Autonomous and Adaptive Systems
course notes
University of Bologna

prof. Mirco Musolesi

AY. 2020-2021

Alessandro Pomponio
alessandro.pomponio2@studio.unibo.it

June 7, 2021

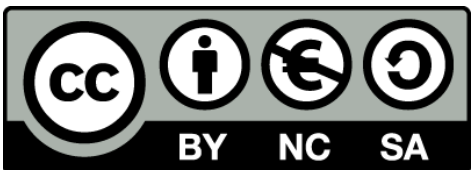
Preface

The 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`
- Ian Goodfellow, Yoshua Bengio and Aaron Courville's "Deep Learning" book, available here: `https://www.deeplearningbook.org/`

This work has not been checked or endorsed in any way by the people mentioned above.

While these notes try to be complete and self-contained as much as possible, the reader is highly encouraged to follow the lectures and read the slides provided by prof. Musolesi, especially given the ever-evolving nature of the matters at hand. It is also important to note that the coding parts related to OpenAI Gym, Keras and Tensorflow will not be covered in these notes.



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

Contents

List of Algorithms

List of Figures

Chapter 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

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

of learning machines, that 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 "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. We will refer to agents that are physically situated as **robots**, while we will call **software agents** or **bots** those that are not.

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.

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 without knowing their effect; from a practical point of view, it then 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 also 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 1.1, along with a few examples of agents in figure 1.2.

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).

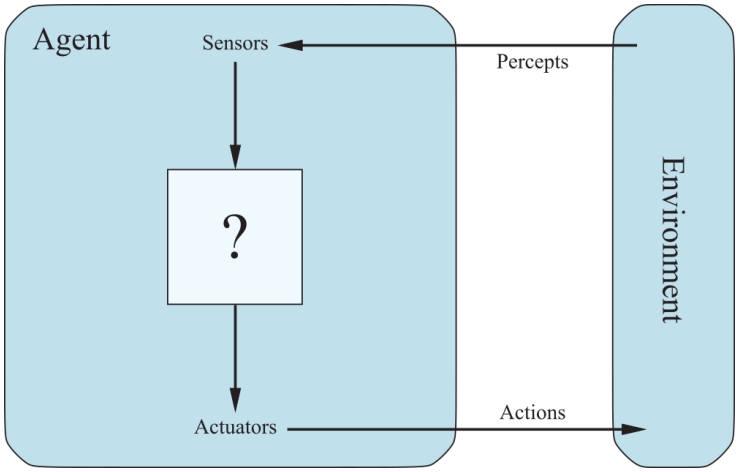


Figure 1.1: Schema of the interaction between an agent and the environment.

Source: Russel and Norvig, 2020

	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 1.2: Examples of agents and their characteristics.

Source: Prof. Mirco Musolesi

- **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).
- **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 on 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; etc.).
- **Discrete vs continuous**: the environment may be represented by means of discrete or continuous values (e.g., a switch can be ON or OFF; the wind speed might be 14.6 km/h from N/E; etc.).
- **Single-agent or multi-agent**: there may be multiple agents in the environment, and we might 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**.

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 (figure 1.3) 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).

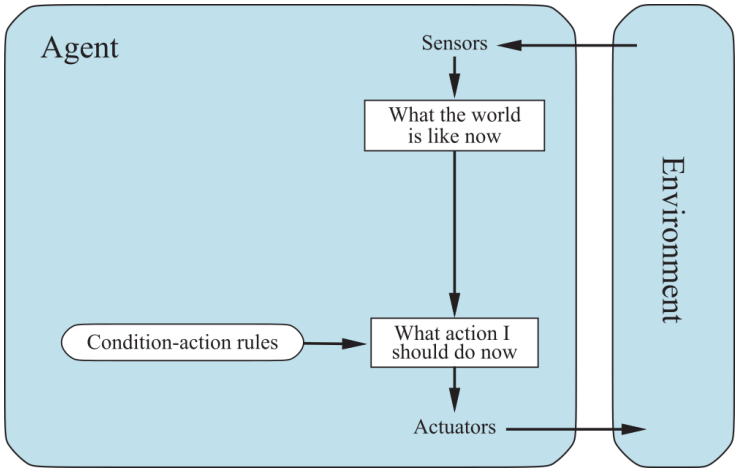


Figure 1.3: Schema of a simple reflex agent.

Source: Russel and Norvig, 2020

1.4.2 Model-based reflex agents

Model-based agents (figure 1.4) 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.

1.4.3 (Model-based) Goal-based agents

Goal-based agents (figure 1.5) 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.

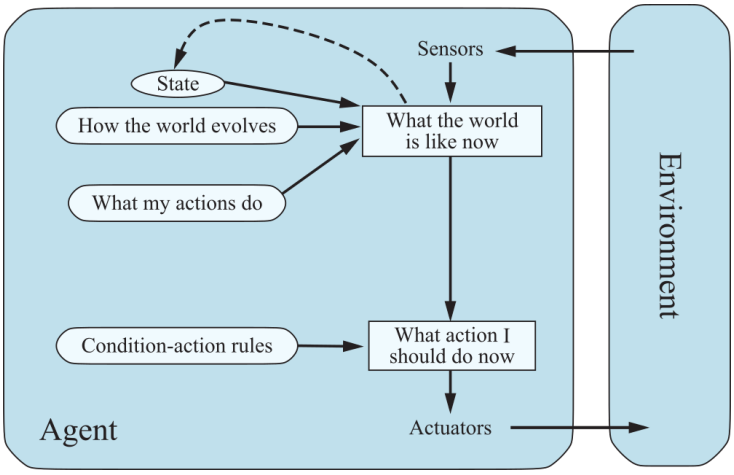


Figure 1.4: Schema of a model-based reflex agent.
Source: Russel and Norvig, 2020

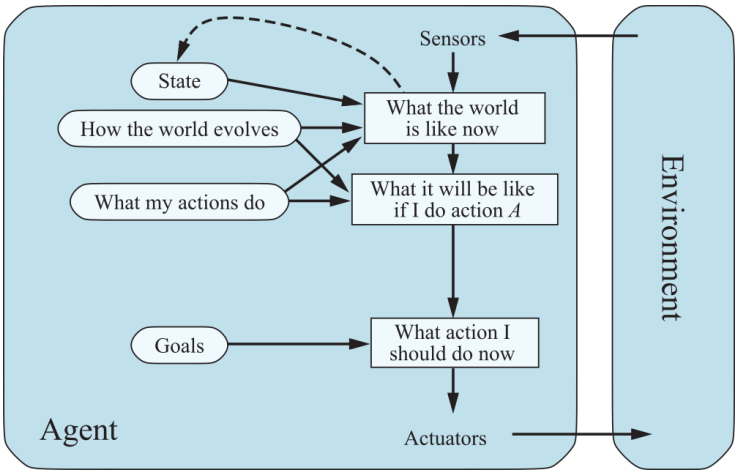


Figure 1.5: Schema of a goal-based agent.
Source: Russel and Norvig, 2020

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 1.6)

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

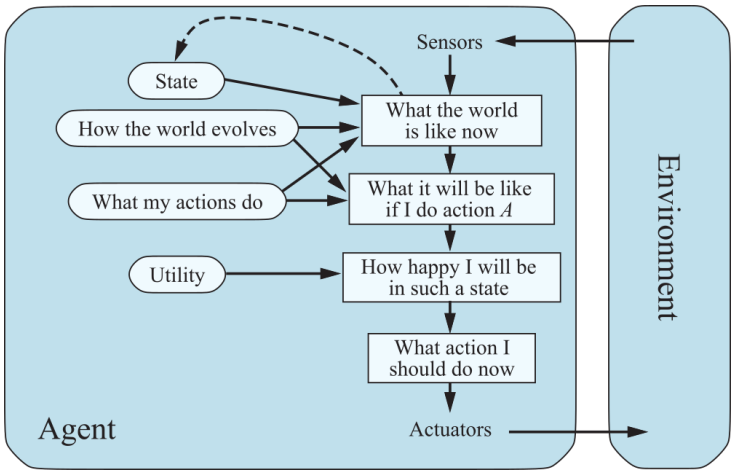


Figure 1.6: Schema of a utility-based agent.

Source: Russel and Norvig, 2020

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

1.5 Learning

As we 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 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"*.

Chapter 2

Introduction to Reinforcement Learning

The Merriam-Webster dictionary defines learning as the *"modification of a behavioral tendency by experience"*[1]. Interacting with the external world to learn, in fact, is an idea that is at the basis of nearly all theories of learning and intelligence. In the artificial intelligence world, this approach is leveraged with success in 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 out. 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

[1] `https://www.merriam-webster.com/dictionary/learning`

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)*.

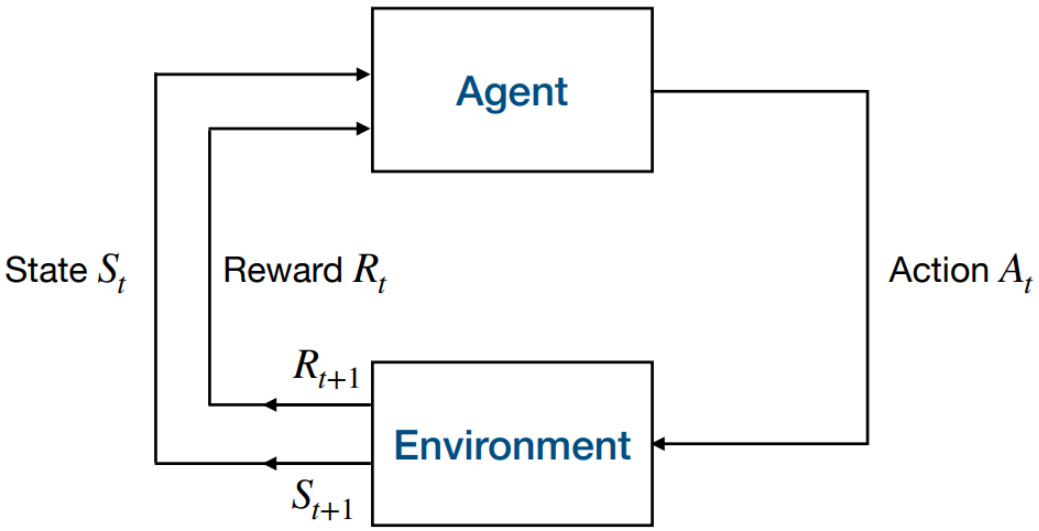


Figure 2.1: Schema of a Markov Decision Process.

Source: Prof. Mirco Musolesi

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*

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

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 future rewards** in the case of **episodic tasks**:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T \tag{2.1}$$

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

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.2}$$

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 would be equal to ∞ (in the limit of an infinite number of time steps) in some cases, such as when the reward is equal to 1 at each time step. The discount rate then 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 an agent received in the past (and, consequently, the expected returns). Formally, a policy is a mapping from states to probabilities of taking 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:

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \;\middle|\; S_t = s, A_t = a \right] \quad \forall s \in \mathcal{S} \tag{2.3}$$

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

$\boldsymbol{q_\pi(s, a)}$, as the expected return starting from s, taking the action a, and following the policy π thereafter:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \;\middle|\; S_t = s, A_t = a \right] \tag{2.4}$$

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 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 reward in 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:

$$v_*(s) \doteq \max_\pi v_\pi(s) \quad \forall s \in \mathcal{S} \tag{2.5}$$

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:

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a) \quad \forall s \in \mathcal{S}, a \in \mathcal{A}(s) \tag{2.6}$$

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:

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

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:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi \left[G_t \mid S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} \mid S_t = s \right] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r \mid s, a) \Big[r + \gamma \mathbb{E}_\pi \left[G_{t+1} \mid S_{t+1} = s' \right] \Big] \\ &= \sum_a \pi(a|s) \sum_{s', r} p(s', r \mid s, a) \Big[r + \gamma v_\pi(s') \Big], \quad \forall s \in \mathcal{S} \end{aligned} \tag{2.7}$$

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**:

$$
\begin{aligned}
q_*(s,a) &= \mathbb{E}_\pi \left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \;\middle|\; S_t = s, A_t = a \right] \\
&= \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma \max_{a'} q_*(s', a') \right]
\end{aligned}
\tag{2.8}
$$

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 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.

Chapter 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.

3.1 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 (the state is not taken into account). 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 previous actions. This problem is at the basis of ad-serving platforms, and we will talk more about it at the end of this chapter.

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 in multi-armed bandits we do not consider the state, 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 greater than –or equal to– all the other ones (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, we are **exploring**, as 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.

3.2 Evaluating action-value methods

Now that we have understood the need for **action-value methods**, meaning methods that estimate the values of the available actions and use them to choose actions to take, we will look at some instances of these methods.

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:

$$\begin{aligned} Q_t(a) &\doteq \frac{\text{sum of rewards when an action } a \text{ is taken prior to time } t}{\text{number of times an action } a \text{ is taken prior to time } t} \\ &= \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}} \end{aligned} \tag{3.1}$$

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:

$$A_t \doteq \operatorname*{argmax}_a Q_t(a) \tag{3.2}$$

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).

3.3 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 usage and computation time.

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:

$$\text{NewEstimate} \leftarrow \text{OldEstimate} + \text{StepSize}(\text{Target} - \text{OldEstimate}) \qquad (3.3)$$

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).

3.4 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 $\boldsymbol{\alpha}$** in the range $]0, 1]$, obtaining a formula such as the following:

$$Q_{n+1} \doteq Q_n + \alpha \left[R_n - Q_n\right] \tag{3.4}$$

This results in $\boldsymbol{Q_{n+1}}$ being a weighted average of past rewards and the initial estimate $\boldsymbol{Q_1}$, as we can see in the demonstration below:

$$\begin{aligned}
Q_{n+1} &= Q_n + \alpha\Big[R_n - Q_n\Big] \\
&= \alpha R_n + (1-\alpha)Q_n \\
&= \alpha R_n + (1-\alpha)[\alpha R_{n-1} + (1-\alpha)Q_{n-1}] \\
&= \alpha R_n + (1-\alpha)\alpha R_{n-1} + (1-\alpha)^2 Q_{n-1} \\
&= \alpha R_n + (1-\alpha)\alpha R_{n-1} + (1-\alpha)^2 \alpha R_{n-2} + ... + (1-\alpha)^{n-1}\alpha R_1 + (1-\alpha)^n Q_1 \\
&= (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} R_i
\end{aligned} \tag{3.5}$$

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**.

3.5 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, while for methods with a constant α the bias is permanent (although decreasing over time as given by the equation 3.5). 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).

3.6 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:

$$A_t \doteq \underset{a}{\operatorname{argmax}} \left[Q_t(a) + c + \sqrt{\frac{\ln t}{N_t(a)}} \right] \tag{3.6}$$

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.

3.7 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.

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

We will consider three problems:

1. The **prediction problem**, where we are given a fixed policy (that is, we know how we play) and we want to estimate v_π and q_π.
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 for the first time 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 ensure that all state-action pairs are encountered by taking –at times– random actions.

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:

$$\pi_0 \xrightarrow{E} q_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} q_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} ... \xrightarrow{I} \pi_* \xrightarrow{E} q_* \tag{4.1}$$

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 $\frac{\varepsilon}{|\mathcal{A}(s)|}$ 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: On-policy first-visit MC control (for ε-soft policies)

Parameters: small $\varepsilon > 0$

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)

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

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

Chapter 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 (3.3) we saw in section 3.3, 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:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right] \tag{5.1}$$

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 **TD(λ)** and ***n*-step TD methods**.

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'
  - $V(S) \leftarrow V(S) + \alpha \left[R + \gamma V(S') - V(S)\right]$
  - $S \leftarrow S'$
- 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 δ_t** and it represents the error in the estimate made at time t, as defined by the following:

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{5.2}$$

Note that δ_t depends on the value of the next state. Since $v_\pi(S_{t+1})$ is not known at time t, we use the current estimate $V(S_{t+1})$ instead. Updating estimates of the values of states based on estimates of the values of successor states is called **bootstrapping**.

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[1] 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:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right] \tag{5.3}$$

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

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

greediness with respect to q_π. The **SARSA (on-policy TD control) algorithm for estimating Q values** is presented in the box below:

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

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

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right] \tag{5.4}$$

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 $\varepsilon > 0$
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**.

Chapter 6

Introduction to Deep Learning and Neural Architectures

Let us now take a step back and think about the initial developments of artificial intelligence. Since their early days, AI-powered systems have been able to tackle and solve problems that had proven to be very difficult for humans, such as those that could be described by a list of formal and mathematical rules. The true challenge to artificial intelligence proved in fact to be solving tasks that are easy for people to perform, but hard for people to describe formally: problems that we solve intuitively, that feel automatic, like recognizing spoken words or faces in images. Several attempts were made to hard-code knowledge about the world in formal languages, but they were never able to achieve major successes.

Much of our knowledge is in fact based on **unstructured** inputs: computers then need to be able to capture this same knowledge and extract patterns from raw data, in order to behave in an intelligent way. **Machine learning** techniques can help in accomplishing this, enabling classification, clustering and rule discovery when given in input the right set of **features**. For many tasks, however, it is difficult to know which features should be extracted (like when recognizing objects in a photo, emotions from speech, etc.) and how information is structured and "represented". One solution to this problem is to use machine learning to discover not only the mapping from representation to output, but also the representation itself, with an approach known as **representation learning**. Learned representations often result in much better performance than the one that can be obtained by hand.

When designing features or algorithms for learning features, our goal is to

separate the **factors of variation** that explain the observed data and that help in classification. These factors, unfortunately, are often quantities that cannot be directly observed but affect the ones that are observable, like in the case of **latent factors**. We then need to disentangle the factors of variation that allow us to complete our machine learning task successfully from the ones we do not care about (e.g., if we want to classify cars and trucks, the angle of the photo is not fundamental for classification purposes).

6.1 Deep Learning

Deep learning can address this problem of representation, learning by introducing representations that are expressed in terms of other, simpler representations. This allows computers to build complex concepts out of simpler ones. The quintessential example of a deep learning model is the **feedforward deep network** (also known as **multilayer perceptron**), which consists of a mathematical function mapping some set of input values to output values. An example can be seen in picture 6.1:

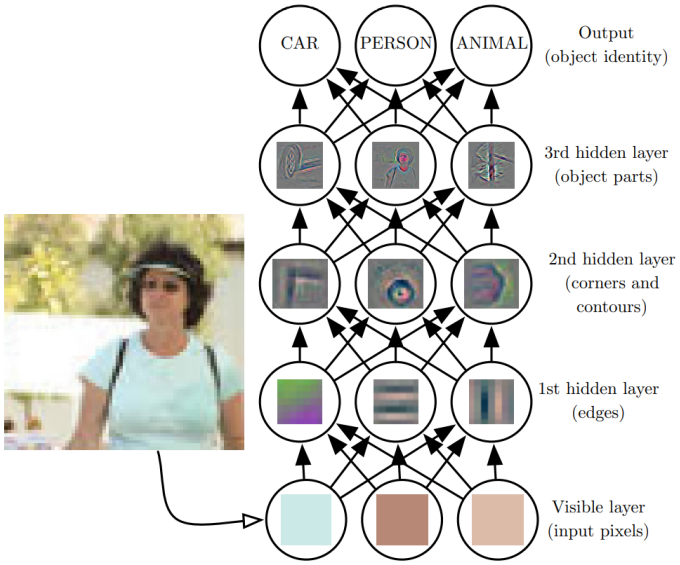


Figure 6.1: Inner workings of a feedforward deep network

Source: Goodfellow et al. 2016

The idea of learning the right representation for the data provides one perspective on deep learning. A different one is that depth allows the computer to learn a multi-step computer program. Each layer of the representation can be thought of as the state of the computer's memory after executing another

set of instructions in parallel. Networks with greater depth can execute more instructions in sequence. Sequential instructions offer great power because later instructions can refer back to the results of previous ones.

The picture below sums up the differences between various types of artificial intelligence-based systems:

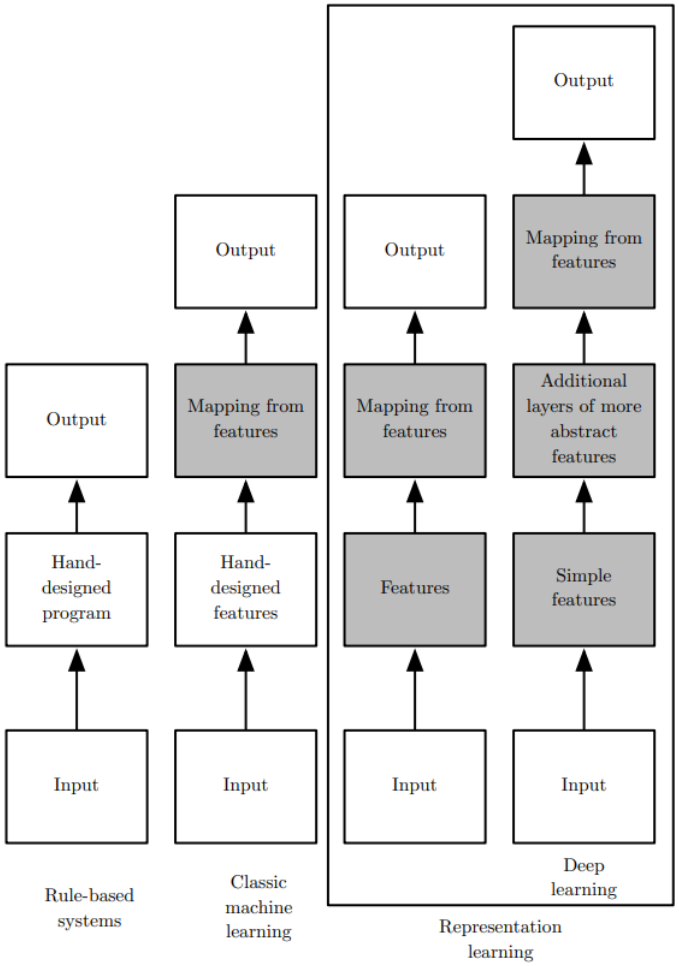


Figure 6.2: Differences between various types of AI-based systems

Source: Goodfellow et al. 2016

6.1.1 From theories of Biological Learning to Deep Learning

Deep learning only *appears* to be new due to its recent meteoric rise in popularity. As a matter of fact, instead, deep learning has been around since

the 1940s, facing alternating periods of fervor and disappointment. There have been three main waves of development of deep learning:

- Cybernetics (1940s-1960s).
- Connectionism (1980s-1990s).
- Deep learning (2006-today).

Let us now dive deeper into each of them.

Cybernetics

The origins of deep learning go by the name of "cybernetics", a term that became widespread thanks to Norbert Weiner's book "Cybernetics: control and communication in the animal and machine". As we can imagine from this title, neuroscience was of great inspiration in this period: some of the earliest learning algorithms, in fact, were intended to be computational models of biological learning, attempting to reproduce what happens (or could happen) in the brain. The brain, in fact, provided a proof by example that intelligent behavior was possible, and a conceptually straightforward path to building intelligence seemed to be to reverse engineer the computational principles behind the brain and to duplicate its functionality. As a result, one of the names that deep learning has gone by is **artificial neural networks (ANNs)**. This school of thought led to the creation of simple linear models designed to take a set of n input values $x_1, ..., x_n$ and associate them with an output y. They would then learn a set of weights $w_1, ..., w_n$ and compute the output $y = f(\boldsymbol{x}, \boldsymbol{w}) = x_1 w_1 + ... + x_n w_n$.

As a brief excursus and to understand the thought process that led to this formulation, we will now spend a few words describing what neurons are and how they work.

Neurons are electrically excitable cells that communicate with other cells via specialized connections called synapses. Communication consists of an all-or-nothing electrochemical pulse (of the same intensity each time the cell is stimulated) that takes place when the voltage of the neuron membrane changes by a large enough amount over a short interval. A typical neuron consists of a cell body (soma), dendrites and a single axon (the latter two being filaments that extrude from the soma) and can be seen in figure 6.3.

Warren McCulloch (a neuroscientist) and Walter Pitts (a logician) studied the behavior of neurons and concluded that *"Because of the "all-or-none" character of nervous activity, neural events and the relations among them can*

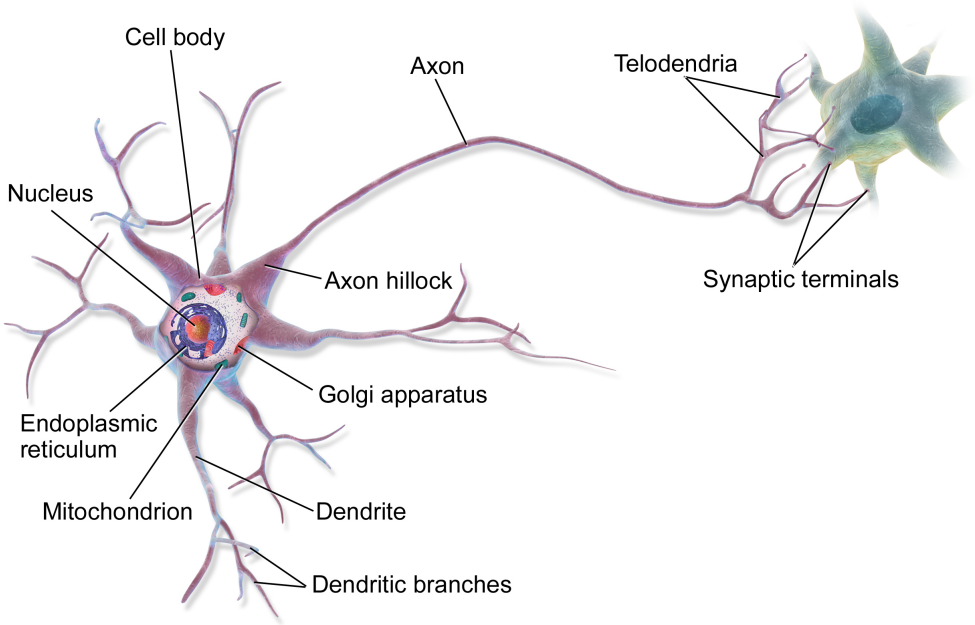


Figure 6.3: Anatomy of a multipolar neuron.
Source: BruceBlaus on Wikipedia

be treated by means of propositional logic" [MP90]. In that article they also proposed a model of a simple artificial neuron which output 1 if the weighted sum of the inputs was above a certain threshold and 0 otherwise. Inputs could only have two values, 0 and 1, and be interpreted as *excitatory* or *inhibitory* according to their weights (+1 or -1). This model enabled operations such as `NOT` (one inhibitory input, with output threshold 0), `AND` (n excitatory inputs with threshold n) and `OR` (n excitatory inputs with threshold 1) by changing the weights of the inputs. A schema of their model can be found in figure 6.4.

It is important to note that in their model the weights are **fixed** and must be set by a human operator. In 1949, though, the neuropsychologist Donald Hebb very importantly pointed out in his book "The Organization of Behavior" that *"When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased"* [Sha86]. This means that synaptic connections are reinforced when two or more neurons are activated contiguously in time and space. In simpler words, "neurons that fire together, wire together": the more we do something, the more "habitual" that learning

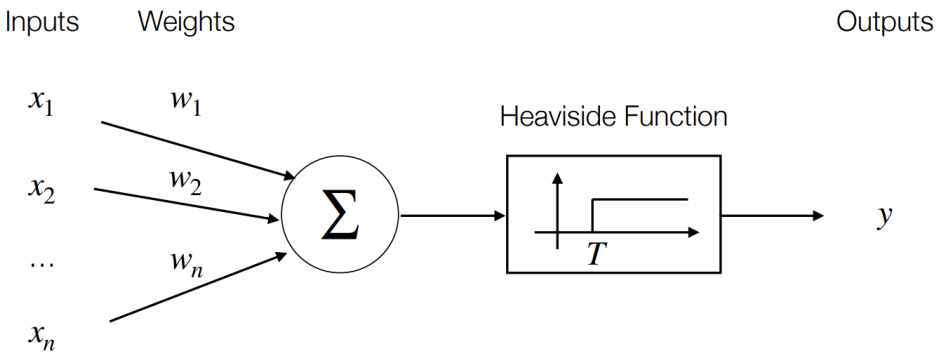


Figure 6.4: A schema of McCulloch and Pitt's artificial neuron.

Source: Prof. Mirco Musolesi

will become. Hebb's model led to the first simulations of artificial neural networks in the 1950s and can be seen in figure 6.5.

The Hebbian network model has a n-node input layer $\boldsymbol{x} = [x_1, x_2, ..., x_n]^T$ and a m-node output layer $\boldsymbol{y} = [y_1, y_2, ..., y_m]^T$, with each output being the weighted sum of the corresponding inputs:

$$y_i = \sum_{j=1}^{n} w_{j,i} x_j$$

The strengthening of the connections is represented by the following learning rule:

$$w_{j,i}^{new} \leftarrow w_{j,i}^{old} + \eta x_j y_i$$

Where η is the **learning rate**. *It must be noted that this model is different from the way current neural networks work.*

We had to wait until 1958, when Frank Rosenblatt created the **Perceptron** –a model for a machine that could learn the weights of different categories given examples of those categories– for these ideas to be put into practice.

Unfortunately, linear models such as the ones we just presented have many limitations (most famously, they cannot learn the `XOR` function) and this contributed to a period of disappointment and of little work on this field known as the "AI winter".

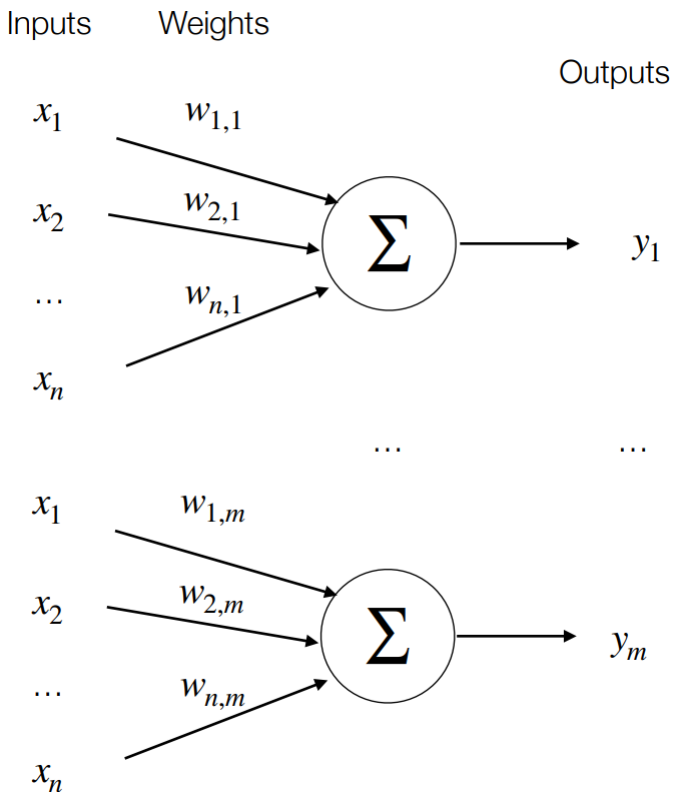


Figure 6.5: Hebb's neuron model

Source: Prof. Mirco Musolesi

Connectionism

The second wave of research on neural networks emerged in the 1980s primarily via a movement called **connectionism** or **parallel distributed processing**. Their core idea was that **many simple computational units could achieve intelligent behavior when networked together**.

Several key concepts that arose from the connectionist movement remain central to today's deep learning. Among these we find:

- **Distributed representation**: each concept is not represented by a single neuron, but by a pattern of activation over a large number of neurons. The advantage of distributed representation is that when a small random subset of the network is altered it does not change the macroscopic behavior of the network. However, the disadvantage is that it is hard to interpret or modify the connection strength of the network by an outside observer.
- The **backpropagation algorithm** for training neural networks: the error obtained from the output layer is propagated backwards to the hidden layer and is used to guide training of the weights between the hidden layer and the input layer.

Another very important idea that can be found –along with the previous ones– in a book titled "Parallel Distributed Processing" is that of "Learning and Relearning in Boltzmann Machines" [HS86]. A Boltzmann Machine is a network of symmetrically connected, neuron-like units that make stochastic decisions about whether to be on or off based on the data fed to the network while training. We consider two types of Boltzmann Machines, which we can also see in figure 6.6:

- **Restricted Boltzmann Machines**, made up of an input and a hidden layer. In this machine neurons must form a bipartite graph[1].
- **Deep Boltzmann Machines** in which there may be multiple hidden layers, with hidden nodes connected to each other.

Note that, unlike most neural networks, the connections are bidirectional and there is no output layer. The training phase for this network leverages the bidirectionality, as it allows for a continuous feedback loop between the input and the hidden layer that is bound to eventually stabilize, allowing

[1] A graph whose vertices can be divided into two disjoint and independent sets U and V such that every edge connects a vertex in U to one in V, see `https://en.wikipedia.org/wiki/Bipartite_graph`

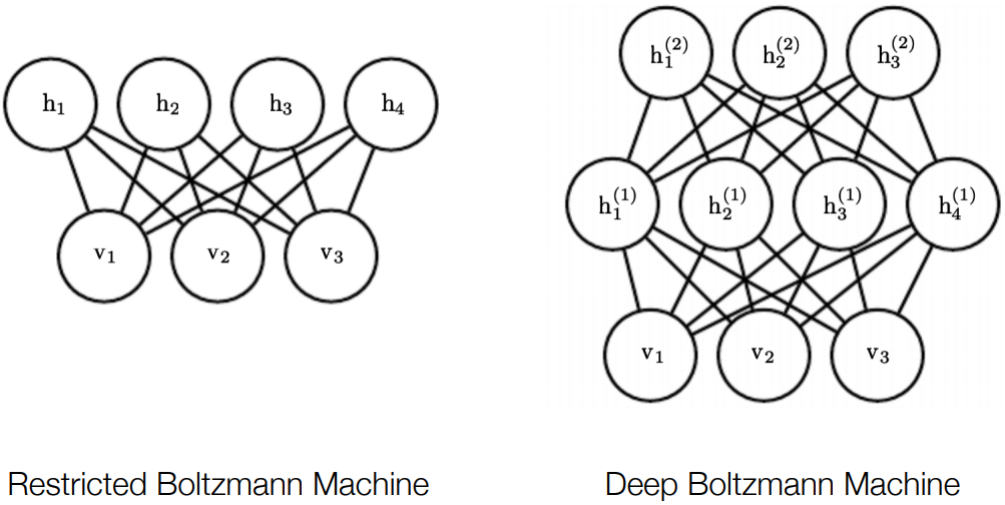


Figure 6.6: Boltzmann machines

Source: Goodfellow et al. 2016

the determination of the connection weights. The lack of an output layer is instead explained by the generative nature of these networks: if we give them a partial input (like a section of an image), the network will generate the missing parts, effectively making the results appear in the input layer.

6.1.2 Second AI winter and the current AI summer

The amount of progress being made at the time led to the creation of unrealistic expectations that, once they could not be met, brought in a second AI winter that lasted until the mid-1990s.

The current AI summer, instead, started in 2006, when Geoffrey Hinton and Simon Osindero proposed "A fast learning algorithm for deep belief nets" [HOT06] (with a strategy called *greedy layer-wise pre-training*) that enabled fast training even for networks with high dimensionality.

This breakthrough opened up a myriad of new possibilities for deep learning. Nowadays, in fact, deep learning applications are aplenty:

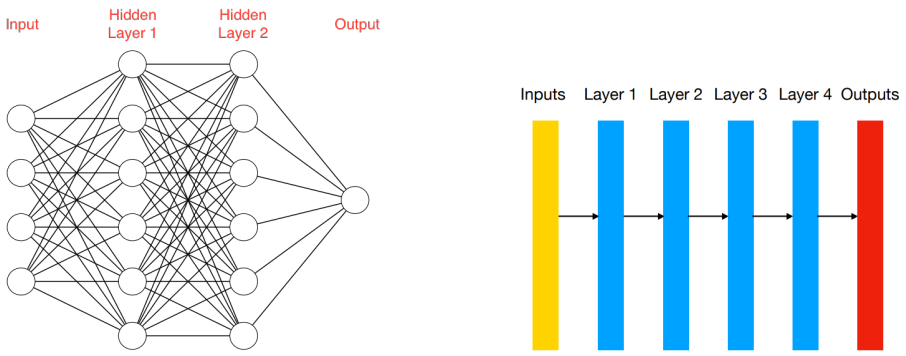
- Computer vision.
- Machine translation.
- Speech generation.
- Protein folding.

- ...

One of deep learning's biggest achievements (and the more relevant one for us) has been its extension to the field of reinforcement learning, enabling what is now known as **deep reinforcement learning**.

6.2 Deep Neural Networks

After a general overview of the history of deep learning, we now move on to more practical details about how deep neural networks work. Let us start by looking at the structure of a modern neural network and the corresponding schematized version we will use in this course (figure 6.7).



(a) (b)

Figure 6.7: (a) A modern deep neural network and (b) the simplified notation we will use in this course

Source: (a) andrewkruger on GitHub, (b) prof. Mirco Musolesi

These types of networks are known as **deep feedforward networks** or **multilayer perceptrons (MLP)** and are the quintessential deep learning models. Their goal is to approximate some function f^*: for example, for a classifier, $y = f^*(\boldsymbol{x})$ maps an input $\boldsymbol{x}$ to a category y. A feedforward network defines a mapping $y = f(\boldsymbol{x}, \boldsymbol{\theta})$ and learns the value of the parameters $\boldsymbol{\theta}$ that result in the best function approximation. This is done by feeding the network some training data and comparing our estimates to the true labels by means of a **loss function**. The **loss scores** thus obtained, will then be given to an **optimizer** which will update the weights, as we can see from figure 6.8.

Each neuron (also referred to as unit or node) of the network will then be as shown in figure 6.9.

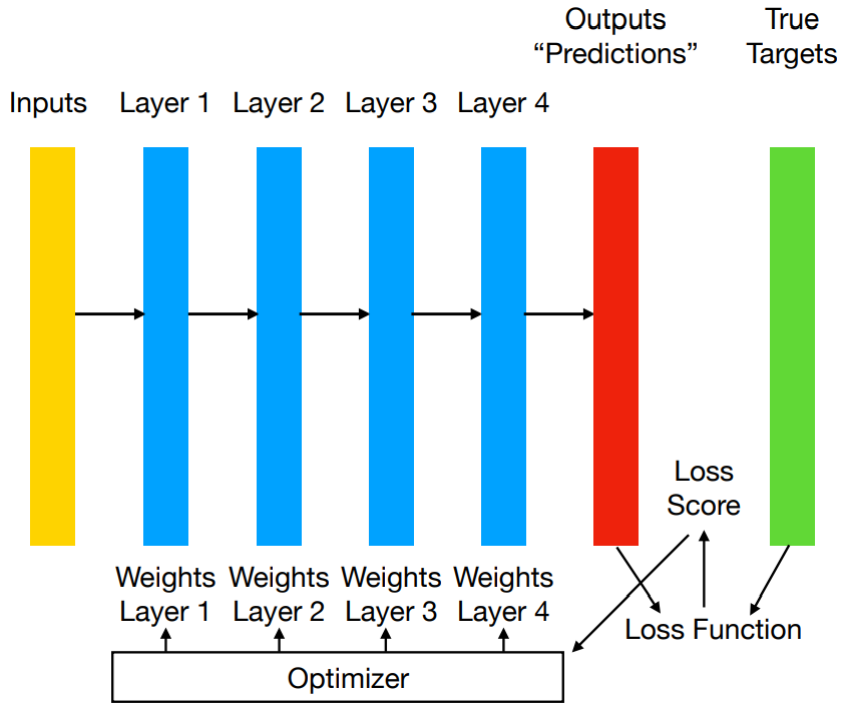


Figure 6.8: Loss scores optimization in a DNN

Source: Prof. Mirco Musolesi

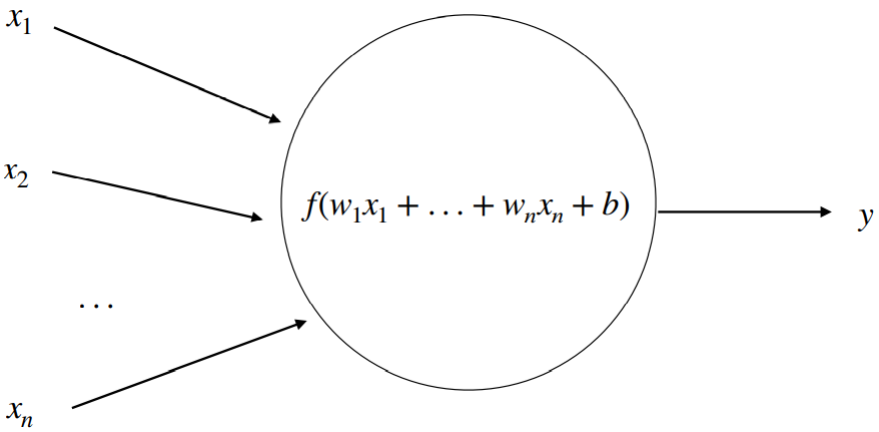


Figure 6.9: A neuron of a DNN

Source: Prof. Mirco Musolesi

Where f is the activation function, a nonlinear combination of the inputs and the weights, and b is the **bias**, a fixed, (typically) random value that works as "noise" for the network and that helps in avoiding suboptimal solutions.

There are many activation functions, with various characteristics and properties. Among the most common ones[2] we find:

- **Rectified Linear Unit (RELU)**: given the value z in input, it returns $\max(0, z)$.
- **Logistic sigmoid**: given the value z in input, it returns $\frac{1}{1+e^z}$.
- **Arctan**: given the value z in input, it returns $\tan^{-1} z$.
- **Softmax**: given a vector of real numbers in input $\boldsymbol{z}$ of dimension n, the softmax function normalizes it into a probability distribution consisting of n probabilities proportional to the exponential of each element z_i of the vector $\boldsymbol{z}$.

It is very important to note that, while the RELU function, the logistic sigmoid and the arctangent only receive in input (and output) a single real value, **the softmax function receives in input a vector of real numbers, and outputs a vector of the same dimension**. This is particularly useful for the output layer of our network in classification problems: we can use one of the 1-dimensional activation functions in the hidden layers and use the softmax function to output probabilities[3] of the input belonging to one of the n classes of our classification problem. Formally, the softmax function is defined as such:

$$\text{softmax}(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}} \text{ for } i = 1, ..., n \tag{6.1}$$

6.2.1 Gradient-based optimization

After seeing a handful of activation functions that allow our networks to work and to produce outputs, we can now focus on another very important piece of the puzzle: the optimizer.

[2]For a more comprehensive list of activation functions, see `https://en.wikipedia.org/wiki/Activation_function`.

[3]We say that these probabilities are uncalibrated: until we calibrate them, they do not represent actual probabilities, they are only monotonically ordered from less to more likely.

As we saw before, each neural layer transforms its input by applying a function as follows:

$$output = f(w_1 x_1 + ... + w_n x_n + b)$$

The learning is predicated on the gradual adjustment of the weights based on a feedback signal (typically the **loss function**, also called **cost function** or **error function**) and can be summed up in the following steps:

1. Draw a batch of training examples $\boldsymbol{x}$ and their corresponding targets $\boldsymbol{y}_{target}$.
2. Run the network on $\boldsymbol{x}$ (forward pass) to obtain the predictions $\boldsymbol{y}_{pred}$.
3. Compute the loss of the network on the batch (a measure of the mismatch between $\boldsymbol{y}_{pred}$ and $\boldsymbol{y}_{target}$).
4. Update all the weights of the network in a way that it reduces the loss of this batch. By working in terms of batches, we avoid overfitting to a single example.

Given a differentiable function, the last step can be performed by means of **gradient descent** as shown in figure 6.10, even analytically if the number of weights is particularly small.

This, however, is almost never the case, making it necessary to use approximate methods. Gradient descent methods follow the blueprint just presented, and update the weights by moving the parameters in the opposite direction of the gradient as follows:

1. Draw a batch of training examples $\boldsymbol{x}$ and their corresponding targets $\boldsymbol{y}_{target}$.
2. Run the network on $\boldsymbol{x}$ (forward pass) to obtain the predictions $\boldsymbol{y}_{pred}$.
3. Compute the loss of the network on the batch (a measure of the mismatch between $\boldsymbol{y}_{pred}$ and $\boldsymbol{y}_{target}$).
4. Compute the gradient of the loss function with regards to the network's parameters (backward pass).
5. Move the parameters in the opposite direction of the gradient with:
   - $w_j \leftarrow w_j + \Delta w_j = w_j - \eta \frac{\partial J}{\partial w_j}$ for a single piece of data at a time.
   - $w_j \leftarrow w_j + \Delta w_j = w_j - \eta$ average $\left(\frac{\partial J}{\partial w_j}\right)$ for batches of k samples.

Where J is the loss function.

If the input is drawn randomly from the available data, we refer to this as **Stochastic Gradient Descent**. If we use batches of data selected randomly, we call it **(mini)-batch Stochastic Gradient Descent**. It is also important to note that this is a simplified model, where we assume only one layer (and thus only one loss function): if this was not the case, we would have to calculate the gradients of all the loss functions and apply the weight correction to all the layers. A more realistic visualization of the "landscape" that we have to traverse during gradient descent is shown in figure 6.11.

6.2.2 The backpropagation algorithm

Now that we have seen how to perform weight optimization, we briefly introduce the **backpropagation algorithm**, a method through which we can apply this correction.

Let us imagine we have a network with L layers, each with a set of weights represented by the vector $\boldsymbol{W}^l$, with l being the layer we are considering. The network output will be then given by:

$$\boldsymbol{y}_{pred} = f^L\left(\boldsymbol{W}^L\Big(f^{L-1}\big(\boldsymbol{W}^{L-1}...f^1\left(\boldsymbol{W}^1\boldsymbol{x}\right)...\big)\Big)\right) \tag{6.2}$$

To perform the update, we can apply the chain rule (shown here) to calculate the derivative of the composite function represented by our network as such:

$$\frac{d}{dx}\Big[f\big(g(x)\big)\Big] = f'\big(g(x)\big)g'(x)$$

It is called backpropagation algorithm because it starts by calculating the gradient of the final loss value and works backwards from the right-most layers to the left-most ones, applying the chain rule to compute the contribution that each weight had in the loss value.

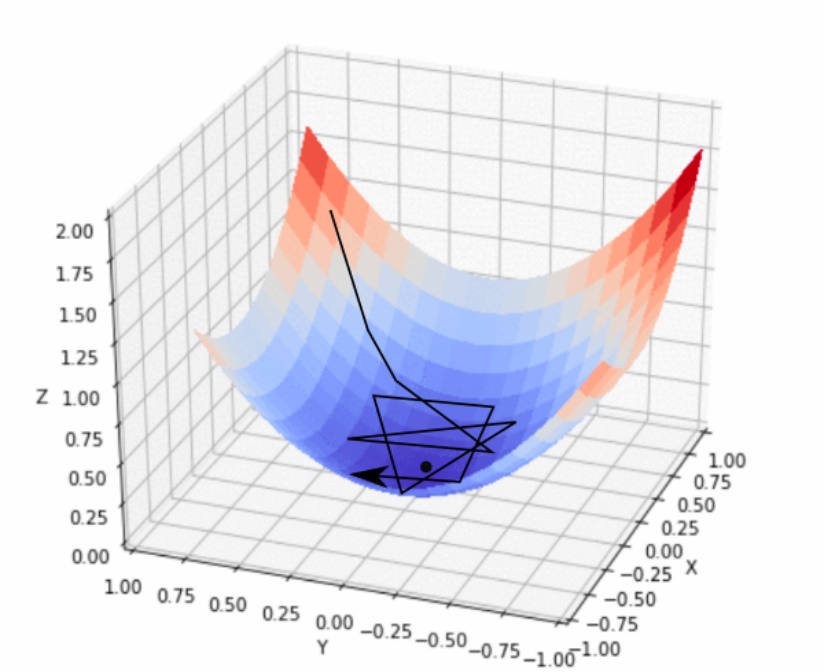


Figure 6.10: Gradient descent in a 3-dimensional space.
Source: Ayoosh Kathuria on PaperSpace

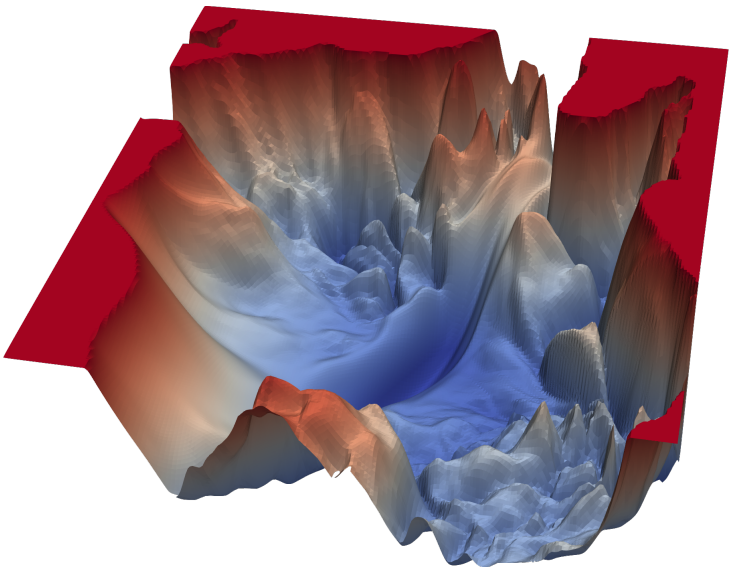


Figure 6.11: The loss landscape of a neural network.
Source: University of Maryland

Chapter 7

Value Function Approximation in Reinforcement Learning

In this chapter, we will focus on the study of function approximation for estimating state-value functions from on-policy data. That is, approximating v_π from experience generated using a known policy π. The novelty in this chapter is that the approximate value function is not represented as a table, but as a parametrized functional form with a weight vector $\boldsymbol{w} \in \mathbb{R}^d$.

We will write $\hat{v}(s, \boldsymbol{w}) \approx v_\pi(s)$ for the approximate value of state s given a weight vector $\boldsymbol{w}$, with $\hat{v}$ being either a linear function of the weights or, more generally, a (nonlinear) function computed by a multi-layer artificial neural network. Typically, the number of weights is much less than the number of states ($d \ll |\mathcal{S}|$) and changing one weight changes the estimated value of many states. Consequently, when a single state is updated, the change **generalizes** from that state to affect the values of many other states. Such generalization makes the learning potentially more powerful, but also potentially more difficult to manage and understand.

In addition to that, if the parametrized function form for $\hat{v}$ does not allow the estimated value to depend on certain aspects of the state, it will behave as if those aspects are unobservable to the agent, making it **possible to apply these techniques to partially observable problems** as well.

7.1 Value function approximation

Until now, we have seen methods that performed updates to an estimated value function by shifting its value at particular states towards an "up-

date target" for that state. Let us refer to an individual update by the notation $s \mapsto u$, where s is the state updated and u is the update target that s's estimated value is shifted towards. For example, the Monte Carlo update for value prediction is $S_t \mapsto G_t$; the TD(0) update is instead $S_t \mapsto R_{t+1}\gamma\hat{v}\left(S_{t+1}, \boldsymbol{w}_t\right)$.

It is natural to interpret each update as specifying an example of the desired input-output behavior of the value function. In a sense, the update $s \mapsto u$ means that the estimated value for state s should be more like the update target u. If we were to perform this update in tabular methods, we would shift s's estimated value a fraction of the way towards u, while leaving the values of the other states unchanged. Now we permit arbitrarily complex and sophisticated methods to implement the update, and updating at s generalizes so that the estimated values of many other states are changed as well. Machine learning methods that learn to mimic input-output examples in this way are called **supervised learning methods**, and when the outputs are numbers, like u, the process is called **function approximation**.

Artificial neural networks (ANN) are widely used for approximating nonlinear functions, especially because of the guarantees that the **universal approximation theorem** (initially proven by George Cybenko in 1989) provides. Cybenko, in fact, showed that an ANN with a single hidden layer containing a large enough finite number of sigmoid units can approximate, on a compact region of the network's input space, any continuous function to any degree of accuracy [Cyb89].

7.2 Stochastic Gradient Descent for function approximation

We now develop in detail stochastic gradient descent (SGD) methods for function approximation. SGD methods are among the most widely used function approximation methods and are particularly well suited for online reinforcement learning.

To limit the complexity of the matter at hand, we will consider only the case in which we update our network using *only one example at a time*. Let us recall the formula we introduced in the previous chapter for updating the weights using SGD (for batches of examples we can use the mini-batch SGD):

$$w_j \leftarrow w_j + \Delta w_j = w_j - \eta \frac{\partial J}{\partial w_j} \tag{7.1}$$

Since we will be updating $\boldsymbol{w}$ at each of a series of discrete time steps, $t = 0, 1, 2, ...$, we modify our notation to include the time information as such:

$$w_{j,t+1} \leftarrow w_{j,t} + \Delta w_{j,t} = w_{j,t} - \eta \frac{\partial J}{\partial w_{j,t}} \tag{7.2}$$

Assuming that our loss function is represented by the **mean square error** shown here:

$$J(\boldsymbol{w}) = \sum_{s \in \mathcal{S}} \big(U_t - \hat{v}(S_t, \boldsymbol{w}_t)\big)^2 \tag{7.3}$$

–where U_t is our **target** and $\hat{v}(S_t, \boldsymbol{w}_t)$ is our current estimate– we will then obtain the following weight update formula:

$$\begin{aligned} \boldsymbol{w}_{t+1} &\doteq \boldsymbol{w}_t - \frac{1}{2}\eta \nabla \big[U_t - \hat{v}(S_t, \boldsymbol{w}_t)\big]^2 \\ &= \boldsymbol{w}_t + \eta \big[U_t - \hat{v}(S_t, \boldsymbol{w})\big] \nabla \hat{v}(S_t, \boldsymbol{w}_t) \end{aligned} \tag{7.4}$$

Since the step-size parameter η is chosen arbitrarily, we multiplied it by $\frac{1}{2}$ to cancel out the 2 that comes from the derivative.

7.2.1 Semi-gradient one-step SARSA prediction

The weight update rule we just introduced can be used to extend to state-action value functions the mathematical framework we discussed for state value functions . In this case, the update target U_t will be an approximation of $q_\pi(S_t, A_t)$, making the gradient-descent update as such:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \eta \big[U_t - \hat{q}(S_t, A_t, \boldsymbol{w}_t)\big] \nabla \hat{q}(S_t, A_t, \boldsymbol{w}_t) \tag{7.5}$$

If we combine this formula with the one we introduced in equation 5.3, we obtain the **episodic semi-gradient one-step SARSA** update rule:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \eta \big[R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \boldsymbol{w}_t) - \hat{q}(S_t, A_t, \boldsymbol{w}_t)\big] \nabla \hat{q}(S_t, A_t, \boldsymbol{w}_t) \tag{7.6}$$

It is called semi-gradient because we are not really taking the "real" gradient, as it would require us to calculate the gradient of U_t as well (simply speaking,

we assume the output is not function of $\boldsymbol{w}_t$ even though it is). This is essential to avoid introducing too much complexity and, if the policy is constant, we still have guarantees of convergence.

7.2.2 Semi-gradient one-step SARSA control

As usual, however, we are more interested in the problem of *control*, rather than the one of *prediction*. To tackle it in the case of function approximation, we employ methods that are essentially the same as the ones we used in the tabular case: for each possible action a available in the current state S_t, we can compute $\hat{q}(S_t, a, \boldsymbol{w}_t)$ and then find the greedy action using:

$$A_t^* = \operatorname*{argmax}_a \hat{q}(S_t, a, \boldsymbol{w}_t)$$

Policy improvement is then done by changing the estimation policy to a soft approximation of the greedy policy (e.g., ε-greedy policy). Since we are in the *on-policy* case, actions are selected according to this same policy. We introduce the pseudo-code for the **episodic semi-gradient on-policy SARSA algorithm** in the box below:

Algorithm 7: Episodic Semi-gradient SARSA for estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parametrization
$\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Parameters: step size $\eta > 0$, small $\varepsilon > 0$
Initialize: value-function weights $\boldsymbol{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\boldsymbol{w} = \boldsymbol{0}$)
Loop *for each episode* **:**
  $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
  Loop *for each step of the episode* **:**
    Take action A, observe R, S'
    if S' *is terminal* **then**
      $\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta\big[R - \hat{q}(S, A, \boldsymbol{w})\big]\nabla\hat{q}(S, A, \boldsymbol{w})$
      Go to next episode
    Choose A' as a function of $\hat{q}(S', \cdot, \boldsymbol{w})$ (e.g., ε-greedy)
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta\big[R + \gamma\hat{q}(S', A', \boldsymbol{w}) - \hat{q}(S, A, \boldsymbol{w})\big]\nabla\hat{q}(S, A, \boldsymbol{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$

7.2.3 Semi-gradient Q-learning

The same thought process can be applied to obtain **semi-gradient Q-learning**, in which the update rule will be:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \eta\big[R_{t+1} + \gamma \operatorname*{argmax}_a \hat{q}(S_{t+1}, a, \boldsymbol{w}_t) - \hat{q}(S_t, A_t, \boldsymbol{w}_t)\big]\nabla\hat{q}(S_t, A_t, \boldsymbol{w}_t) \tag{7.7}$$

Where $\boldsymbol{w}_t$ is the vector of the network's weights, A_t is the action selected at time t, and S_t and S_{t+1} are the inputs to the network at their respective time steps.

Semi-gradient Q-learning (or, interchangeably, **Deep Q-Network**, **DQN**) has been used to solve Atari games. It combines Q-Learning with a deep (convolutional) neural network, allowing, in this case, the conversion of the original frames from the 60Hz video feed (210×160 pixels, 128 colors) in an 84×84 matrix of luminance values. Since movement is very important in videogames (e.g., what is the direction of the ball in a tennis match, what is its speed, etc.) and it cannot be necessarily inferred from a single frame, researchers decided to feed the network a stack of four frames at a time (making the input $84 \times 84 \times 4$). In addition to that, the rewards of the DQN have been restricted to be one of $\{-1, 0, +1\}$. This has proven to yield better results (possibly) due to a reduction of instability (we learn more quickly if actions that are similar give us the same reward). More information on this can be found in [Mni+13] and [Mni+15]. Let us now look at some key features of DQN.

We are in the **control problem**: we want to learn the optimal way to play a game. This, of course, means that the more we learn, the greedier will our policy have to be: once we are confident in our policy, we will have to start playing at the best of our abilities. The value of both ε and the decay will be part of our **hyperparameters** and we will have to find their best value experimentally. In the examples cited before, ε was set to decrease linearly over the first million frames and the network was trained using a mini-batch stochastic gradient with batches of 32 images (each made up of 4 frames), with changes proportional to the running average of the squares of gradients (RMSprop).

Another important thing that may be somewhat counterintuitive, is that the network is made to play all Atari games, and not necessarily any specific one: the output layer (as shown in figure 7.1) will then always be made up of 18 nodes, even though only a subset may ever have values that are not 0.

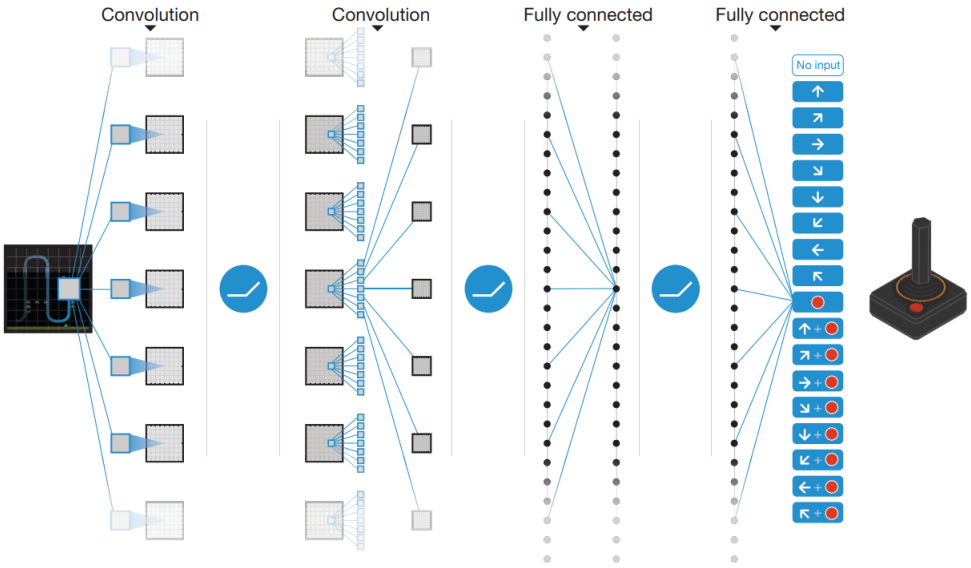


Figure 7.1: DQN Schema

Source: Mnih et al. 2013

The authors of DQN made a series of changes to the "standard" Q-learning, the most important of which has probably been the introduction of **experience replay** to deal with the convergence problem of Q-learning. Picking actions that are highly correlated, in fact, has the effect of limiting our exploration of the space, thus leading to a much slower process of convergence (e.g., we may learn to run well on tarmac, but as soon as we go on sand, it will take us quite a while to adapt, causing updates with **high variance** to compensate).

The key idea is then to perform training through an **emulator** and storing the agent's experience in a **replay memory** that is accessed to perform the weight updates. In practice, after the emulator executes action A_t in a state represented by the image stack S_t, receiving reward R_{t+1}, we add to the replay memory (or **replay buffer**) a tuple made up of these values and the image stack S_{t+1}: $\left(S_t, A_t, R_{t+1}, S_{t+1}\right)$. Each Q-learning update will then be performed by randomly sampling from the experience replay buffer. Instead of S_{t+1} becoming the new S_t for the next update, as it happens in normal Q-learning, a new, unconnected experience is drawn from the replay memory, helping us avoid getting stuck with actions that do not generalize. It is important to note that this is possible because Q-learning is an *off-policy* algorithm, and so it does not need to learn from "connected" trajectories. More information can be found in [Lin92].

Chapter 8

Policy Gradient Methods

Until now we have discussed so-called **action-value methods**, in which we learn the action values and use a policy (e.g., ε-greedy) to select actions based on those estimates. A possible alternative would be to have a different set of methods that could learn a **parametrized policy that allowed them to select actions without consulting a value function** (a value function might be used to learn the policy parameter, but it is not required for action selection; we will keep using the $\boldsymbol{w} \in \mathbb{R}^d$ notation for the weight vector of the value function).

For **policy networks** we will use $\boldsymbol{\theta}$ to indicate the **policy's parameter vector**, with $\boldsymbol{\theta} \in \mathbb{R}^{d'}$. The probability of taking an action a at time t, given that the environment is in state s at time t with parameter $\boldsymbol{\theta}$, will then be:

$$\pi\left(a|s, \boldsymbol{\theta}\right) = \Pr\left\{A_t = a \mid S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}\right\}$$

The methods that will be introduced in this chapter will learn the policy parameter $\boldsymbol{\theta}$ by means of (the gradient of) a scalar performance measure $J(\boldsymbol{\theta})$. Since these methods will seek to *maximize* performance[1], their updates will work towards approximating **gradient ascent** in J:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\widehat{\nabla J(\boldsymbol{\theta}_t)} \tag{8.1}$$

Where $\widehat{\nabla J(\boldsymbol{\theta}_t)} \in \mathbb{R}^{d'}$ is a stochastic estimate whose expectation approximates the gradient of the performance measure with respect to its argument $\boldsymbol{\theta}_t$. In

[1]Earlier we were trying to **minimize the error** in the Q-value estimation, whereas now we are trying to **maximize the performance** of our policy.

simpler terms, we perform our weight correction in the direction that, based on our estimate of $J(\boldsymbol{\theta})$, will lead to a gradient ascent. Methods that follow this schema are called **policy gradient methods**.

8.1 Policy approximation and its advantages

In policy gradient methods, the policy can be parametrized in any way, with one caveat: $\pi(a|s, \boldsymbol{\theta})$ must be differentiable with respect to its parameters, that is, as long as $\nabla\pi(a|s, \boldsymbol{\theta})$ exists and is finite $\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$ and $\boldsymbol{\theta} \in \mathbb{R}^{d'}$. In practice, however, to ensure exploration it is usually required that the policy never becomes deterministic.

A common way to parametrize policies is to form a **parametrized numerical preference** $h(s, a, \boldsymbol{\theta}) \in \mathbb{R}$ for each state-action pair (this is often the case when the action space is discrete and not too large). The actions with the highest preferences in each state will then be given the highest probabilities of being selected, typically according to an exponential soft-max distribution as such:

$$\pi(a|s, \boldsymbol{\theta}) \doteq \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_b h(s, b, \boldsymbol{\theta})} \tag{8.2}$$

Where $e \approx 2.71828$ is Euler's number, the base of the natural logarithm. This type of policy parametrization is called **soft-max in action preferences**.

The parametrization of the action preferences is arbitrary, too. The $h(s, a, \boldsymbol{\theta}) \in \mathbb{R}$ values might be computed, for example, by a deep artificial neural network, where $\boldsymbol{\theta}$ is the vector of all the connection weights of the network (like in the case of AlphaGo). This is just one of the options, but it helps us scale up.

One advantage of parametrizing policies according to the *soft-max in action preferences* is that the approximate policy can approach a deterministic one, whereas with ε-greedy we would always have an ε probability of selecting a random action. We could also select actions according to a soft-max distribution based on action values, but this alone would not allow the policy to approach a deterministic one. The action-value estimates would end up converging to their true values, which would differ by a finite amount, causing the soft-max function to output specific probabilities, and not just 0 and 1. Action preferences are different because they *do not approach specific values*; instead, *they are driven to produce the optimal stochastic policy*: if the opti-

mal policy is deterministic, then the preferences of the optimal actions will be driven infinitely higher than all suboptimal actions (if permitted by the parametrization).

A second advantage is that this method enables the selection of actions with arbitrary probabilities. This means that, when dealing with problems where the best approximate policy is stochastic (such as in rock-paper-scissors), we can even learn to choose one of the available actions with equal probability, something that is not possible with action-value methods.

Last, and perhaps the simplest, advantage that policy parametrization may have over action-value parametrization is that the policy may be a simpler function to approximate, converging faster and more smoothly.

8.2 Policy Gradient Theorem

As we just said, with continuous policy parametrization the action probabilities change smoothly as a function of the learned parameter, whereas in ε-greedy selection the action probabilities may change dramatically for an arbitrary small change in the estimated action values, if that change results in a different action having the maximal value. Largely because of this, policy gradient methods have stronger convergence guarantees compared to action-value methods.

Let us consider **episodic learning**, for which we define the performance measure as the value of the start state of the episode[2]. We simplify the notation (without losing any meaningful generality) by assuming that every episode starts in some particular (non-random) state s_0. The performance is then defined as:

$$J(\boldsymbol{\theta}) \doteq v_{\pi_{\boldsymbol{\theta}}}(s_0)$$

Where $v_{\pi_{\boldsymbol{\theta}}}$ is the true value function for $\pi_{\boldsymbol{\theta}}$, the policy determined by $\boldsymbol{\theta}$. The issue now becomes how to change the policy parameter $\boldsymbol{\theta}$ by means of function approximation in a way that ensures improvement.

Doing so is more complicated than it may seem from a superficial look. Performance –as in, how good our policy is– depends on two things:

[2]Our objective is to play to the best of our abilities, which means maximizing the expected cumulative rewards G. We obtain exactly that by maximizing the value of the initial state.

- The actions we choose in a certain state.
- The states that the policy makes us go through.

Both factors are affected by the policy parameter. However, while it is easy to compute, given a state and knowledge of the parametrization, the effect of the policy parameter on the actions, the effect of the policy on the state distribution (the states that we end up visiting) is a function of the environment, which is typically unknown, making it impossible to estimate.

How can we then perform gradient ascent (formula 8.1) if the gradient of the performance with respect to the policy parameter depends on the unknown effect of policy changes on the state distribution? The **policy gradient theorem** comes to our rescue, providing us with an analytic expression of the gradient that does not involve the derivative of the state distribution, as we can see below:

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta}) \tag{8.3}$$

Where the gradients are column vectors of partial derivatives with respect to the components of $\boldsymbol{\theta}$ and π denotes the policy corresponding to parameter vector $\boldsymbol{\theta}$. $\mu(s)$ is the **on-policy distribution** under policy π and it describes the normalized frequency with which states are encountered while the agent is selecting actions according to π or, to put it differently, it is the normalized fraction of time spent in each state. A proof for this formula can be found in Sutton and Barto's book on page 325.

We might then ask: the policy gradient theorem only provides us with something that is *proportionate to* the gradient, and not *equal to*, does it still work? The answer is yes, in the weight update formula, in fact, we can use the step size α to turn this proportionality into an equality by setting it to the average length of an episode.

8.3 REINFORCE: Monte Carlo Policy Gradient

As we mentioned in the beginning of this chapter, our strategy of stochastic gradient ascent is based on $\widehat{\nabla J(\boldsymbol{\theta}_t)}$, a stochastic estimate whose expectation approximates the gradient of the performance measure with respect to its argument $\boldsymbol{\theta}_t$. Thanks to the policy gradient theorem, this approximation is

sufficient, and we can "absorb" the error of this estimate in the step size constant α.

The problem at hand now becomes finding a way of sampling whose expectation equals or approximates the expression given by the policy gradient theorem[3]. Given the definition of μ we gave earlier, we can see the right-hand side of the theorem as a sum over states weighted by how often the state occurs under the target policy π. If π is followed, then the states will be encountered in these proportions, which means that:

$$\begin{aligned} \nabla J(\boldsymbol{\theta}) &\propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta}) \\ &= \mathbb{E}_\pi \left[\sum_a q_\pi(S_t,a) \nabla \pi(a|S_t,\boldsymbol{\theta}) \right] \end{aligned} \tag{8.4}$$

As a reminder, we defined $\mathbb{E}_\pi[\cdot]$ as the expected value of a random variable given that the agent follows π.

In theory, we could stop here and instantiate the stochastic gradient-ascent algorithm (8.1) as:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \sum_a \hat{q}(S_t,a,\boldsymbol{w}) \nabla \pi(a|S_t,\boldsymbol{\theta})$$

Where $\hat{q}$ is some learned approximation of q_π. We obtained an **all-action method**, because its update involves all the actions in each update, but our focus is on REINFORCE, whose update at time t involves just A_t, the action actually taken at time t.

We continue our derivation of REINFORCE by introducing A_t in the same way we introduced S_t just now: by replacing a sum over the random variable's possible values by an expectation under π and then sampling the expectation. Equation 8.4 now becomes:

[3]Since the update formula is based on $\widehat{\nabla J(\boldsymbol{\theta}_t)}$, a stochastic estimate of the gradient of the performance function with respect to its parameters, we need to find a way to sample $\nabla J(\boldsymbol{\theta})$. We do so by applying the policy gradient theorem.

$$
\begin{aligned}
\nabla J(\boldsymbol{\theta}) &\propto \mathbb{E}_\pi\left[\sum_a q_\pi(S_t,a)\nabla\pi(a|S_t,\boldsymbol{\theta})\right] \\
&= \mathbb{E}_\pi\left[\sum_a \pi(a|S_t,\boldsymbol{\theta})\cdot q_\pi(S_t,a)\nabla\pi(a|S_t,\boldsymbol{\theta})\cdot\frac{1}{\pi(a|S_t,\boldsymbol{\theta})}\right] \\
&= \mathbb{E}_\pi\left[\mathbb{E}_\pi\left[q_\pi(S_t,A_t)\right]\frac{\nabla\pi(a|S_t,\boldsymbol{\theta})}{\pi(a|S_t,\boldsymbol{\theta})}\right] \quad \text{since } \mathbb{E}_\pi\left[\mathbb{E}_\pi[\cdot]\right]=\mathbb{E}_\pi[\cdot] \\
&= \mathbb{E}_\pi\left[q_\pi(S_t,A_t)\frac{\nabla\pi(a|S_t,\boldsymbol{\theta})}{\pi(a|S_t,\boldsymbol{\theta})}\right] \quad \text{since } \mathbb{E}_\pi\left[G_t|S_t,A_t\right]=q_\pi(S_t,A_t) \\
&= \mathbb{E}_\pi\left[G_t\frac{\nabla\pi(a|S_t,\boldsymbol{\theta})}{\pi(a|S_t,\boldsymbol{\theta})}\right]
\end{aligned}
\tag{8.5}
$$

We now have an expression that can be sampled on each time step and whose expectation is proportional to the gradient. Using this sample to instantiate our generic stochastic gradient ascent algorithm, we obtain the REINFORCE update rule:

$$
\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha G_t\frac{\nabla\pi(a|S_t,\boldsymbol{\theta})}{\pi(a|S_t,\boldsymbol{\theta})} \tag{8.6}
$$

With this rule, each increment is proportional to the product of a return G_t and a vector, the gradient of the probability of taking the action actually taken divided by the probability of taking that action. The vector is the direction in the parameter space that most increases the probability of repeating the action A_t on future visits to state S_t. The update increases the parameter vector in this direction proportional to the return, and inversely proportional to the action probability. The former makes sense because it causes the parameter to move most in the directions that favor actions that yield the highest return. The latter makes sense because otherwise actions that are selected frequently are at an advantage (the updates will be more often in their direction) and might win out even if they do not yield the highest return.

If we then apply the identity $\nabla \ln x = \frac{\nabla x}{x}$, we obtain the final formulas:

$$
\nabla J(\boldsymbol{\theta}) \propto \mathbb{E}_\pi\Big[G_t\nabla\ln\pi(a|S_t,\boldsymbol{\theta})\Big] \tag{8.7}
$$

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha G_t \nabla \ln \pi(a|S_t, \boldsymbol{\theta}) \tag{8.8}$$

A pseudocode implementation of REINFORCE is provided here:

Algorithm 8: REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parametrization $\pi(a|S_t, \boldsymbol{\theta})$
Parameters: step size $\alpha > 0$
Initialize: policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)
Loop *for each episode* **:**
  Generate an episode $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  Loop *for each step of the episode* $t = 0, 1, ..., T-1$ **:**
    $G \leftarrow \sum_{k=t+1}^{T} R_k$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

To have a better idea of how this is could be put into practice in a neural network, we can look at figure 8.1.

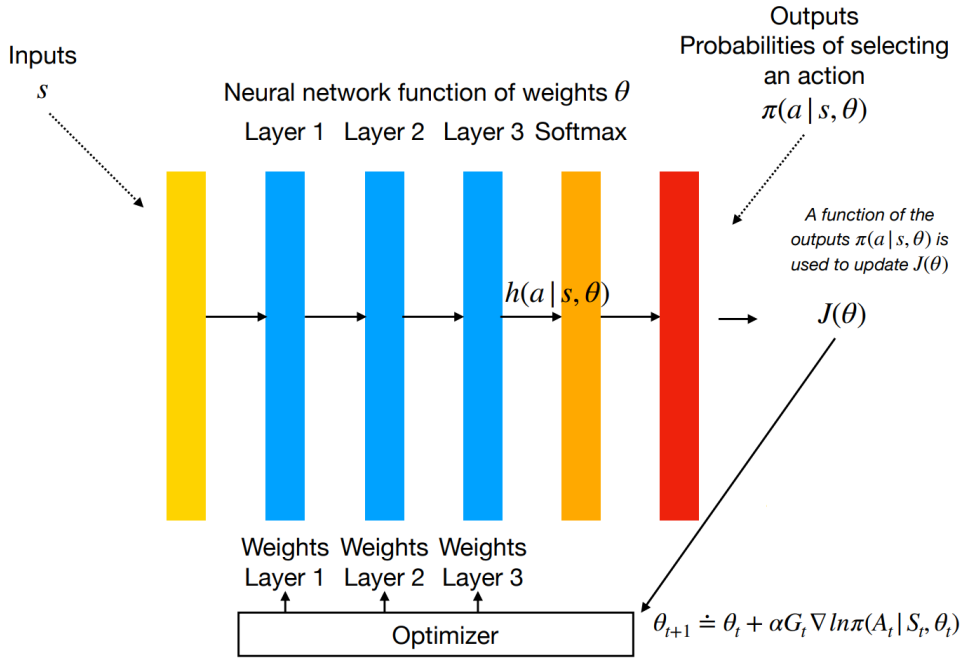


Figure 8.1: A neural network using REINFORCE

Source: Prof. Mirco Musolesi

8.3.1 Stochastic Gradient Ascent

What we have seen in picture 8.1 works, except for one not-so-minor detail: neural networks (and the backpropagation algorithm) work by minimizing the loss function, performing stochastic gradient *descent*, which is the opposite of what we want to do (an explanation of how stochastic gradient descent works can be found in subsection 6.2.1).

A simple –but effective– trick to have neural networks perform stochastic gradient *ascent* is then to have a new function $J'(\boldsymbol{\theta})$ that is the opposite of $J(\boldsymbol{\theta})$. Formally, we will have:

- $J'(\boldsymbol{\theta}) = -(G_t \ln \pi - 0) = -G_t \ln \pi$
- $\nabla J'(\boldsymbol{\theta}) = -G_t \nabla \ln \pi$

8.3.2 How well does REINFORCE perform?

As a stochastic gradient method, REINFORCE has good theoretical convergence properties: by construction, the expected update over an episode is in the same direction as the performance gradient. This assures an improvement in expected performance for sufficiently small α, and convergence to a local optimum under standard stochastic approximation conditions for decreasing α. However, being a Monte Carlo method, REINFORCE suffers from high variance[4], causing a slow learning process. To deal with this problem, we can use **baselines** or **actor-critic** methods.

8.4 REINFORCE with Baseline

To overcome the issue of high variance in Monte Carlo updates, we can generalize the *policy gradient theorem* (equation 8.3) by including a comparison of the action value to an arbitrary **baseline** $b(s)$:

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a \big(q_\pi(s,a) - b(s)\big) \nabla \pi(a|s, \boldsymbol{\theta}) \tag{8.9}$$

[4]The variance is high because the return G_t is the sum of many random variables $R_t, R_{t+1}, ..., R_T$, each of which depends on distributions not just in the case of the reward R received by the environment, but also in the choice of action A by the ε-soft policy, and the state transition dynamics that pick each S. Ref: `https://stats.stackexchange.com/questions/364997/what-is-intuition-behind-high-variance-of-monte-carlo-method`

The baseline will act as a *smoothing factor* by limiting the magnitude of the updates. We can choose any function, even a random variable, to use as a baseline, as long as it does not vary with a. This detail allows the equation to remain valid and unbiased, as the quantity that we subtracted is zero, as proven here:

$$
\begin{aligned}
&\sum_a b(s)\nabla\pi(a|s,\boldsymbol{\theta}) \\
&= b(s)\sum_a \nabla\pi(a|s,\boldsymbol{\theta}) \quad \text{since } b(s) \text{ does not depend on } a \\
&= b(s)\nabla\sum_a \pi(a|s,\boldsymbol{\theta}) \quad \text{since the sum of gradients is the gradient of the sum} \\
&= b(s)\nabla 1 \quad \text{since the sum of probabilities is 1} \\
&= 0
\end{aligned}
\tag{8.10}
$$

The **policy gradient theorem with baseline** can then be used to derive the REINFORCE with baseline update rule using similar steps as the ones we used in 8.5:

$$
\begin{aligned}
\boldsymbol{\theta}_{t+1} &\doteq \boldsymbol{\theta}_t + \alpha\big(G_t - b(S_t)\big)\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})} \\
&= \boldsymbol{\theta}_t + \alpha\big(G_t - b(S_t)\big)\nabla\ln\pi(A_t|S_t,\boldsymbol{\theta})
\end{aligned}
\tag{8.11}
$$

In order to achieve the best convergence speed, it is important to choose the baseline appropriately for each state: in some states all actions have high values, and we need a high baseline to differentiate the higher valued actions from the less highly valued ones; in other states all actions will have low values and a low baseline is appropriate. A natural choice for the baseline will then be an estimate of the state value, $\hat{v}(S_t,\boldsymbol{w})$, where $\boldsymbol{w}\in\mathbb{R}^d$ is a weight vector learned using –for example– another deep neural network.

A pseudocode algorithm for **REINFORCE with baseline** is given in the box below:

Algorithm 9: REINFORCE with Baseline (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parametrization $\pi(a|s, \boldsymbol{\theta})$
a differentiable state-value function parametrization $\hat{v}(s, \boldsymbol{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\boldsymbol{w}} > 0$
Initialize: policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\boldsymbol{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)
Loop *for each episode* **:**
  Generate an episode $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  Loop *for each step of the episode* $t = 0, 1, ..., T-1$ **:**
    $G \leftarrow \sum_{k=t+1}^{T} R_k$
    $\delta \leftarrow G - \hat{v}(S_t, \boldsymbol{w})$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha^{\boldsymbol{w}} \delta \nabla \hat{v}(S_t, \boldsymbol{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \delta \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

8.5 Actor-Critic methods

After seeing *REINFORCE with baseline*, we introduce **actor-critic methods**, named after their two components:

- The **actor**: a parameterized policy that defines how actions are selected (how we act).
- The **critic**: a learned value function that evaluates each action the agent takes in the environment.

At this point we may be wondering whether REINFORCE with baseline is considered an actor-critic method or not, given that it learns both a policy and a state-value function. This is not the case: the key characteristic of actor-critic methods, in fact, is that the state-value function is used to **bootstrap**[5], performing updates at each step, in an online fashion.

Let us rephrase this by considering a transition from a state S_t to another state S_{t+1}. In REINFORCE with baseline, the learned state-value function estimates the value of *only the first state of each transition*. The estimate we see in the update, $\hat{v}(S_t, \boldsymbol{w})$, sets a baseline for the subsequent return G_t, but since it is only based on the state that precedes the transition, it cannot be used to assess the action we took. As we will see, instead, in actor-critic methods the state-value function *is also applied to the second state of the*

[5] In reinforcement learning, bootstrapping means updating a value based on estimates of the value at subsequent time steps.

transition, allowing the critic to give the actor a feedback at each step (we will consider **one-step actor-critic methods**).

Intuitively, knowing immediately whether our actions are good or not allows us to be more effective, enabling quick corrections that could prevent us from going completely off track. As an example, imagine we are writing our dissertation: if we submit it to the supervisor on a chapter-by-chapter basis, we might learn to adapt quickly to the required style, performing only relatively minor corrections. If, on the other hand, we just submit it once it is complete, we might not only have to apply major corrections, but each iteration will also take longer.

8.5.1 One-step actor-critic

Given what we just said, it does not come as a surprise that one-step actor-critic methods replace the full return G_t of REINFORCE with the one-step return $G_{t:t+1}$ (and use a learned state-value function as baseline) as follows[6]:

$$\begin{aligned}
\boldsymbol{\theta}_{t+1} &\doteq \boldsymbol{\theta}_t + \alpha\big(G_{t:t+1} - \hat{v}(S_t, \boldsymbol{w})\big)\frac{\nabla\pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})} \\
&= \boldsymbol{\theta}_t + \alpha\big(R_{t+1} + \hat{v}(S_{t+1}, \boldsymbol{w}) - \hat{v}(S_t, \boldsymbol{w})\big)\frac{\nabla\pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})} \\
&= \boldsymbol{\theta}_t + \alpha\delta_t\frac{\nabla\pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})} \\
&= \boldsymbol{\theta}_t + \alpha\delta_t\nabla \ln\pi(A_t|S_t, \boldsymbol{\theta}
\end{aligned} \tag{8.12}$$

By bootstrapping, *one-step actor-critic methods* become **biased**, as their estimates depend on values at successive time steps. This is not necessarily a bad thing, as in this case it reduces variance and accelerates learning.

It is also important to note that the weights of both the actor and the critic network will be learnt independently, but at the same time.

A pseudocode implementation of one-step actor-critic is shown here:

[6] $G_{t:t+n}$ is the n-step return from $t+1$ to $t+n$.

Algorithm 10: One-step Actor-Critic (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parametrization $\pi(a|s, \boldsymbol{\theta})$
a differentiable state-value function parametrization $\hat{v}(s, \boldsymbol{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\boldsymbol{w}} > 0$
Initialize: policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\boldsymbol{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)
Loop *for each episode* **:**
  Initialize S (first state of the episode)
  Loop *while S is not terminal (for each time step)* **:**
    Select A using policy π
    Take action A, observe S', R
    $\delta \leftarrow R + \hat{v}(S', \boldsymbol{w}) - \hat{v}(S_t, \boldsymbol{w})$ (if S' is terminal, then $\hat{v}(S', \boldsymbol{w}) \doteq 0$)
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha^{\boldsymbol{w}} \delta \nabla \hat{v}(S_t, \boldsymbol{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \delta \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$
    $S \leftarrow S'$

8.6 Continuous action space

Policy-based methods offer practical ways of dealing with large action spaces, even continuous ones, with an infinite number of actions. In those cases, instead of computing learned probabilities for each of the many actions, we learn statistics of the probability distribution, like the *mean* and the *variance* of a Gaussian distribution. Actions will then be sampled from the learned distributions.

Chapter 9

Multiagent learning

Until now we have been studying situations in which only one autonomous agent was present in the environment, learning how it could be trained to act in optimal ways. Often, however, there may be multiple agents in the same environment, interacting with each other and learning. This will be the focus of this chapter.

Let us start with two definitions of multiagent systems:

- *"Multiagent systems are distributed systems of independent actors called agents that are independently controlled but that interact with one another in the same environment".* [Woo02] [TS18]
- *"Multiagent systems are systems that include multiple autonomous entities with (possibly) diverging information".* [SL09]

These definitions highlight a few very important characteristics of multiagent systems:

- Each agent is *independent* and has its own "brain", a (possibly) totally different piece of software controlling the agent.
- Agents can take actions in the environment, possibly changing it for the rest of the agents.
- Each agent may have a limited -and different, possibly even diverging- view of the environment.

Our focus will be the case in which the behavior of the agents is not hard-wired, but it can be changed and improved over time towards a goal. Our definition of multiagent learning will then be *"the study of multiagent sys-*

tems in which one or more of the autonomous entities improves automatically through experience".

It is not difficult to imagine how different characteristics of the environment and of the agents might play a role in the overall complexity of the system. As an example, we might consider different scales of environments (such as a city, an ant colony, or a football team) and agents with different degrees of complexity (like a human, a machine, or an insect). A particularly hot topic of research in the field of multiagent systems is cooperation and trust among agents. An agent set in a city, for example, might give less importance to "behaving well" compared to one set in a small village, as it tends to interact with different agents each time (it does not necessarily need to establish relationships based on trust).

Possibly the most important feature that a system should have to allow agents to learn is **regularity**. To learn, in fact, we must have something that does not change drastically over time, something that can be relied upon to implement a strategy. We will make the assumption that past experience is somehow predictive of future expectations (dealing with non-stationarity is another topic of current research).

In this chapter we will consider five paradigms:

- Online multi-agent reinforcement learning towards an individual utility.
- Online multi-agent reinforcement learning towards social welfare.
- Co-evolutionary learning.
- Swarm intelligence.
- Adaptive mechanism design.

9.1 Online RL towards individual utility

One of the most studied scenarios in multiagent learning is that in which multiple independent agents take actions in the same environment and learn online to maximize their own utility function.

From a formal point of view (leveraging game theory), this can be considered a **repeated normal-form game**. It is repeated because it is based on a certain number of repetitions and normal form because it is represented by means of a matrix[1].

[1]More can be found on Wikipedia: `https://en.wikipedia.org/wiki/Repeated_game`

9.1.1 (Repeated) Normal-form games

To better understand normal-form games, we introduce the *"Prisoner's dilemma"*, a standard example of a game analyzed in game theory that shows why two completely rational individuals might not cooperate, even if it appears that it is in their best interests to do so

The game is as such: two members of a criminal gang are arrested and imprisoned. Each prisoner is in solitary confinement with no means of communicating with the other. The prosecutors lack sufficient evidence to convict the pair on the principal charge, but they have enough to convict both on a lesser charge. Simultaneously, the prosecutors offer each prisoner a bargain. Each prisoner is given the opportunity either to betray the other by testifying that the other committed the crime, or to cooperate with the other by remaining silent. The possible outcomes are:

- If the two prisoners remain silent (they **cooperate**), they will both serve one year in prison (corresponding to a payoff of 2).
- If A betrays B (A **defects**), A will be set free (payoff of 3), and B will serve 3 years in prison (payoff of 0).
- If the prisoners betray each other, they will both serve two years in prison (with a payoff of 1)[2].

Since it is a normal-form game, we can represent it in a matrix:

	Defect	Cooperate
Defect	(1,1)	(3,0)
Cooperate	(0,3)	(2,2)

These types of games were initially introduced as one-shot games, where the players knew each other's reward functions and play the game only once. In this setting, the concept of Nash equilibrium was introduced: a set of actions such that no player has anything to gain from changing their strategy, assuming that the opponent's strategy is fixed. Games can have one or multiple Nash equilibria. In the case of the prisoner's dilemma, the only equilibrium is for both agents to defect.

In the case of repeated normal-form games, players interact with one another multiple times, with the objective of maximizing their expected returns over time.

`https://en.wikipedia.org/wiki/Normal-form_game`

[2]Adapted from `https://en.wikipedia.org/wiki/Prisoner%27s_dilemma`

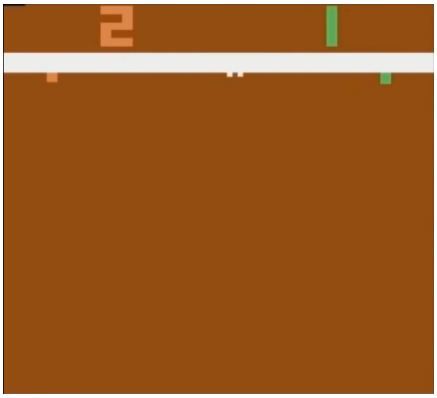


Figure 9.1: Pong being played by two agents

Source: Computational Neuroscience Lab at University of Tartu on YouTube

9.2 Online RL towards social welfare

An alternative paradigm to the one just presented is one in which multiple independent agents take actions in the same environment and learn online to maximize a global utility function. These are also called coordination games, where different players coordinate to achieve a given objective (e.g., maximize the global expected return).

An application of this can be seen in the "Multiagent Cooperation and Competition with Deep Reinforcement Learning" paper by Tampuu et al. [Tam+15] where two agents learn how to play Pong in a way that maximizes the length of the game, making sure none of the two players lose. To achieve this, the agents learn techniques such as not serving the ball or bouncing it in a straight line, as depicted in figure 9.1.

9.3 Co-evolutionary approaches

Evolution can also be used to model and learn agent behavior as well. According to this paradigm, abstract Darwinian models of evolution are applied to refine populations of agents (known as *individuals*) over generations by means of **evolutionary algorithms**.

Evolutionary algorithms are made up of five steps[3]:

[3]More can be read on `https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3` and `https://en.wikipedia.org/wiki/Evolutionary_algorithm`

- Creation of the initial population by randomly generating agents. Each agent is characterized by a set of parameters called **genes** (that are joined into strings called **chromosomes**).
- Calculation of the **fitness score** of the individuals by means of a **fitness function** (e.g., reward obtained in the episode).
- **Selection** of the best-performing individuals to be used as **parents** in reproduction.
- **Generation of offspring** by applying **crossover** and **mutations** on the parents.
- **Replacement** of the least-fit individuals with new individuals.

Repeating the last four steps multiple times allows the refinement of a population. Let us consider the most important parts of evolutionary algorithms more in detail.

9.3.1 Fitness function

The fitness function allows us to evaluate the performance of a certain phenotype (i.e., the actual performance of the behavior of your agent encoded through its genotype). From a more RL-oriented point of view, the fitness function is the equivalent of the performance measure we introduced in chapter 8.

9.3.2 Selection

At each generation, we allow the reproduction of only some of the individuals. To choose which of the individuals will reproduce we can adopt two strategies:

- Proportionate selection, i.e., the chance of reproducing is proportional to the agent's fitness with respect to the population's fitness using the formula:
$$p_i = \frac{fitness_i}{\sum_j fitness_j}$$
- Choosing the top-K individuals of the current generation.

9.3.3 Cross-over

Crossover, also called *recombination*, is a genetic operator used to combine the genetic information of two parents to generate new offspring. It is one

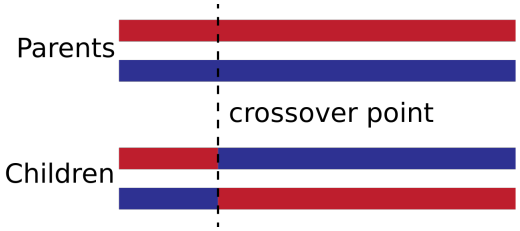


Figure 9.2: Crossover in genetic algorithms

Source: R0oland on Wikipedia

way to stochastically generate new solutions from an existing population and is analogous to the crossover that happens during sexual reproduction in biology.

9.3.4 Mutation

Mutation is a genetic operator used to maintain genetic diversity from one generation of a population of genetic algorithm chromosomes to the next. It alters one or more gene values in a chromosome from its initial state. The classic example of a mutation operator involves a probability that an arbitrary bit in a genetic sequence will be flipped from its original state.

The purpose of mutation in evolutionary algorithms is to introduce diversity into the sampled population. Mutation operators are used in an attempt to avoid local minima by preventing the population of chromosomes from becoming too similar to each other, thus slowing or even stopping convergence to the global optimum.

9.3.5 Evaluation and replacement

At each generation, the fitness of each individual is evaluated, and the population is usually replaced in its entirety by the offspring. Another possible solution is to keep the n "elite" individuals from the previous generation to reduce the effects of mutations or sub-optimal fitness evaluation.

9.3.6 Coevolution

Coevolution is an extension of evolutionary algorithms for domains with multiple agents. By using evolutionary algorithms, we can train a policy to perform a state-to-action mapping. In this approach, rather than update the parameters of a single agent interacting with the environment as it is done

in reinforcement learning, one searches through a population of policies that have the highest fit for the task at hand.

9.4 Swarm Intelligence

One of the most famous techniques in bio-inspired machine learning methods is **swarm intelligence**. It is based on the behavior of social insects (like ants) and it attempts to develop self-organized and decentralized adaptive algorithms.

Swarm intelligence systems consist typically of a large population of simple agents interacting locally with one another and with their environment. The agents follow very simple rules, and although there is no centralized control structure dictating how individual agents should behave, local –and to a certain degree random– interactions between such agents lead to the emergence of "intelligent" global behavior, unknown to the individual agents[4].

Earlier, when we talked about *online RL towards social welfare*, we assumed that the agents had access to a shared utility function. This, however, is not the case for real systems: we need to communicate in some way to achieve coordination.

A possible way of achieving communication is by modifying the environment, with a process known as **stigmergy**. Ants, for example, leave chemical trails behind to trace their paths: this allows them not only to find their way back towards the nest, but also to find the shortest path to do so. The strength of these chemical trails, in fact, decays over time and, since shorter paths allow for a higher number of passages, pheromone levels will be higher on them and they will be chosen more often. This process is mimicked in the *Ant Colony Optimization* algorithm, used to find the shortest path in a network, typically on problems that can be described by means of the *Traveling Salesman Problem*.

9.5 Adaptive Mechanism Design

It is also possible to think of a multi-agent learning setting in which the agents are fixed (or not controllable by us), but the interaction mechanism is to be learned.

[4]Adapted from `https://en.wikipedia.org/wiki/Swarm_intelligence`

To understand this better, let us imagine we are running an auction house: we can control parameters such as the minimum price of the auctioned items, whether to allow simultaneous actions or not and the format of the auction (English auction, Vickrey auction, Dutch auction, etc.). The auction house, of course, is only able to control the rules of interaction between the bidders, and not the bidders themselves. Nonetheless, by learning the way the (auctioning) mechanism work, the auction house can refine the auction parameters over time in order to maximize the price of the items.

Chapter 10

Generative Machine Learning

In the first chapter we said that in this course we would consider *"machines that think, that learn and that create"*, following Herbert Simon's definition of autonomous and adaptive systems. Until now we have focused on the first two points, discussing reinforcement learning and how we can build agents that take actions given a representation of the state. It is now time to move on to machines that *create*.

Can machines even be creative? We will now see a few examples of generative models but let us always keep in mind the definition of **creativity** given by Margaret Ann Boden, a research professor of cognitive science at the University of Sussex: *"Creativity can be defined as the ability to generate novel, and valuable, ideas. Valuable, here, has many meanings: interesting, useful, beautiful, simple, richly complex, and so on. Ideas covers many meanings too: not only ideas as such (concepts, theories, interpretations, stories), but also artifacts such as graphic images, sculptures, houses, and jet engines. Computer models have been designed to generate ideas in all these areas and more"*.

10.1 Generative modeling

Generative models are a class of statistical models that allow for the generation of new data instances. Given a set of data instances X and a set of labels Y, they capture the joint probability $p(X, Y)$, or just $p(X)$ if there are no labels[1]. An example of such models are ones that predict the next word in a sequence.

[1]From `https://developers.google.com/machine-learning/gan/generative`

These models, however, tend to generate data that is just a variation of the one that has been used to train them, not meeting the novelty criterion we have presented earlier on. Given a dataset $\boldsymbol{X}$, produced according to an unknown distribution p_{data}, we want to create a generative model p_{model} that can be used to generate samples that look like they were drawn from p_{data} while also being different enough to be considered novel.

10.1.1 Deep Dream

Deep Dream is a computer vision program created by Alexander Mordvintsev, Christopher Olah and Mike Tika in 2015. It uses a deep convolutional neural network to find and enhance patterns in images creating a dream-like appearance in deliberately over-processed images[2].

While interpretability is still an open question in the deep learning field, image classifiers do not suffer from the same issue. Each layer, in fact, extracts higher and higher-level features of the images in order to make its classification decision. This can be exploited by means of a technique known as *gradient ascent in input space*: we apply gradient *descent* to the value of the input image of a convolutional neural network so as to maximize the response of a specific filter, starting from a blank input image. The resulting input image will be one that the chosen filter is maximally responsive to.

The same technique is used in Deep Dream: one or more layers are chosen, and an input is built so as to maximize the excitement from those layers. If we give the network an actual image as an input instead of noise, the network will then modify the image to maximize the response of the layers we chose, outputting the "hallucinogenic" results Deep Dream is known for.

10.1.2 Generative Adversarial Networks

Performing gradient ascent on a certain input image is not the only way for machines to create something original. Many of us will have probably come across the website "This Person Does Not Exist"[3], where we can see pictures of people that have been generated by a neural network called StyleGAN2.

GAN stands for Generative Adversarial Network, a class of machine learning frameworks designed by Ian Goodfellow and his colleagues in 2014 [Goo+14]. They are made up of two neural networks: a **generative network**, responsible for generating candidates and a **discriminative network** that evaluates

[2]From `https://en.wikipedia.org/wiki/DeepDream`

[3]`https://thispersondoesnotexist.com/`

Figure 10.1: *"A Sunday Afternoon on the Island of La Grande Jatte"* reimagined by DeepDream

Source: GoogleAI Blog

them. The objective of the generative network is to fool the discriminative network into believing that the image was part of the ground truth data used to train it[4].

10.1.3 Text generation

Neural networks can also generate text with great results. The latest and greatest example in these types of networks is GPT-3 (Generative Pre-trained Transformer 3), an autoregressive language model developed by OpenAI and published in May 2020. Trained with 175 billion parameters, it has been able to generate text with a quality so high that it difficult to distinguish from that written by a human.

[4]From: `https://en.wikipedia.org/wiki/Generative_adversarial_network`

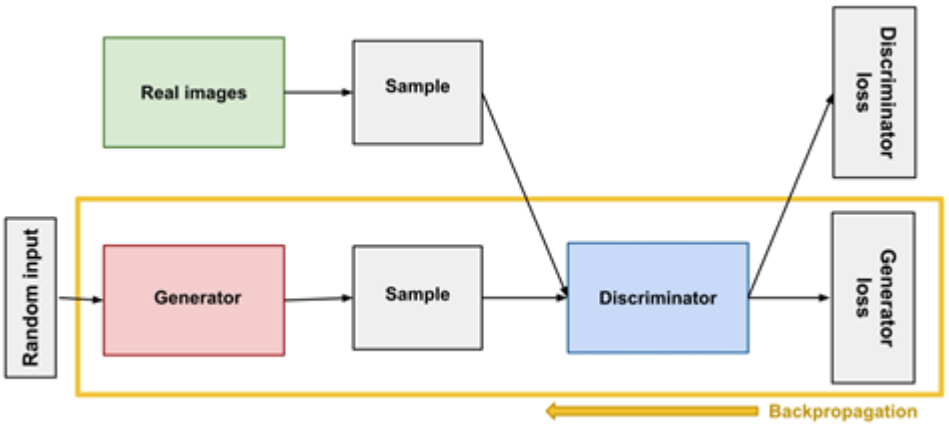


Figure 10.2: Structure of a Generative Adversarial Network
Source: Google Developers website

Chapter 11

Autonomous robots and self-driving cars

As we have seen in the first chapter, the term "autonomous and adaptive systems" is not limited to only software agents –like the ones that we have considered so far– but it also encompasses physically situated **robots**.

After seeing some examples of unmanned robots (see slides 4-9 of the "Autonomous Robots and Self-driving Cars" slide deck), we ask two questions:

- What is the difference between **automation** and **autonomy**?
- Why does it matter that there is a difference between autonomy and automation?

In first approximation we can say that *automation* is about robots as tools, while *autonomy* is about robots as agents. As it often happens, the line between the two is actually much more blurred. We present the following questions to help us discern between automation and autonomy:

- Does the system focus on **executing** plans or on **generating** them?
- Does the system use a **closed** or an **open-world model**?
- Does the system use **deterministic** or **non-deterministic algorithms**?
- Does the system manipulate **signals** or **symbols/concepts**?

In many situations knowing the difference between the two terms is not enough, and we must choose which type of system to develop to best suit our needs. Automatic systems follow a deterministic behavior, making them predictable and, most importantly, testable. This may be sufficient for our

use case, but in some instances we might want a certain degree of adaptability, through which the robot can autonomously handle unexpected changes without stopping. This degree of freedom must be kept in mind, and attention must be paid to the consequences of potential failures, both in terms of risks and liability for the damages that might be caused.

11.1 World models

Robots need to have a representation of the state around them, called a **world model**, to keep track of everything that is needed for their computation. The model may be pre-programmed and hard coded into the robot, or it may be learnt by the robot itself. It will not only include rules, constraints, and maps, but also beliefs and intentions.

If we work under the assumption that we know everything that might happen in our world, we are dealing with a **closed-world model** (e.g., any predicate that is not in our database is considered as false), in which we can specify in its entirety the list of possible states, objects and conditions. If this is not the case, we are working under an **open-world** hypothesis.

Closely related to these assumptions, we have the **frame problem**: how do we correctly identify the things that do not change over time in our world (e.g., walls, trees, etc.) to reduce computation? This is very important as it is linked to the concept of **bounded rationality**, originally introduced by Herbert Simon, stating that the decision-making capabilities of all agents (whether they are human or artificial) is limited by how much information they have, their computational abilities and the amount of time they have available to make a decision. We must also remember that, while a robot may dynamically adapt or change its plan of action to overcome the occurrence of unexpected events, it cannot go beyond what it has been programmed or trained for.

11.2 Self-driving cars

For the part on self-driving cars, refer to the slides.

Bibliography

[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.

[HS86] G. E. Hinton and T. J. Sejnowski. "Learning and Relearning in Boltzmann Machines". In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. Cambridge, MA, USA: MIT Press, 1986, pp. 282–317. ISBN: 026268053X.

[Sha86] G. L. Shaw. "Donald Hebb: The Organization of Behavior". In: *Brain Theory*. Ed. by Günther Palm and Ad Aertsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 231–233. ISBN: 978-3-642-70911-1.

[Cyb89] G. Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals and Systems* 2 (1989), pp. 303–314.

[MP90] Warren S. McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *Bulletin of Mathematical Biology* 52.1 (1990), pp. 99–115. ISSN: 0092-8240. DOI: https://doi.org/10.1016/S0092-8240(05)80006-0. URL: https://www.sciencedirect.com/science/article/pii/S0092824005800060.

[Lin92] Long-Ji Lin. "Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching". In: *Mach. Learn.* 8.3–4 (May 1992), pp. 293–321. ISSN: 0885-6125. DOI: 10.1007/BF00992699. URL: https://doi.org/10.1007/BF00992699.

[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.

[Woo02] Michael Wooldridge. *Introduction to MultiAgent Systems*. John Wiley and Sons, 2002. ISBN: 047149691X.

[HOT06] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. “A Fast Learning Algorithm for Deep Belief Nets”. In: *Neural Comput.* 18.7 (July 2006), pp. 1527–1554. ISSN: 0899-7667. DOI: 10.1162/neco.2006.18.7.1527. URL: https://doi.org/10.1162/neco.2006.18.7.1527.

[SL09] Yoav Shoham and Kevin Leyton-Brown. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge, UK: Cambridge University Press, 2009. ISBN: 978-0-521-89943-7.

[Mni+13] Volodymyr Mnih et al. “Playing Atari with Deep Reinforcement Learning”. In: (2013). cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013. URL: http://arxiv.org/abs/1312.5602.

[Goo+14] Ian J. Goodfellow et al. *Generative Adversarial Networks*. cite arxiv:1406.2661. 2014. URL: http://arxiv.org/abs/1406.2661.

[Mni+15] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 00280836. URL: http://dx.doi.org/10.1038/nature14236.

[Tam+15] Ardi Tampuu et al. “Multiagent Cooperation and Competition with Deep Reinforcement Learning”. In: *CoRR* abs/1511.08779 (2015). arXiv: 1511.08779. URL: http://arxiv.org/abs/1511.08779.

[TS18] K. Tuyls and P. Stone. “Multiagent Learning Paradigms”. In: *Multi-Agent Systems and Agreement Technologies*. Ed. by Francesco Belardinelli and Estefanía Argente. Cham: Springer International Publishing, 2018, pp. 3–21. ISBN: 978-3-030-01713-2.