Autonomous and Adaptive Systems
course notes
University of Bologna

prof. Mirco Musolesi

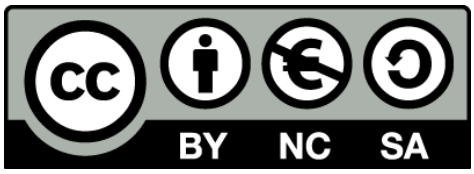
AY. 2020-2021

Alessandro Pomponio

February 2021

Material used for these notes includes:

- Prof. Musolesi's slides, available here: `https://www.mircomusolesi.org/courses/AAS20-21/AAS20-21-main/`
- Richard S. Sutton and Andrew G. Barto's "Reinforcement Learning, An Introduction" book, available here: `http://incompleteideas.net/book/RLbook2020.pdf`



The latest version of these notes can be found in my GitHub repository here: `https://github.com/AlessandroPomponio/aas-course-notes`.

Contents

1 Intelligent Systems

1.1 Computing machinery and intelligence

The course starts by reading Turing's article *"Computing machinery and intelligence"*, published on the Mind journal in October 1950 [Tur50]. In it, Turing complains about the intrinsic ambiguity of the question "can machines think?", which relies on the definition of both "machine" and "think". To reformulate this problem by means of less ambiguous words, he introduces **the imitation game**, in which an interrogator (C) tries to guess by asking questions and receiving answers, which of the other two participants (that are in a different room and that he only knows by means of the labels X and Y) is a man (A) and which is a woman (B). A's goal is to cause C to make the wrong identification, while B's goal is to help the interrogator. To limit the number of indirect clues that the interrogator may receive, the answers should be typewritten or repeated by an intermediary.

Turing then suggests replacing the original question "can machines think?" with a new one: *"what will happen when a machine*[1] *takes the part of A in this game? Will the interrogator decide wrongly as often when the game is played like this as he does when the game is played between a man and a woman?"*. The goal of the game is not to find out *"whether all digital computers would do well in the game nor whether the computers at present available would do well, but whether there are imaginable computers which would do well"*.

In the last chapter, Turing shows a handful of solutions to tackle the problem of learning machines, which he argues to be a programming problem; he starts with an analysis of the human brain which, to him, has three components:

- The initial state of the mind (say at birth).
- The education to which it has been subjected.
- Other experience, not to be described as education, to which it has been subjected.

Instead of producing a program to simulate an adult mind, he suggests producing one that simulates the one of a child and subject it to an appropriate

[1] Later he restricts the definition of machine to digital computers

"education" to obtain an adult brain. He is very aware that *"We cannot expect to find a good child-machine at the first attempt. One must experiment with teaching one such machine and see how well it learns. One can then try another and see if it is better or worse. There is an obvious connection between this process and evolution* [...]*"*. The teaching process will also have to involve punishments and rewards: *"The machine has to be so constructed that events which shortly preceded the occurrence of a punishment-signal are unlikely to be repeated, whereas a reward-signal increased the probability of repetition of the events which led up to it"*. Finally, he foresees one of the problems that is still unsolved in the machine learning field: *"An important feature of a learning machine is that its teacher will often be very largely ignorant of quite what is going on inside, although he may still be able to some extent to predict his pupil's behaviour"*.

1.2 Intelligent machines

After seeing a few examples of autonomous systems, like self-driving cars and agents that play videogames, we understand that our goal is to be able to build machines that can learn from experience, trying things out on their own, without any human intervention. We first start by giving some definitions.

1.2.1 Intelligent agents

We define an **intelligent agent** as an entity that perceives its environment and takes actions that maximize the probability of achieving its goals. It is important to note that the agent does not know what set of actions will allow it to reach the goal, it just "moves" towards actions that maximize the probability of it happening. Agents may also be physically situated (we call them **robots**) or not (we refer to them as **software agents** or **bots**).

1.2.2 Adaptive agents

We define an **adaptive agent** as an entity that can respond to changes in its environment. This is possible thanks to a lack of determinism: the agent will adapt and react to the environment (which may also include other agents) and take different actions.

Learning can take place in various ways: at the end of a generation, with **natural selection** and the survival of the fittest, or during a generation,

with a method that is more similar to **reinforcement learning**.

1.2.3 Autonomous agents

We define an **autonomous agent** as an entity that only relies on its perception and acts in the world independently from its designer. A key characteristic of this type of agent is that they should be able to compensate for partial knowledge: in the beginning they may only know how to perceive the environment and how to take a certain set of actions; from a practical point of view, it makes sense to provide the agent with some knowledge of the world and the ability to learn. After sufficient experience of its environment, an intelligent agent can become effectively independent of its prior knowledge.

1.2.4 Designing agents

When designing agents, we need to take into consideration the following dimensions:

- Performance: how "good" is the agent.
- Environment: what is "around" the agent.
- Actuators: how the agent can take actions in the environment.
- Sensors: how the agent perceives the environment.

A schema of how these dimensions are linked can be found in figure 1, along with a few examples of agents in figure 2.

1.3 Characteristics of the environments

The environment in which our agent is situated may be of different types:

- **Fully observable vs partially observable**: we may or may not be able to see the entire environment (e.g., there may be occlusions limiting our sight).
- **Deterministic vs stochastic**: the environment may be predictable (e.g., governed by the laws of Newtonian physics) or not (e.g., the environment may be subject to changes performed by another agent).

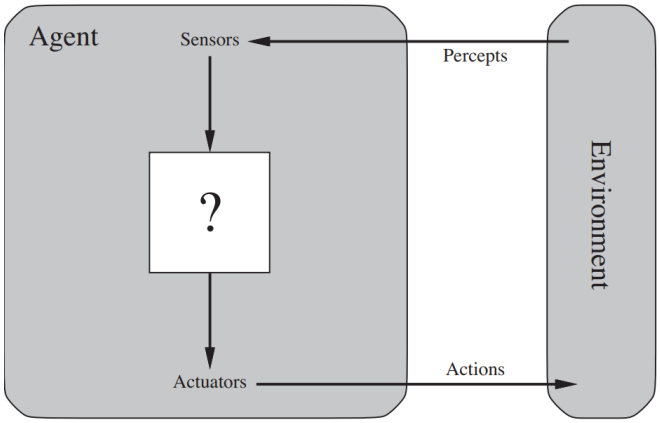


Figure 1: Schema of the interaction between an agent and the environment.

- **Episodic vs sequential**: the environment may be divided in "episodes" that have a beginning and an end (e.g., the levels of a game) or open-ended (e.g., a self-driving car that keeps driving).
- **Static vs dynamic**: the environment may or may not change over time (an action that we take now could have a different result compared to when we took it in the past, e.g., certain agents with whom we collaborated in the past, may not do so anymore; driving in dry conditions is different compared to driving in the rain or in the snow).
- **Discrete vs continuous**: the environment may be discrete or continuous (e.g., the wind speed is a continuous attribute of the environment).
- **Single agent or multi-agent**: there may be multiple agents in the environment, and we may want them to collaborate.

1.4 A categorization of intelligent agents

There are essentially four basic kinds of agents:

- **Simple reflex agents**.
- **Model-based reflex agents**.
- **Goal-based agents**.
- **Utility-based agents**.

	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Health patient, minimise cost, lawsuit	Patient, hospital, staff	Display questions, tests, treatments, etc.	Keyboard entry, patients' answers
Satellite image analysis system	Correct image categorisation	Downlink from satellite	Display categorisation of scenes	Colour pixel arrays
Part-picking robot	Percentage parts in correct bins	Conveyor belt with parts, bins	Jointed arm and hand	Camera, joint angle sensors
Refinery controller	Maximise purity, yield, safety	Refinery, operators	Valves, pumps, heaters, displays	Temperature, pressure, sensors
Interactive language tutor	Maximise student's test score	Set of students	Displays exercises, suggestions	Keyboard entry
Automatic display of advertisements	Click rates/ purchase conversion	Websites, online retailers, users	Display advertisements	Automatic extraction of content, clicks

Figure 2: Examples of agents and their characteristics.

The behavior of these agents can be hard-wired, or it can be acquired, improved and optimized through learning.

1.4.1 Simple reflex agents

Simple reflex agents select actions on the basis of the current perceptions, ignoring the perception history. They are the most basic form of agents and are based on condition-action rules (also called **stimulus-response** rules, productions, or if-then rules). (See figure 3)

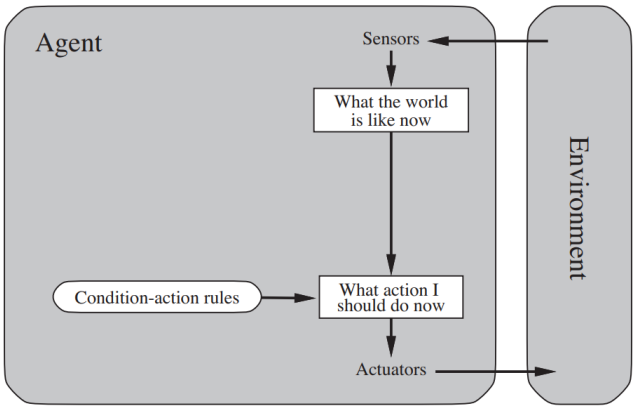


Figure 3: Schema of a simple reflex agent.

1.4.2 Model-based reflex agents

Model-based agents keep an internal state and depend on two types of knowledge:

- How the world evolves independently from the agent (e.g., the trajectory that a bullet/a stone follows when shot/thrown).
- How the actions of the agent affect the world (e.g., if I turn the wheel to the right, the car moves to the right).

The internal state is essentially used to keep track of what it is not possible to see/perceive at the current time. It depends on the perception history and, for this reason, it reflects at least some of the unobserved aspects of the current state. (See figure 4)

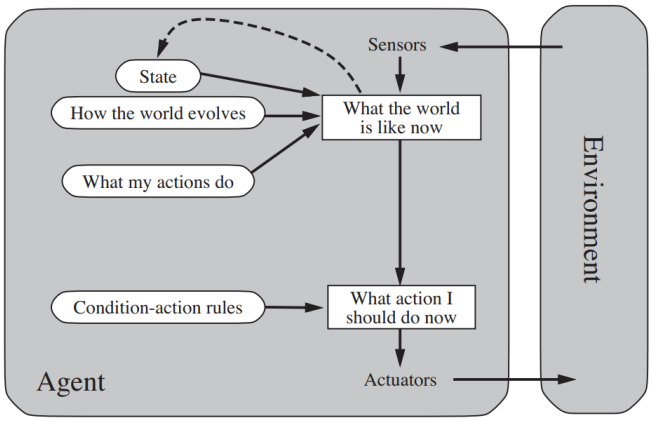


Figure 4: Schema of a model-based reflex agent.

1.4.3 (Model-based) Goal-based agents

Goal-based agents act in order to achieve their goals. If we can achieve the goal by carrying out a single action, goal-based action selection is straightforward; in the other cases, the agent needs to consider a long sequence of actions by means of search and planning. (See figure 5)

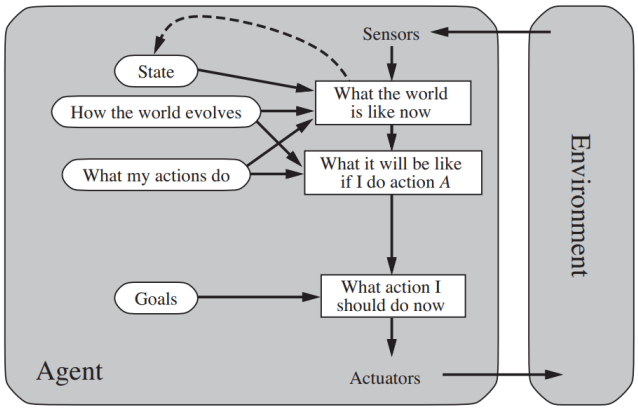


Figure 5: Schema of a goal-based agent.

1.4.4 (Model-based) Utility-based agents

Goals are not sufficient to generate "high-quality behavior" in most environments, since there are usually states that are preferrable to others. In order

to code this "preference", we use utility functions that map a state (or a sequence of states) to a real number (e.g., we want to get to a destination by following the shortest or quickest path). (See figure 6)

Note that how to model these preferences is one of the current unsolved and "hot" topics in the artificial intelligence field.

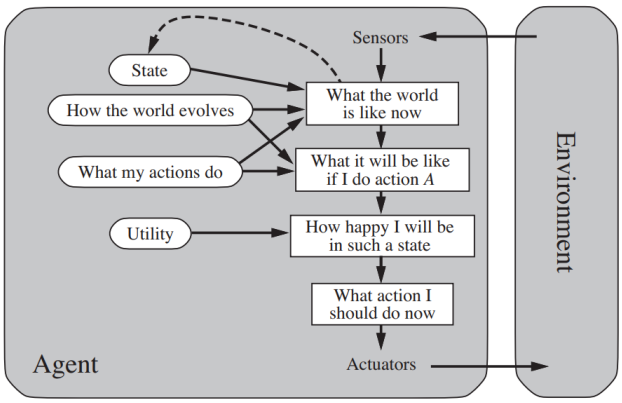


Figure 6: Schema of a utility-based agent.

To conclude this first introduction, let us quickly consider the topic of learning.

1.5 Learning

As we have said before, the behavior of the agents can be pre-programmed (hard-wired, fixed) or it can be learned by means of a learning component. This component can be based on a model of the world and the gain towards a certain goal (possibly expressed in terms of the change of the value of utility functions) can be expressed through rewards. This behavior is at the basis of the type of learning that we will explore in detail in this course, called **reinforcement learning**. Following Herbert Simon's definition of autonomous and adaptive systems, we will consider *"machines that think, that learn and that create"*.

2 Introduction to Reinforcement Learning

Learning from interaction is an idea that is at the basis of nearly all theories of learning and intelligence, among which we find reinforcement learning.

Reinforcement learning is learning what to do and how to map situations to actions, so as to maximize a numerical reward (it is goal-directed learning from interaction). The learner is not told which actions to take, but instead it must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics (trial-and-error search and delayed reward) are the two most important distinguishing features of reinforcement learning.

We will now introduce finite Markov decision processes, a mathematical framework that we are going to use.

2.1 Finite Markov Decision Processes

Markov Decision Processes (MDPs) are a mathematically idealized formulation of reinforcement learning for which precise theoretical statements can be made. They provide a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker[2].

At each time step t, the process is in some state s, and the decision maker may choose an action a that is available in state s. The process responds at the next time step by randomly moving into a new state s', and giving the decision maker a corresponding reward $R_a(s, s')$. The probability that the process moves into its new state s' is influenced by the chosen action. Specifically, it is given by the state transition function $P_a(s, s')$. Thus, the next state s' depends on the current state s and the decision maker's action a. But **given s and a, it is conditionally independent of all previous states and actions**; in other words, the state transitions of an MDP satisfy the Markov property *(memoryless property of a stochastic process)*.

[2]See `https://en.wikipedia.org/wiki/Markov_decision_process`

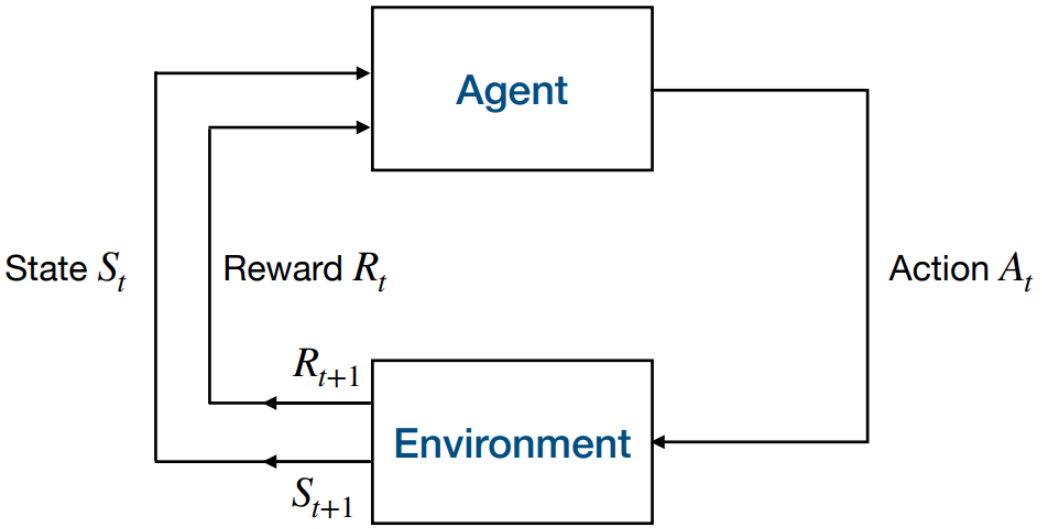


Figure 7: Schema of a Markov Decision Process.

2.2 Rewards and expected returns

Informally, the agent's goal is to maximize the **total amount** of rewards it receives (note how the agent should not maximize the immediate reward, but the cumulative reward). We can formalize this with the **"reward hypothesis"**: *"That all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward)"*.

We will now try to conceptualize the idea of **cumulative rewards** more formally by means of the notion of **expected return G_t**. To do so, we first need to distinguish between two cases:

- **Episodic tasks**, in which we can identify a final step of the sequence of rewards (i.e., in which the interaction between the agent and the environment can be broken into sub-sequences called **episodes**, such as playing a game, repeated tasks, etc.). Each episode ends in a terminal state after T steps, followed by a reset to a standard starting state or to a sample of a distribution of starting states (the next episode will be completely independent from the previous one).
- **Continuous tasks**, in which the agent-environment interaction does not break naturally into identifiable episodes, but goes on continually

without limit (e.g., an ongoing monitoring of a process).

The expected return G_t associated to the selection of an action A_t, assuming that the agent receives over time a sequence of rewards $R_{t+1}, R_{t+2}, R_{t+3}, ...$ is defined as:

- The sum of the future rewards **in the case of episodic tasks**:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + ... + R_t \tag{1}$$

- The weighted sum of the future rewards **in the case of continuing tasks**:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2}$$

Where γ is the **discount rate**, with $0 \leq \gamma \leq 1$. The discount rate is used to give more importance to the rewards that are closer to us in time; this is particularly useful in dynamic environments. The definition of expected return that we used for episodic tasks would in fact be problematic for continuing tasks: the expected return at the time of termination T would be equal to ∞ in some cases, such as when the reward is equal to 1 at each time step. The discount rate determines the "present value of future rewards" (how much future rewards mean to us at the current time): a reward received k time steps in the future is worth γ^{k-1} of what it would be worth if it were received immediately.

Returns at successive time steps are related to each other as follows:

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + ... \\ &= R_{t+1} + \gamma \left(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + ... \right) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned}$$

2.3 Policies and value functions

Almost all reinforcement learning algorithms involve estimating value functions, i.e., functions of states (or of state-action pairs) that estimate how good it is for the agent to be in a given state (or how good it is to perform

a given action in a given state). The notion of "how good" here is defined in terms of future rewards that can be expected, or, to be precise, in terms of expected return.

A **policy** is used to model how the agent will behave based on the previous experience and the rewards (and, consequently, the expected returns) an agent received in the past. Formally, a policy is a mapping from states to probabilities of each possible action (the probability of taking a certain action in a certain state). If the agent is following the policy π at time t, then $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$.

The value function of a state s under a policy π, denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter (the expected return I can have in the future state, considering all the actions I might take from there). For Markov Decision Processes, we can define **the state-value function** v_π for the policy π formally as:

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \;\middle|\; S_t = s, A_t = a \right] \quad \forall s \in \mathcal{S} \tag{3}$$

Where $\mathbb{E}_\pi[\cdot]$ denotes the expected value of a random variable given that the agent follows π and t is any time step. Note that the value of the terminal state, if any, is always 0. The formula above denotes a weighted average of the expected value (it is averaged because the values depend on the probability of a certain action being taken, which is a fraction).

Similar to what we just did, we can define **the action-value function**, i.e., the value of taking an action a in the state s under a policy π, denoted $q_\pi(s, a)$, as the expected return starting from s, taking the action a, and following the policy π thereafter:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \;\middle|\; S_t = s, A_t = a \right] \tag{4}$$

2.4 Choosing the rewards

When we model a real system as a reinforcement learning problem, the most difficult task is selecting the right rewards. Typically, we use negative values for actions that do not help us in reaching our goal, and positive if they do (it is also possible using 0 as a value for actions that do not help us). An alternative is to set the values of the rewards to a negative number until we reach our goal (using 0 as the value when we reach it).

When choosing the rewards, it is very important that **we should not "reward" the intermediate steps or the single actions**. The agent, in fact, always learns to maximize its reward. If we want it to do something for us, we must provide rewards to it in such a way that in maximizing them the agent will also achieve our goals. It is thus critical that the rewards we set up truly indicate what we want accomplished. If we were to give importance to certain sub-goals, the agent might find a way to achieve them without achieving the real goal (e.g., taking the opponent's pieces while playing chess but losing the game). Note that the reward signal is our way of communicating to the agent *what* we want achieved, not *how* (a better place for imparting this kind of prior knowledge would be the initial policy or the initial value function).

Giving rewards to the agent could be a challenging task, as we will see from the two examples that follow. Let us first imagine that we want to create an agent that completes a maze in the least time possible: we could give a reward of -1 for every step it takes inside the maze and 0 for reaching the exit. This could work even if we assume that we only have one episode to base our rewards on. There are situations, though, in which we need additional information to quantify how good an action is for us, like in a game of chess, where we can only assign the rewards at the end of the game (e.g., assigning 1 to every step if we won, -1 if we lost). This is usually called **credit assignment problem** (i.e., the problem of assigning a reward to each step) and a discussion on it can be found in Marvin Minsky's "Steps Towards Artificial Intelligence" paper [Min61].

We now need to think about how we can solve this problem and estimate the value functions v_π and q_π. If the behavior of the Markov Decision Process is known (i.e., the transition probabilities between all the states are known), we could do so by considering all the possible moves, although this poses

strict requirements in terms of prior knowledge and system complexity. A more general option is to estimate them through experience: if an agent follows policy π and maintains an average, for each state encountered, of the actual returns that have followed that state, then the average will converge asymptotically to the state's value, $v_\pi(s)$, as the number of times that state is encountered approaches infinity (these methods are referred to as **Monte Carlo methods** because they involve averaging over many random samples of actual returns). This option is still problematic when it comes to very large number of states, though, as it would involve keeping separate averages for each state individually. In those cases, instead, we will maintain v_π and q_π as parametrized functions, with fewer parameters than the number of states, using approximators such as artificial neural networks.

2.5 Optimal policies and optimal value functions

Solving a reinforcement learning task is roughly equivalent to finding a policy that maximizes the amount of reward over the long run. In finite Markov Decision Processes, there is always at least one policy π that is better than or equal to all the other policies, meaning that its expected return is greater than or equal to that of a different policy π' for all states. More formally:

$$\pi \geq \pi' \text{ if and only if } v_\pi(s) \geq v_{\pi'}(s) \quad \forall s \in \mathcal{S}$$

Although there may be more than one, we denote all the optimal policies with π_*. They share the same state-value function, called the **optimal state-value function**, denoted v_* and defined as:

$$v_*(s) \doteq \max_\pi v_\pi(s) \quad \forall s \in \mathcal{S} \tag{5}$$

This means that, given a state and the value function, the optimal policy is the one that gives us the maximum reward. The same goes for the **optimal action-value function**, denoted q_* and defined as:

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a) \quad \forall s \in \mathcal{S}, a \in \mathcal{A}(s) \tag{6}$$

For the state-action pair (s, a), this function gives the expected return for taking the action a in the state s and thereafter following an optimal policy. Thus, we can write q_* in terms of v_* as follows:

$$q_*(s, a) = \mathbb{E}\left[R_{t+1} + \gamma v_* S_{t+1} \mid S_t = s, A_t = a\right]$$

2.6 Optimality and approximation

2.6.1 Bellman equation

What we are doing is closely related to the issues of automatic control: we both want to have knowledge and control over the evolution of a system.

For any policy π and any state s, the following consistency condition holds between the value of s and the value of its possible successor states:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi\left[G_t \mid S_t = s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s\right] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r \mid s, a)\Big[r + \gamma \mathbb{E}_\pi\left[G_{t+1} \mid S_{t+1} = s'\right]\Big] \\ &= \sum_a \pi(a|s) \sum_{s', r} p(s', r \mid s, a)\Big[r + \gamma v_\pi(s')\Big], \quad \forall s \in \mathcal{S} \end{aligned} \tag{7}$$

What we have in the end is known as the **Bellman equation** for v_π and it states that the value of the start state must equal the (discounted) value of the expected next state, plus the reward expected along the way.

2.6.2 Bellman optimality equation

We can re-write the Bellman equation under the optimal policy, obtaining what is known as the **Bellman optimality equation**:

$$\begin{aligned} q_*(s, a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \;\middle|\; S_t = s, A_t = a\right] \\ &= \sum_{s', r} p(s', r \mid s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right] \end{aligned} \tag{8}$$

Intuitively, the Bellman optimality equation must equal the expected return for the best action from that state.

The Bellman optimality equation is actually a system of equations, one for each state, so if there are n states, then there are n equations in n unknowns. If the dynamics p of the environments are known, then in principle one can solve this system of equation for v_*. Once we have v_* (or q_*), the actions that select the highest value for them in each state will then be the optimal actions. Another way of saying this is that any policy that is **greedy** with respect to the optimal evaluation function is an optimal policy. As a reminder, the term *greedy* is used in computer science to describe any search or decision procedure that selects alternatives based only on local or immediate considerations, without considering the possibility that such a selection may prevent future access to even better alternatives. This is not an issue in the case of Markov Decision processes, though, as they indeed depend only on the current state: **a greedy policy is then optimal both in the short and in the long-term**.

As we were hinting at earlier, it may not always be possible to solve the Bellman optimality equations, both due to the huge number of states (and equations) involved in non-trivial problems and because the state may not be fully observable, or we may not be able to know its dynamics.

2.7 Differences between Reinforcement Learning and other types of learning

As the last thing in this chapter, we try to make sure we have a clear idea of the differences between reinforcement learning and other types of learning.

2.7.1 Reinforcement learning vs Supervised learning

Supervised learning mainly deals with classification: being able to map a certain vectorial input to a set of labels. The supervised algorithms learn by being "fed" labelled examples that must be representative of runtime inputs to work: this clashes with the typical reinforcement learning application scenario of unknown situations.

2.7.2 Reinforcement learning vs Unsupervised learning

Unsupervised learning deals with information provided in unlabeled datasets and tries to find patterns, typically by means of clustering based on a distance function. Reinforcement learning, despite not relying on labels of correct behavior like unsupervised learning, has the goal of maximizing a reward signal instead of trying to find a hidden structure in a dataset.

In general, the most important feature distinguishing reinforcement learning from other types of learning is that it uses training information that **evaluates** the actions taken rather than **instructs** by giving correct actions.

3 Multi-Armed Bandits

In this chapter we want to study the evaluative and explorative aspects of reinforcement learning in a simplified setting that does not involve learning to act in more than one situation.

3.0.1 The k-armed bandit problem

To do so, we introduce the ***k*-armed bandit problem** (a slot machine with k levers), where we must choose between k different options (the k arms of the bandit) and after each choice we receive a reward taken from a **stationary** probability distribution depending on the action we selected. Our objective is to maximize the expected total reward over a certain number of time steps.

As a note, there is a different version of the k-armed bandit problem, known as **contextual bandits**, in which we do consider the state but assume that it does not depend on the previous actions. This problem is at the basis of ad-serving platforms.

Let us now see things from a more formal point of view: in our k-armed bandit problem, each of the k actions has an expected or mean reward given that that action is selected; let us call this the *value* of that action. We denote the action selected on time step t as A_t, and the corresponding reward as R_t. Our goal is then to maximize the following:

$$\mathbb{E}\left[G_t \mid S_t = s, A_t = a\right] \doteq \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \;\middle|\; S_t = s, A_t = a\right]$$
$$= \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \;\middle|\; S_t = s, A_t = a\right]$$

Since one of the assumptions of this problem is that the state does not depend on the actions taken, we can think of it as a constant $\bar{s}$ and substitute it in the formula above, obtaining:

$$\mathbb{E}\left[G_t \mid S_t = \bar{s}, A_t = a\right] \doteq \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \;\middle|\; S_t = \bar{s}, A_t = a\right]$$
$$= \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \;\middle|\; S_t = \bar{s}, A_t = a\right]$$

The value of an arbitrary action a denoted as $q_*(a)$ is the expected reward given that a is selected. More formally:

$$q_*(a) \doteq \mathbb{E}\left[R_t \mid A_t = a\right]$$

(Note how the expected reward does not include the state, as it is always the same). If we disregard the trivial case in which we already know the value of each action, we need to play to be able to estimate the value of the various actions: we denote the estimated value of action a at time step t as $Q_t(a)$ and we want it to be as close as possible to $q_*(a)$.

If we maintain the estimates of each action value, then at any time step there will be at least one action whose value is the greatest (we refer to this action as the **greedy action**). When we select one of these actions, we are **exploiting our current knowledge** of the value of the actions. If instead we choose one of the nongreedy actions, then we are **exploring**, because this enables us to improve our estimate of the nongreedy action's value. Exploitation maximizes the expected reward on the step, but by exploring, we may obtain a greater total reward in the long run (this is particularly true if our estimates have high uncertainty). In any specific case, whether it is better to explore or exploit depends in a complex way on the precise values of the estimates, uncertainties, and the number of remaining steps.

3.1 Evaluating action-value methods

Now that we have understood the need for estimating the values of actions and for using the estimates to make action selection decisions (which we collectively call **action-value methods**), we must look at some methods to do so.

In the case of the k-armed bandits problem, the value of an action is the mean reward when that action is selected, since we only have one state. A possible way to estimate this is by averaging the rewards we actually received:

$$\begin{aligned} Q_t(a) &\doteq \frac{\text{sum of rewards when an action } a \text{ is taken prior to time } t}{\text{number of times an action } a \text{ is taken prior to time } t} \\ &= \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}} \end{aligned} \tag{9}$$

Where $\mathbb{1}_{\text{predicate}}$ denotes the random variable that is 1 if predicate is true and 0 if it is not. If the denominator is zero, then we instead define $Q_t(a)$ as some default value, such as 0. As the denominator goes to infinity, by the law of large numbers, $Q_t(a)$ converges to $q_*(a)$. We call this the **sample-average method** for estimating action values because each estimate is an average of the sample of relevant rewards.

When it comes to selecting actions, the simplest action selection rule is to select one of the actions with the highest estimated value, that is, one of the greedy actions. If there is more than one greedy action, then a selection is made among them in some arbitrary way, perhaps randomly. We can formalize this **greedy action selection method** as:

$$A_t \doteq \operatorname*{argmax}_a Q_t(a) \tag{10}$$

Where argmax_a denotes the action a for which the expression that follows is maximized (with ties broken arbitrarily). The greedy action selection always exploits the current knowledge to maximize the immediate reward, disregarding exploration. To include it, we can add a small probability ϵ where the next action is selected randomly from all the actions with equal probability. This is called the **ϵ-greedy selection rule** and it has the advantage that

in the limit, as the number of steps increases, every action will be sampled an infinite number of times, ensuring that all the $Q_t(a)$ converge to their respective $q_*(a)$ (these, however, are just asymptotic guarantees and say little about the practical effectiveness of these methods).

3.2 Incremental implementation

We now move to the question of how the methods that we have seen before can be computed in a computationally efficient manner, with constant memory and constant per-time-step computation.

To simplify the notation, we will concentrate on a single action a. Let $R_i(a)$ denote the reward received after the i-th selection of the action a, and let Q_n denote the estimate of its action value after it has been selected $n-1$ times. We can now write:

$$Q_n(a) \doteq \frac{R_1(a) + R_2(a) + ... + R_{n-1}(a)}{n-1}$$

A trivial implementation of this would be to maintain a record of all the rewards and then perform this computation whenever the estimate value is needed. However, both memory and computation requirements would grow over time as more and more rewards are received. This can be avoided by considering the following steps:

$$
\begin{aligned}
Q_{n+1} &= \frac{1}{n}\sum_{i=1}^{n} R_i \\
&= \frac{1}{n}\Big(R_n + \sum_{i=1}^{n-1} R_i\Big) \\
&= \frac{1}{n}\Big(R_n + (n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} R_i\Big) \\
&= \frac{1}{n}\Big(R_n + (n-1)Q_n\Big) \\
&= \frac{1}{n}\Big(R_n + nQ_n - Q_n\Big) \\
&= Q_n + \frac{1}{n}\Big[R_n - Q_n\Big]
\end{aligned}
$$

This type of update rule is quite common in reinforcement learning and follows this general formula:

$$\text{NewEstimate} \leftarrow \text{OldEstimate} + \text{StepSize}(\text{Target} - \text{OldEstimate}) \qquad (11)$$

The expression (Target − OldEstimate) is usually defined as the **error** in the estimate. It is reduced by taking a step towards the Target, which is presumed to indicate a desirable direction in which to move (though it may be noisy).

3.3 Tracking a nonstationary problem

The averaging method we just discussed is appropriate for stationary bandit problems, where the distribution of the rewards does not change over time. Many problems, however, fall in the nonstationary category and in those cases, it makes sense to give more weight to recent rewards, rather than long-gone ones. One of the most popular ways of dealing with these problems is using a **constant step-size parameter** $\boldsymbol{\alpha}$ (like $\frac{1}{n}$ in the formula above) in the range $]0, 1]$ as such:

$$Q_{n+1} \doteq Q_n + \alpha \left[R_n - Q_n \right] \tag{12}$$

This results in Q_{n+1} being a weighted average of past rewards and the initial estimate Q_1:

$$\begin{aligned}
Q_{n+1} &= Q_n + \alpha \Big[R_n - Q_n \Big] \\
&= \alpha R_n + (1-\alpha) Q_n \\
&= \alpha R_n + (1-\alpha)[\alpha R_{n-1} + (1-\alpha) Q_{n-1}] \\
&= \alpha R_n + (1-\alpha)\alpha R_{n-1} + (1-\alpha)^2 Q_{n-1} \\
&= \alpha R_n + (1-\alpha)\alpha R_{n-1} + (1-\alpha)^2 \alpha R_{n-2} + ... + (1-\alpha)^{n-1} \alpha R_1 + (1-\alpha)^n Q_1 \\
&= (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha (1-\alpha)^{n-i} R_i
\end{aligned} \tag{13}$$

This is called a weighted average since the sum of the weights $(1-\alpha)^n + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} = 1$, as we can see here:

$$
\begin{aligned}
&(1-\alpha)^n + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} \\
&= (1-\alpha)^n + \alpha(1-\alpha)^n \sum_{i=1}^{n} (1-\alpha)^{-i} \\
&= (1-\alpha)^n + \alpha(1-\alpha)^n \Big(-1 + \sum_{i=0}^{n} (1-\alpha)^{-i} \Big) \\
&= (1-\alpha)^{n+1} + \alpha(1-\alpha)^n \sum_{i=0}^{n} (1-\alpha)^{-i} \\
&= (1-\alpha)^{n+1} + \alpha(1-\alpha)^n \frac{\alpha + (1-\alpha)^{-n} - 1}{\alpha} \\
&= (1-\alpha)^{n+1} + (1-\alpha)^n \Big(\alpha + (1-\alpha)^{-n} - 1 \Big) \\
&= (1-\alpha)^{n+1} + \alpha(1-\alpha)^n + 1 - (1-\alpha)^n \\
&= (1-\alpha)^{n+1} + (1-\alpha)^n(\alpha - 1) + 1 \\
&= (1-\alpha)^{n+1} - (1-\alpha)^{n+1} + 1 = 1
\end{aligned}
$$

The quantity $1-\alpha$ is less than 1, and thus the weight given to R_i decreases as the number of rewards increases. In fact, the weight decreases exponentially according to the exponent on $1-\alpha$; accordingly, this is sometimes called an **exponential recency-weighted average**.

3.4 Optimistic initial values

All the methods that we have discussed until now depend to some extent on the initial action-value estimates (they are **biased** by their initial estimates). For the sample-average methods, the bias disappears once all actions have been selected at least once, but for methods with a constant α the bias is permanent (although decreasing over time as given by the equation (13)). In practice, this kind of bias is usually not a problem and can sometimes be helpful, especially to supply some prior knowledge or to encourage exploration. The downside is that the initial estimates become, in effect, a set of parameters that must be picked by the user.

Choosing a wildly optimistic initial value encourages action-value methods

to explore: whichever actions are initially selected, the reward is less than the starting estimates; the learner will then switch to other actions, being "disappointed" with the rewards it is receiving. The result is that all actions are tried several times before the value estimates converge, making the system do a fair amount of exploration even with a fully greedy action selection strategy. Initially, the optimistic method performs worse because it explores more, but eventually it performs better as its exploration decreases with time. We call this technique for encouraging exploration **optimistic initial values**. It is a simple trick that can be quite effective on stationary problems, while it is not well suited to nonstationary problems, as its drive for exploration is inherently temporary (any method that focuses on the initial conditions is unlikely to help with the general nonstationary case).

3.5 Upper-Confidence-Bound Action Selection

Exploration is needed because there is always uncertainty about the accuracy of the action-value estimates. ϵ-greedy action selection forces non-greedy (exploratory) actions to be tried, but indiscriminately, with no preference for those that are nearly greedy or particularly uncertain. It would be better to select among the non-greedy actions according to their potential for actually being optimal, taking into account both how close their estimates are to being maximal and the uncertainties in those estimates. One effective way of doing this is to select actions according to the following formula:

$$A_t \doteq \underset{a}{\operatorname{argmax}} \left[Q_t(a) + c + \sqrt{\frac{\ln t}{N_t(a)}} \right] \tag{14}$$

Where t is the time at which action A_t is taken, $N_t(a)$ denotes the number of times that action a has been selected prior to time t and the number $c > 0$ controls the degree of exploration. The idea of this **upper confidence bound (UCB)** action selection is that the square-root term is a measure of the uncertainty or variance of the estimate of the actual value of action a. The quantity being maxed over is thus a sort of upper bound on the possible true value of action a, with c determining the confidence level. Each time a is selected, the uncertainty is presumably reduced: $N_t(a)$ increments, and, as it appears in the denominator, the uncertainty term decreases. On the other hand, each time an action other than a is selected, t increases but $N_t(a)$ does

not; because t appears in the numerator, the uncertainty estimate increases. The use of the natural logarithm means that the increases get smaller over time, but are unbounded; all actions will eventually be selected, but actions with lower value estimates, or that have already been selected frequently, will be selected with decreasing frequency over time.

3.6 Contextual bandits

Multi-armed bandits are used when the selection of the action does not depend on the state, but for many application (e.g., ads on a webpage), the state is very important. When the action selection depends on the state, but not on its previous history, we speak of **contextual bandits**. They are examples of **associative search tasks**, as they involve both trial-and-error learning to **search** for the best actions, and **association** of these actions with the situation in which they are best (e.g., if we are on a website about sportscars, we will be served ads about cars and not about flowers). Contextual bandits are still not a "full" reinforcement learning problem, as actions do not affect the future state, but only the immediate reward.

4 Monte Carlo Methods

Monte Carlo methods are ways of solving the reinforcement learning problem based on averaging sample returns. To ensure that well-defined returns are available, we will focus on episodic tasks. That is, we assume experience is divided into episodes, and that all episodes eventually terminate no matter what actions are selected. Only on the completion of an episode are value estimates and policies changed. Monte Carlo methods can thus be incremental in an episode-by-episode sense, but not in a step-by-step (online) sense.

Monte Carlo methods sample and average **returns** for each state-action pair much like the bandit methods we explored earlier sample and average **rewards** for each action. The main difference is that now there are multiple states, each acting like a different (but interrelated) bandit problem. That is, **the return after taking an action in one state depends on the actions taken in later states in the same episode** (it is the **average expected cumulative reward**).

From this point on, we will assume that we do not have full knowledge

of the underlying Markov Decision Process. We are then entering the **full reinforcement learning problem**, in which the underlying dynamics and characteristics of the system are unknown (e.g., robot exploration) or because the system is too complex (e.g., games). In these types of problems, we will have to **learn the value functions from sample returns**.

We will consider three problems:

1. The **prediction problem**, where we want to estimate v_π and q_π for a fixed policy π. We are given a fixed policy (that is, we know how we play) and we try to estimate the expected cumulative reward.
2. The **policy improvement problem**, where we still try to estimate v_π and q_π, but this time we also try to improve the policy π. We try to estimate the value of each state and we use this information to play better.
3. The **control problem**, where we try to estimate an optimal policy π_* (e.g., we try to find the best way to play a game; the best way to build a datacenter to minimize energy consumption; ...).

4.1 Monte Carlo Prediction

We begin by considering Monte Carlo methods for learning the state-value function for a given policy. Before doing so, we must have clear in our minds that **the value of a state is the expected return (expected cumulative future discounted reward) starting from that state**. An obvious way to estimate it from experience, then, is simply to average the returns observed after visits to that state. As more returns are observed, the average should converge to the expected value. This idea underlies all Monte Carlo methods.

More formally: we want to estimate $v_\pi(s)$, the value of a state s under policy π, given a set of episodes obtained by following π and passing through s. Each occurrence of state s in an episode is called a **visit** to s. The same state may be visited multiple times in a certain episode: let us call the first time this happens the **first visit to *s***. The **first-visit Monte Carlo method** estimates $v_\pi(s)$ as the average of the returns following the *"first visits"* to s, whereas the **every-visit Monte Carlo method** averages the returns following *all visits* to s.

4.1.1 First-visit Monte Carlo Prediction

The algorithm for the *first-visit Monte Carlo Prediction* is shown here:

Algorithm 1: First-visit Monte Carlo Prediction

Input: a policy π to be evaluated
Initialize:
$V(s) \in \mathbb{R}$, arbitrarily, $\forall s \in \mathcal{S}$
$Returns(s) \leftarrow$ an empty list, $\forall s \in \mathcal{S}$
Loop *forever (for each episode)* **:**
Generate an episode following π: $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$;
Loop *for each step of the episode,* $t = T-1, T-2, ..., 0$ **:**
$G \leftarrow \gamma G + R_{t+1}$;
Unless S_t *appears in* $S_0, S_1, ..., S_{t-1}$ **:**
Append G to $Returns(s)$;
$V(S_t) \leftarrow average(Returns(S_t))$;

This means generating an episode where we play following policy π (the set of probabilities associated to each action) and analyzing the "trajectory" that we followed during the game in a backwards manner. During this phase we will calculate the values for all the states that we traversed and the average cumulative reward we can expect from them.

4.1.2 Every-visit (multi-visit) Monte Carlo Prediction

If we remove from the *first-visit Monte Carlo Prediction* algorithm the check on whether a state S_t has already occurred, we obtain the **every-visit Monte Carlo Prediction**, which also converges to $v_\pi(s)$ as the number of visits to a certain state s goes to infinity.

4.2 Monte Carlo Estimation of Action Values

The estimation of a state value makes sense when we have a model of the system: with it, in fact, state values alone are sufficient to determine a policy; one simply looks ahead one step and chooses whichever action leads to the best combination of reward and next state. Without a model, however, it is necessary to estimate the value of each action in order for the value to be useful in suggesting a policy.

The **policy evaluation problem** for action values is to estimate $q_\pi(s, a)$, the expected return when starting in state s, taking action a and then following policy π. The methods for the Monte Carlo estimation of action values are essentially the same as those presented for state values, except that now we talk about **visits to the state-action pair** rather than to a state. A state-action pair s, a is said to be visited in an episode if ever the state s is visited and the action a is taken in it. The first-visit Monte Carlo method averages the returns following the first time in each episode that the state was visited and the action was selected.

The only complication is that many state-action pairs may never be visited: if π is a deterministic policy (to a certain state corresponds one and only one action), in fact, we would observe returns only for one of the actions of each state. With no returns to average, the Monte Carlo estimates of the other actions would subsequently not improve with experience. In order to compare alternatives, we then need to estimate the value of all the actions from each state, not only the one that is favored by our policy. This is the general problem of **maintaining exploration**.

For policy evaluation to work for action values, we must assure **continual exploration**. One way to do this is by specifying that the episodes start in a state-action pair, and that every pair has a nonzero probability of being selected as the start. This guarantees that all state-action pairs will be visited an infinite number of times in the limit of an infinite number of episodes; we call this the assumption of **exploring starts**. This assumption is sometimes useful, but it cannot be relied upon in general (we may need to enumerate all the states, or they may not be valid): the most common alternative approach is to assure that all state-action pairs are encountered. This means **not following the current policy** (for example by using a stochastic policy). In other words, the exploration is not performed **on-policy**, but **off-policy** (various methods are possible, like *Off-policy Predictions via Importance Sampling*).

4.3 Monte Carlo Policy Improvement and Monte Carlo Control

So far, we have assumed that the policy π was fixed, like in the case of a statically defined set of probabilities for each action (the *prediction problem*).

However, by using methods typically known as **Monte Carlo Control**, we can improve the policy and reach the optimal one. In this section we will focus on **on-policy methods**, meaning **methods that attempt to evaluate or improve the policy that is used to make decisions**.

The overall idea is to proceed according to the **Generalized Policy Iteration (GPI)**, in which we maintain both an approximate policy and an approximate value function. The value function is repeatedly altered to more closely approximate the value function for the current policy, and the policy is repeatedly improved with respect to the current value function. As we can imagine, these two changes work against each other to some extent, as each creates a "moving target" for the other, but together they cause both policy and value function to approach optimality.

With this method, we perform alternating complete steps of policy evaluation and policy improvement, beginning with an arbitrary policy π_0 and ending with the optimal policy and action-value function as such:

$$\pi_0 \xrightarrow{E} q_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} q_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} ... \xrightarrow{I} \pi_* \xrightarrow{E} q_* \tag{15}$$

Where $\xrightarrow{E}$ denotes a complete policy evaluation and $\xrightarrow{I}$ denotes a complete policy improvement. *Policy evaluation* is performed as we mentioned in previous sections: many episodes are experienced, with the approximate action-value function approaching the true function asymptotically. *Policy improvement* is done by making the policy greedy with respect to the current value function. Note how, since we have an action-value function, we do not need any model to construct the greedy policy: for any action-value function q, the corresponding greedy policy is the one that for each $s \in \mathcal{S}$, deterministically chooses an action with maximal action-value as such:

$$\pi(s) \doteq \operatorname*{argmax}_a q(s, a)$$

Policy improvement can then be done by constructing each π_{k+1} as the greedy policy with respect to q_{π_k}. The **policy improvement theorem** assures us that each π_{k+1} is uniformly better than π_k or just as good (in which case they are both optimal policies). This confirms that **the overall process converges to the optimal policy and optimal value function**.

4.3.1 Monte Carlo Control algorithm with Exploring Starts

Since we need to guarantee that all actions are selected "infinitely often", in some cases we can do so by means of **exploring starts**. A possible implementation of the **Monte Carlo Control algorithm with Exploring Starts** is shown in the box below:

Algorithm 2: Monte Carlo Control algorithm with Exploring Starts

Initialize:

$\pi(s) \in \mathcal{A}(s)$ (arbitrarily), $\forall s \in \mathcal{S}$
$Q(s,a) \in \mathbb{R}$ (arbitrarily), $\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$
$Returns(s,a) \leftarrow$ an empty list, $\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$

Loop *forever (for each episode)* **:**

Choose a starting state and action $S_0 \in \mathcal{S}, A_0 \in \mathcal{A}(S_0)$ randomly, such that all pairs have probability > 0;
Generate an episode starting with S_0, A_0 and following π: $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$;
$G \leftarrow 0$;
Loop *for each step of the episode,* $t = T-1, T-2, ..., 0$ **:**
$G \leftarrow \gamma G + R_{t+1}$;
Unless *the pair* S_t, A_t *appears in* $S_0, A_0, S_1, A_1, ..., S_{t-1}, A_{t-1}$ **:**
Append G to $Returns(S_t, A_t)$;
$Q(S_t, A_t) \leftarrow average(Returns(S_t, A_t))$;
$\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t, a)$;

With this algorithm, all the returns for each state-action pair are accumulated and averaged, irrespective of what policy was in force when they were observed. It is easy to see that it cannot converge to any suboptimal policy: if it did, then the value function would eventually converge to the value function for that policy, and that in turn would cause the policy to change. **Stability is achieved only when both the policy and the value function are optimal**.

4.3.2 Monte Carlo Control without Exploring Starts

As we mentioned earlier on, the assumption of exploring starts is often unlikely to be valid; a possible alternative is to use **soft policies**, meaning that each action in each state has a nonzero probability of being chosen ($\pi(a|s) > 0, \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$), but gradually shifts closer and closer to a

deterministic, optimal policy. The ϵ-greedy policy we have already seen previously is an example of ϵ-soft policy, in which all non-greedy actions have a (small) probability $\frac{\epsilon}{|\mathcal{A}(s)|}$ of being selected.

Contrarily to what we did under the assumption of exploring starts, we cannot simply improve the policy by making it greedy with respect to the current value function, as it would prevent any exploration of the nongreedy actions. Fortunately, the Generalized Policy Iteration method does not require that the policy be taken all the way to a greedy policy, but only that it be moved *towards* a greedy policy. That any ϵ-greedy policy with respect to q_π is an improvement over any ϵ-soft policy π is once again assured by the policy improvement theorem.

The **on-policy first-visit Monte Carlo control (for ϵ-soft policies) algorithm** is presented in the box below:

Algorithm 3: Monte Carlo Control algorithm with Exploring Starts

Parameters: small $\epsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ϵ-soft policy

$Q(s, a) \in \mathbb{R}$ (arbitrarily), $\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$

$Returns(s, a) \leftarrow$ an empty list, $\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$

Loop *forever (for each episode)* **:**

Generate an episode following π: $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$;

$G \leftarrow 0$;

Loop *for each step of the episode,* $t = T-1, T-2, ..., 0$ **:**

$G \leftarrow \gamma G + R_{t+1}$;

Unless *the pair* S_t, A_t *appears in* $S_0, A_0, S_1, A_1, ..., S_{t-1}, A_{t-1}$ **:**

Append G to $Returns(S_t, A_t)$;

$Q(S_t, A_t) \leftarrow average(Returns(S_t, A_t))$;

$A^* \leftarrow \text{argmax}_a\, Q(S_t, a)$ (with ties broken arbitrarily);

$\forall a \in \mathcal{A}(S_t)$:

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(S_t)|} & \text{if } a = A^* \\ \frac{\epsilon}{|\mathcal{A}(S_t)|} & \text{if } a \neq A^* \end{cases}$$

5 Temporal Difference Learning

Monte Carlo methods introduced us to ways of learning from raw experience without a model of the environment's dynamics, iteratively generating episodes and analyzing their returns once they have been played out. Temporal Difference methods behave similarly, but overcome the limit of having to wait for the whole episode to end, enabling step-by-step learning. Let us analyze this in a bit more detail by first focusing on the *prediction problem*.

Following the structure (11) we saw in chapter 3, we can schematize a simple every-visit Monte Carlo method suitable for nonstationary environment as:

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

Where $V(S_t)$ is the state-value function for state S_t, α is a constant step-size parameter and G_t is the actual **return** following time t (that we can only know at the end of the episode). Temporal Difference methods, on the contrary, only need to wait until the next time step: at time $t+1$ they immediately form a target and make a useful update using the observed **reward** R_{t+1} and the estimate $V(S_{t+1})$. The simplest Temporal difference method makes the update:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right] \tag{16}$$

Immediately on transition to S_{t+1} and receiving R_{t+1}. In effect, the target for the Monte Carlo update is G_t, whereas the target for the Temporal Difference update is $R_{t+1} + \gamma V(S_{t+1})$. This Temporal Difference method is called **TD(0)**, or **one-step TD**, because it is a special case of the **TD($\boldsymbol{\lambda}$)** and n-step TD methods that we will see later on.

5.1 TD(0)

An implementation of the TD(0) algorithm is shown in the box below:

Algorithm 4: Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Parameters: step size $\alpha \in]0, 1]$
Initialize: $V(s)$, $\forall s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$
Loop *for each episode* **:**
    Initialize S;
    Loop *for each step of the episode* **:**
        $A \leftarrow$ action given by π for S;
        Take action A, observe R, S';
        $V(S) \leftarrow V(S) + \alpha\,[R + \gamma V(S') - V(S)]$;
        $S \leftarrow S'$;
    until S is terminal

We can note that the quantity in the square brackets of the TD(0) update is a sort of error that measures the difference between the estimated value of S_t and the better estimate given by $R_{t+1} + \gamma V(S_{t+1})$. This quantity is called **TD error $\boldsymbol{\delta_t}$** and it represents the error in the estimate made at time t, as defined by the following:

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{17}$$

Since δ_t depends on the next state and the next reward, the TD error at time t will only be available at time $t + 1$.

5.2 Advantages and theoretical bases of TD methods

Given what we have seen, we can already understand that Temporal Difference methods have advantages over both methods that require a model of the environment (such as Dynamic Programming) and Monte Carlo methods, as they cannot be applied in an online, fully incremental fashion like TD. This is often crucial because we may not have a complete model of the environment or the tasks that we are studying might not be divisible into episodes (such as the case of continuing tasks).

One may then ask if these methods still guarantee convergence to the correct values. Luckily, this is the case, as for any fixed policy π, TD(0) has been proven to converge to v_π with probability 1 (under stochastic approximation

conditions) if we decrease over time the value of the step size parameter α, while only on average if α is statically selected to be sufficiently small.

5.3 SARSA: on-policy Temporal Difference Control

After seeing how to estimate the state-value function with TD(0) (the *prediction problem*), we will now move on to the *control problem* by introducing SARSA.

SARSA is an on-policy control method (it aims to estimate and improve the policy used to make decisions) and, as such, it requires us to estimate the **action-value function $q_\pi(s, a)$** for the current behavior policy[3] π and for all the states s and actions a, rather than the state-value function v_π as we did in TD(0) (in short, we want to learn the values of the state-action pairs rather than the values of the single states). Formally, these cases are identical: they are both Markov chains with a reward process and the theorems that assured the convergence of state values under TD(0) also apply to the corresponding algorithm for action values:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right] \tag{18}$$

This update is done after every transition from a nonterminal state S_t. If S_{t+1} is terminal, then $Q(S_{t+1}, A_{t+1}) \doteq 0$. The name of this rule comes from the quintuple used to perform this update: $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$.

It is straightforward to design an on-policy control algorithm based on the SARSA prediction method. As in all on-policy methods, we continually estimate q_π for the behavior policy π, and at the same time change π towards greediness with respect to q_π. The **SARSA (on-policy TD control) algorithm for estimating Q values** is presented in the box below:

[3]Since we are talking about on-policy methods, the policy also determines our behavior, thus "behavior policy".

Algorithm 5: SARSA (on-policy TD control) for estimating $Q \approx q_*$

Parameters: step size $\alpha \in]0, 1]$, small $\epsilon > 0$

Initialize:

$Q(s, a)$, $\forall s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop *for each episode* **:**

Initialize S;

Choose A from $\mathcal{A}(S)$ using policy derived from Q (e.g., ε-greedy);

Loop *for each step of the episode* **:**

Take action A, observe R, S';

Choose A' from $\mathcal{A}(S')$ using policy derived from Q (e.g., ε-greedy);

$Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma Q(S', A') - Q(S, A)\right]$;

$S \leftarrow S'$;

$A \leftarrow A'$;

until S is terminal

5.4 Q-Learning: off-policy Temporal Difference Control

On-policy methods like SARSA are not the only ones available to tackle the control problem. One of the early breakthroughs in reinforcement learning, in fact, was the development in 1989 of an off-policy Temporal Difference control algorithm known as **Q-Learning** and defined by:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right] \tag{19}$$

As we can see, the update formula does not consider the action A_{t+1} that we chose at time $t + 1$, but a (possibly) different one: a, the one that has the highest Q value. In this case, the **learned action-value function Q directly approximates the optimal action-value function q_* independent of the policy being followed**. The policy still has an effect in that it determines which state-action pairs are visited and updated; however, all that is required for correct convergence is that all pairs continue to be updated. Under this assumption and a variant of the usual stochastic approximation conditions on the sequence of step-size parameters, Q has been shown to converge with probability 1 to q_*. More information can be found in [WD92], while an implementation is shown in the box below:

Algorithm 6: Q-Learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Parameters: step size $\alpha \in]0, 1]$, small $\epsilon > 0$

Initialize:

$Q(s, a)$, $\forall s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop *for each episode* **:**

- Initialize S;
- **Loop** *for each step of the episode* **:**
  - Choose A from $\mathcal{A}(S)$ using policy derived from Q (e.g., ε-greedy);
  - Take action A, observe R, S';
  - $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$;
  - $S \leftarrow S'$;
- until S is terminal

5.5 Summary

The methods that we have seen in this chapter (SARSA and Q-Learning) are some of the most used ones in the reinforcement learning field, just like Decision Trees in Machine Learning and Min-Max in heuristics. They are often referred to as **tabular methods** since the state-action space must fit in a table in which every row corresponds to a state-action entry. This raises a question: what happens if we cannot fit all the entries in a table because they are too many? In that case we will need **function approximators rather than tables**: functions that, given in input the state (or the state and action), output the value function for the state (or the state and action). For these more complex cases, we will need **deep reinforcement learning**.

References

[Tur50] A. M. Turing. "I.—Computing Machinery and Intelligence". In: *Mind* LIX.236 (Oct. 1950), pp. 433–460. ISSN: 0026-4423. DOI: 10.1093/mind/LIX.236.433. eprint: https://academic.oup.com/mind/article-pdf/LIX/236/433/30123314/lix-236-433.pdf. URL: https://doi.org/10.1093/mind/LIX.236.433.

[Min61] M. Minsky. "Steps toward Artificial Intelligence". In: *Proceedings of the IRE* 49.1 (1961), pp. 8–30. DOI: 10.1109/JRPROC.1961.287775.

[WD92] Christopher J. C. H. Watkins and Peter Dayan. "Q-learning". In: *Machine Learning* 8.3 (May 1992), pp. 279–292. ISSN: 1573-0565. DOI: 10.1007/BF00992698. URL: https://doi.org/10.1007/BF00992698.