each episode **do**
    Initialize s_0
    for each step t in the episode **do**
        Take action $a_t \sim \pi_\epsilon(\cdot|s)$ using (7.7) and observe r_{t+1}, s_{t+1}
        Compute $\text{Predict}\left((s_t, a_t), \theta\right)$ with current θ
        Compute B_t using (7.3) with current θ
        Augment sample set $\mathbb{S} \leftarrow \{\mathbb{S}, (\text{Predict}\left((s_t, a_t), \theta\right), B_t)\}$
    Compute parameter on whole dataset: $\theta = \text{SGD}(\mathbb{S})$
for all $s \in \mathcal{S}$ **do**
    $\pi(s) \leftarrow \arg\max_{a' \in \mathcal{A}} \text{Predict}\left((s, a'), \theta\right)$
return π

Historically, NFQ has used multilayer perceptrons. In theory, NFQ could use a deep architecture; however, the computational cost of using the whole dataset $\mathbb{S}$ at every iteration of the optimization algorithm could be prohibitive. Thus, when approximating the Bellman equation with deep architectures, we need to restrict the optimization dataset to small minibatches, but this could be unstable and some extra techniques are required. This is precisely what Deep Q-learning networks (DQN) does, as explained in the following section.

7.1.2 Deep Q-learning networks (DQN)

Deep Q-learning networks (DQN) is very similar to NFQ. As discussed above, NFQ is mainly a batch algorithm for learning a multilayer perceptron (shallow NN), while DQN looks more like an online algorithm and it is used for learning deep architectures. Note that learning a deep architecture by using the whole dataset at every iteration of SGD would be very costly, so that DQN uses single samples or minibatches in order to reduce the computational cost. One problem with this single-sample/minibatch approach is that sequential samples will be highly correlated. Thus, the SGD algorithm, which assumes uncorrelated samples, may perform poorly. Moreover, SGD assumes that the samples are drawn from a stationary distribution. However, the behavior policy should be updated regularly to sample trajectories resulting from the improved policies, so that we can estimate their value functions more accurately (this is especially important if the initial policy was far from optimal). DQN surmounts these problems as follows.

DQN introduces *experience replay*, which consists in storing samples of agent's experiences $e_t = (s_t, a_t, r_t, s_{t+1})$, pooled over many episodes, into a dataset $\mathbb{S} = \{e_t\}_{t=1}^{T}$ of finite length T, named replay memory, where s_t and s_{t+1} may be replaced by $\phi(x_t)$ and $\phi(x_{t+1})$, if needed. Then, we build a minibatch of N samples, $\mathbb{B} = \{e_j\}_{j\in\mathbb{N}} \subseteq \mathbb{S}$, where $\mathbb{N}$ is a set of indexes drawn randomly from the samples in the replay memory, and apply SGD over $\mathbb{B}$ to minimize the (empirical) Bellman error:

$$\theta^* = \arg\max_{\theta} \frac{1}{N} \sum_{j\in\mathbb{N}} \left(\text{Predict}\left(e_j, \theta\right) - B_j\right)^2 \tag{7.8}$$

where B_j is obtained from (7.3). Typically, when the replay memory is full, the oldest samples are removed so that there is space for the new ones. After performing experience replay, the agent selects and executes an action according to the current ϵ-greedy policy given by (7.7) (i.e., obtained from the current approximation of the state-action value function), and obtain a

new sample e_t that is stored in the dataset.

The experience replay approach has several benefits. First, each step of experience is potentially used in many weight updates, which allows for greater data efficiency. Second, learning directly from consecutive samples is inefficient, due to the strong correlations between the samples; randomizing the samples breaks these correlations and therefore reduces the variance of the updates. Third, when learning on-policy, the current parameters determine the next data sample that the parameters are trained on. For example, if the maximizing action is to move left then the training samples will be dominated by samples from the left-hand side; if the maximizing action then switches to the right then the training distribution will also switch. It is easy to see how unwanted feedback loops may arise and the parameters could get stuck in a poor local minimum, or even diverge catastrophically. By using experience replay, the behavior distribution is averaged over many of its previous states, smoothing out learning and avoiding oscillations or divergence in the parameters. Note that when learning by experience replay, we are implicitly learning off-policy (because the distribution of samples in the replay memory do not match the distribution induced by the current policy).

Regarding the nonstationarity of the sample distribution induced by updating the policy, note that the greedy policy changes rapidly with slight changes to the state-action value function, so that the greedy policy may oscillate abruptly, and the distribution of data can swing from one extreme to another. In order to mitigate this problem, DQN uses two NN, namely one for evaluating the current state-action pair:

$$\text{Predict}\left(e_j, \theta_{\text{sa},j}\right) \tag{7.9}$$

and another network for evaluating the target:

$$\max_{a' \in \mathcal{A}} \text{Predict}\left(\left(s_{t+1}, a'\right), \theta_{\text{target},h}\right) \tag{7.10}$$

Under this implementation, we have two weigh vectors, $\theta_{\text{sa},j}$ and $\theta_{\text{target},h}$, which are updated at different speeds, note that we are using different iteration index j and h. In particular, the target network ($\theta_{\text{target},h}$) is *frozen* and updated much less often than the network used to evaluate the current state-action pair ($\theta_{\text{sa},j}$). This way, we avoid oscillations and break correlations between Q-network and target.

The third key point for making DQN effective is to clip the rewards, preventing the Q-values of being too large, which makes backpropagation more stable—at the cost of not being able to distinguishing between small and large rewards.

The details of DQN are given by Algorithm 7.2, where the input parameters are: ϵ for exploration with behavior policy, numer of samples in the replay memory T, number of samples in the minibatch N, and K number of iterations of θ_{sa} before updating the target network θ_{target}.

Algorithm 7.2 Deep Q-learning networks (DQN)

Input: Parameters: ϵ, T, N, K
Output: π, the approximate optimal policy.

Initialize replay memory $\mathbb{S}$ to capacity T
Initialize θ_{sa} and θ_{target} randomly
for each episode e **do**
    Initialize s_0
    for each step t in the episode **do**
        Take action $a_t \sim \pi_\epsilon(\cdot|s)$ using (7.7) and observe r_{t+1}, s_{t+1}
        Compute Predict $\left((s_t, a_t), \theta_{\text{sa}}\right)$
        Compute B_t using (7.3) with θ_{target}
        Augment sample set $\mathbb{S} \leftarrow \{\mathbb{S}, (\text{Predict}\left((s_t, a_t), \theta_{\text{sa}}\right), B_t)\}$
        Update memory replay $\mathbb{S}$ with current sample $(s_t, a_t, s_{t+1}, r_{t+1})$
        Sample randomly a set $\mathbb{N}$ of N indexes between 0 and T
        Build minibatch $\mathbb{B} \leftarrow \{(s_j, a_j, s_{j+1}, r_{j+1})\}_{j \in \mathbb{N}}$ from $\mathbb{S}$
    Compute: $\theta_{\text{sa}} = \text{SGD}\left(\mathbb{B}\right)$
    if $\text{mod}(e, K)$ **then**
        $\theta_{\text{target}} \leftarrow \theta_{\text{sa}}$
for all $s \in \mathcal{S}$ **do**
    $\pi(s) \leftarrow \arg\max_{a' \in \mathcal{A}}$ Predict $\left((s, a'), \theta\right)$
return π

Exercise 7.1. Solve the mountain car problem described in Exercise 6.2 using the implementation of DQN available in open source library Keras-RL. The exercise is composed of two phases, one to get used to the libraries and another one to tune the algorithm to make it work for the mountain car problem.

1. Setting development environment
    (a) Download PyCharm IDE and install in your computer.
    (b) Download Anaconda and install in your computer
    $ conda -V
    $ conda update conda

(c) Create virtual environment:
$ conda create -n rlclass python=3.5 anaconda
$ conda update conda
$ source activate rlclass

(d) Check that OpenAI Gym package is installed in the environment:
$ conda list -n rlclass | grep gym
$ conda update conda
$ source activate rlclass

i. Create project with file test.py
```
import gym
env = gym.make('MountainCar-v0')
env.reset()
env.render()
```

(e) Install Keras library: $ conda install -n rlclass keras

(f) Read Keras documentation.

(g) Install Keras-RL library (inside the rlclass virtual environment):
$ pip install keras-rl

(h) Read Keras-RL documentation.

(i) Download and run Keras DQN example "dqn_cartpole.py" form github.

2. Solve mountain car problem.

(a) Modify "dqn_cartpole.py" example to solve the mountain car environment. It basically consists in changing the environment ID.

(b) Tune the algorithm and environment parameters so that we can solve this problem. Suggested tunning:

i. Change policy.

ii. Change NN parameters.

iii. Change size of replay memory.

iv. Edit environment to change maximum number of steps.
Do you see any change with any of these changes? Why or why not?

Hint: The key is to understand how the reward is propagated when estimating the value function. If the agent has not visited any landmarks that suggest the right direction, it will only learn random noise.