Introduction to Reinforcement Learning with Applications in Geomechanics

Alexandros Stathas, Diego Gutiérrez-Oribio and Ioannis Stefanou

Nantes Université, École Centrale Nantes, CNRS, GeM, UMR 6183, F-44000 Nantes, France

Reinforcement learning (RL) is a subfield of Machine Learning (ML) that focuses on the development of software agents that are capable of making optimal decisions in dynamic and uncertain environments. It is a powerful learning paradigm that enables machines to learn from their own interactions with the environment, rather than relying on explicit instructions or labeled datasets. In RL, an agent learns through a trial-and-error process, where it takes actions in an environment, receives feedback in the form of rewards or penalties, and adjusts its behaviour to maximize the cumulative reward over time. In this chapter, we present the basic concepts of RL and introduce model-based and model-free methods for deterministic and stochastic cases. Then, we present two applications of Geomechanics: the spring-slider and a geothermal reservoir. On both systems, an RL algorithm is used to design a controller able to prevent seismic events.

1 Introduction

In this chapter, we introduce *Reinforcement Learning* (RL), which is a family of *Machine Learning* (ML) algorithms that allow the learner (*i.e.* software *agent*) to determine an *optimal behaviour* inside an *environment*[1] that will provide the maximum cumulative *reward*[2] (see Figure 1).

RL is neither a *Supervised Learning* nor an *Unsupervised Learning* technique since the agent does not require the existence of a labelled dataset to train on, nor it extracts underlying patterns from already available data (see Figure 2). Instead, the agent

[1] The environment is the set of all states, actions and rewards the agent can take.

[2] A reward is feedback signal from the environment (real value number) reflecting how well the agent is performing.

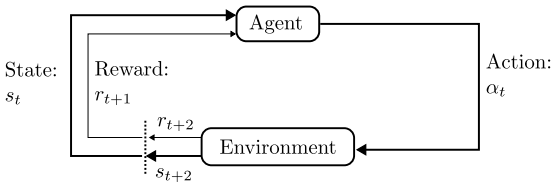


Figure 1: Schematic representation of the key components of *Reinforcement learning*. The agent interacts with the *environment* through *actions* and receives *feedback* through *observations* and *rewards*.

learns by performing *actions*[3] inside the environment and getting a reward. The objective of the RL algorithms is to measure the *value* of the actions taken by the agent and formulate a *policy* (*i.e.* an algorithm that chooses an action from an available set of actions at each state) so that it maximises the cumulative reward.

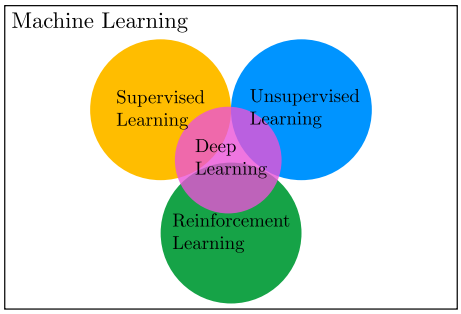


Figure 2: Schematic diagram between the Supervised learning, Unsupervised learning and Reinforcement learning techniques of Machine Learning. Deep learning with the help of neural networks can be applied to all these techniques.

Initial advances in RL took place in the framework of *Dynamic Programming* (DP) through the use of the *Bellman equations* and the *Bellman optimality conditions* [Bel54]. These methods allow us to construct powerful algorithms that given enough information about the environment, are capable of planning a suitable winning strategy (*i.e.* a *policy*) that will maximize the reward they receive [Sut99]. In essence, the problem or RL can be cast into a form of finding the optimal policy function, $\pi(s_t)$[4] that maximizes the expected accumulated reward.

This type of problem can be solved iteratively[5] by applying the Bellman optimality conditions and the fundamental algorithms of *Value Iteration*, *Policy Iteration* and

[3] Input from an agent to the environment.

[4] Equivalently a stochastic policy, $\pi(\alpha_t|s_t)$, indicating the probability of the agent performing action α_t given the state s_t can be used.

[5] Such a method is called fixed point iteration.

Generalized Policy Iteration [Sut99, BT95].

The challenge in these fundamental methods of DP (value iteration and policy iteration) is their scalability to real-world applications. Although game examples such as chess or backgammon have a relatively small number of discrete states the agent can find itself in and possible actions it can perform, real world examples are much more complex in the sense that their state and action space can be continuous, rendering the classical techniques intractable.

A solution to this problem comes from the *Function Approximation* branch of ML. These are techniques of supervised learning (unsupervised learning can also be used), which estimate the state value and action value functions using vectors of interpolation weights.

In recent years, the significant increase in computational power together with the advent of *Deep Artificial Neural Networks* (DANNs) and newer programming environments (see [C$^+$15, ABC$^+$16, PGM$^+$19], among others), which allow for the conceptually simpler implementation of such complicated nonlinear interpolation functions, permitted the combination of DANNs with the classical DP techniques (see [Gér22, VSS$^+$19, Lap18]). This combination of modern computational power and past wisdom allowed researchers to significantly advance in the field of RL by applying their advanced algorithm architectures to previously intractable problems. These methods have been applied successfully to applications that involve:

- **Control applications:** These include a signal the agent observes from the environment and/or a reward that informs the agent that an action needs to be taken. This simple framework applies both to simple applications such as thermostat or pressure switches controllers [Sut90, SBW92] and to fancier controllers, like controlling the movement of robotic components [SSZ$^+$22], or even the movement of plasma in Tockamak fusion reactors! [DFB$^+$22].
- **Game applications:** These include software agents that learn to play optimally in games involving high dimensional state and action spaces such as Go [SHS$^+$17, Pic17] and Starcraft [VEB$^+$17].
- **Generative AI:** These applications of RL include a software agent that learns to create human interpretable content. This allows for the creation of AI art [Euc23], customer service and all-around helper Chatbots, like ChatGPT [Ope23, GBGM23].

Considering the use of RL in geomechanics, we focus on a recent control application from fault mechanics based on earthquake control (see [Ste19, PS21, ST22, GOTSP23, GOOSP23, GOSP22], among others). Roughly speaking, earthquakes are dynamic instabilities caused by frictional weakening. Such instability leads to fast-slip (earthquake-like behaviour) creating waves that travel through the earth's crust and become catastrophic events. The cited works study the influence of injected fluid to avoid such behaviour, by designing strategies based on control theory. Here we solve this problem using RL.

The chapter is structured as follows. We first introduce the fundamental components of RL and their relations to the Bellman equations presenting an example on *policy evaluation*: a miner trapped inside a trembling mine finding his way to the gold (reward). Next, we present the fundamental *model-based* methods of DP: *policy evaluation, state value iteration and policy iteration* algorithms. For the stochastic frame, we show the most known *model-free* methods, which are useful for more practical applications when the states and actions are large and the probability distribution is unknown. Then we extend the applicability of the above methods with the use of function approximation tools, introducing policy approximation methods and discussing the application of policy gradient for finding an optimal policy. At last, we comment on the actor-critic popular approach in RL, which is used for problems with continuous state and action spaces, which is the case for the two geomechanics applications that we present. Inspired by [PS21], two earthquake control applications using a *Deep Neural Network Actor-Critic* (DNNAC) architecture are shown. The first application is the so-called spring-slider system, whereas the second is a geothermal reservoir. In both cases, the RL algorithm should have to adequate the gains of a linear controller, adjusting the fluid pressure inside the earth's crust. This will lead to an aseismic response of both systems, *i.e.*, the prevention of seismic events.

2 Reinforcement Learning: The basics

2.1 Basic Definitions: Deterministic case

Consider the game of a miner of Left Figure 3. The miner (shown as a black dot and known as the *agent*) moves inside the mine (known as the *environment*) to find gold (the *reward*) while avoiding the cliff (known as *penalties*). Each movement constitutes a different *action*. For this example, the miner can perform four actions: go right ($\rightarrow$), go up ($\uparrow$), go left ($\leftarrow$) and go down ($\downarrow$). Therefore, the movements will be discrete. The environment is constituted of 16 cells defined as C1, C2, ..., C16 and known as *states*. The *state space* of the environment is denoted as $\mathcal{S} = \{\text{C1,C2,...,C16}\}$.

The game will finish when the miner reaches either the cliff (located at cell C1) or the gold (located at cell C16). Due to their special nature, these two cells will be known as *terminal states*. When the miner reaches a terminal state, it completes an *episode*.

We define as *trajectory* the set of states, s_t, actions, α_t, and rewards, r_t, taken on every time step, t, of an episode (see Right Figure 3). A trajectory is then denoted as

$$T_r = \{s_0, \alpha_0, r_1, s_1, \alpha_1, ..., s_{T-1}, \alpha_{T-1}, r_T\}, \tag{1}$$

for an episode of time $t = 0$ to $t = T$. Note how time is also a discrete variable. Furthermore, the first state, s_0, does not provide any reward and the final state, s_T, and final action, α_T, are not written because it is when the game finishes.

As in every game, we would like to receive the biggest reward in each episode. For that purpose, we will define the total accumulated reward, G_t, at the time t as the sum

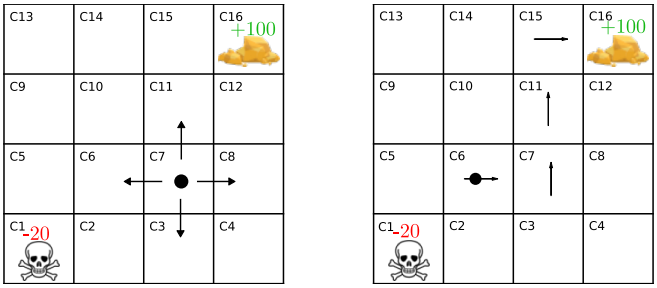


Figure 3: Left: The mine (*environment*) and the miner (*agent*, black dot). The miner can move in the surrounding cells known as *states*. In order to move, the miner can perform four *actions*: go right ($\rightarrow$), go up ($\uparrow$), go left ($\leftarrow$) and go down ($\downarrow$). Two *terminal states* are present, C16, with a positive reward of +100 and C1 with a negative reward of -20. Right: A trajectory, T, followed by the agent. The miner started at cell C6 and followed a path to reach the gold, receiving a reward after taking an action. For this episode, the trajectory is defined as $T_r = \{s_0 = \text{C6}, \alpha_0 = \rightarrow, r_1 = 0, s_1 = \text{C7}, \alpha_1 = \uparrow, r_2 = 0, s_2 = \text{C11}, \alpha_2 = \uparrow, r_3 = 0, s_3 = \text{C15}, \alpha_3 = \rightarrow, r_4 = 100\}$.

of all the rewards from the reward r_{t+1} until the reward of the terminal state, r_T, *i.e.*

$$G_t = \sum_{k=0}^{T-1} r_{t+k+1}, \tag{2}$$

where $r_{t+k+1} = r_{t+k+1}(s_{t+k}, \alpha_{t+k})$ is the intermediate reward between the two adjacent states s_{t+k} and s_{t+k+1} in the trajectory. We note that $T-1$ is the state before reaching the end of the game. Moreover we follow the convention that rewards later than the final reward r_T at the terminal time T are set to zero (*i.e* they are ignored). Between two states the total accumulated reward can also be written as

$$G_t = r_{t+1} + \sum_{k=0}^{T-2} r_{t+k+2} \tag{3}$$

where $r_{t+1} = r_{t+1}(s_t, \alpha_t)$ is the intermediate reward between the current state s_t and the next state s_{t+1} in the trajectory.

2.1.1 Policy evaluation

In order to move inside the mine, the miner will follow a set of rules specifying which *action* to perform at each cell (*state*) of the mine. This mapping between states and actions is called a *policy*. In this example, we will consider *deterministic* policies (*e.g.*, if the miner is in state $s_t = \text{C2}$ do action $\alpha_t = \uparrow$) and they are denoted as $\alpha_t = \pi(s_t)$.

Based on the policy of the miner we can expect that different states exhibit different cumulative rewards, *i.e.*, starting from an initial state, the miner might either reach the gold (C16) or fall in the cliff (C1). Moreover, depending on the given policy, there could be two problematic cases. In the first case, the miner might get stuck in a loop between two states and in the second case, the miner may hit the boundaries of the mine and stay there indefinitely as well (see Figure 4).

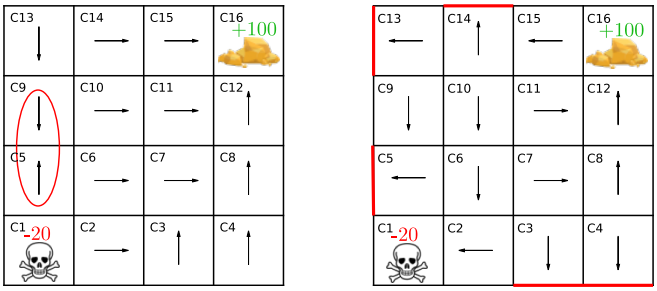


Figure 4: Problematic cases of the environment. Left: The agent falls in a loop between two states. Right: The policy forces the agent to hit the boundaries.

In order to handle these cases, we will force the episode to finish if the time reaches a time limit $T_{\max}$. This will ensure that every episode will terminate and it won't stay in an infinite loop. Moreover, we penalize the agent with a negative reward -10 if it chooses to hit the boundaries of the mine.

We can evaluate the expected accumulated reward G_t during a trajectory where the miner follows a given policy $\pi(s_t)$, starting from the state s_t. This will be known as the *value of the state*:

Definition 1 *[SB18] The value of the state, $V_\pi(s_t)$, is written as*

$$V_\pi(s_t) = \sum_{k=0}^{T-1} r_{t+k+1}(s_{t+k}, \pi(s_{t+k})), \quad \text{or}$$

$$V_\pi(s_t) = r_{t+1}(s_t, \pi(s_t)) + \sum_{k=0}^{T-2} r_{t+k+2}(s_{t+k+1}, \pi(s_{t+k+1})). \tag{4}$$

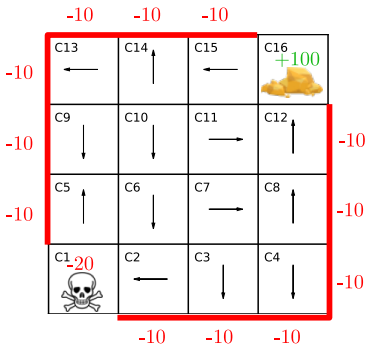
From the definition, we note that the actions taken at each subsequent state s_{t+k} are prescribed by the policy $\pi(s_{t+k})$. The same is also true for the rest of the future actions $\pi(s_{t+k+1})$ where they are only dependent on the future state, s_{t+k+1}, and not on the past states. We will see in the general stochastic setting of DP, that this is a fundamental assumption known as the *Markov property*.

Based on the definition of the *value of the state* in (4), we can replace the previous sum over the future rewards with the value of the future state $V_\pi(s_{t+1})$ as

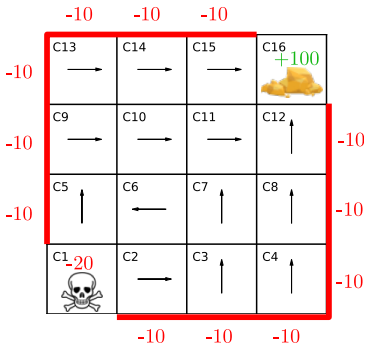
$$V_\pi(s_t) = r_{t+1}(s_t, \pi(s_t)) + V_\pi(s_{t+1}). \tag{5}$$

Such equation is known as the *Bellman equation* for the value function $V_\pi(s_t)$.

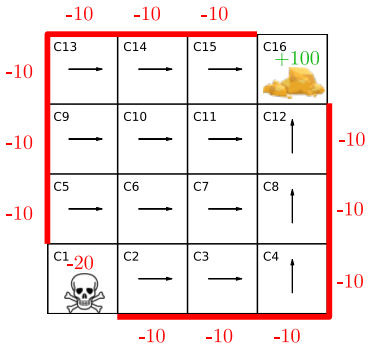
Let's consider now three deterministic policies, $(\pi_{\mathrm{A}}(s_t), \pi_{\mathrm{B}}(s_t), \pi_{\mathrm{C}}(s_t))$, the miner can follow in the environment (see Figure 5). Policy $\pi_{\mathrm{A}}(s_t)$ could be considered as bad because there have three states (C2, C6, C10) that lead to the cliff. Policy $\pi_{\mathrm{B}}(s_t)$ and policy $\pi_{\mathrm{C}}(s_t)$ seem to be good because all of the states will lead to the gold. One may think, which is the best of the three?



(a) Policy $\pi_{\mathrm{A}}(s_t)$



(b) Policy $\pi_{\mathrm{B}}(s_t)$



(c) Policy $\pi_{\mathrm{C}}(s_t)$

Figure 5: Example of three different policies.

We will define a rule to check which policy is better depending on the value of the state as follows:

Definition 2 *A policy $\pi'(s_t)$ is better than policy $\pi(s_t)$ if for every state, s_t, the value of the state under policy π' is larger or equal to the value of the state under policy π, i.e., $V_{\pi'}(s_t) \geq V_{\pi}(s_t)$ for every s_t.*

With this definition, we can compare our three policies in order to check which may be the best. We only need to evaluate the value of each state for each policy, $(V_{\pi_{\mathrm{A}}}(s_t), V_{\pi_{\mathrm{B}}}(s_t), V_{\pi_{\mathrm{C}}}(s_t))$, applying sequentially equation (5) for each state $s_t \in \mathcal{S}$. The results can be found in Table 1 in the columns below $\gamma = 1$ (whose definition will be clarified a few lines later).

	$\gamma = 1$			$\gamma = 0.9$	
	$V_{\pi_{\mathrm{A}}}(s_0)$	$V_{\pi_{\mathrm{B}}}(s_0)$	$V_{\pi_{\mathrm{C}}}(s_0)$	$V_{\pi_{\mathrm{B}}}(s_0)$	$V_{\pi_{\mathrm{C}}}(s_0)$
C1	-20.0	-20.0	-20.0	-20.0	-20.0
C2	-20.0	+100.0	+100.0	+65.0	+65.0
C3	-10.0	+1000.	+100.0	+72.0	+72.0
C4	-10.0	+100.0	+100.0	+81.0	+81.0
C5	0.0	+100.0	+100.0	+65.0	+65.0
C6	-20.0	+100.0	+100.0	+59.0	+72.0
C7	+100.0	+100.0	+100.0	+81.0	+81.0
C8	+100.0	+100.0	+100.0	+90.0	+90.0
C9	0.0	+100.0	+100.0	+72.0	+72.0
C10	-20.0	+100.0	+100.0	+81.0	+81.0
C11	+100.0	+100.0	+100.0	+90.0	+90.0
C12	+100.0	+100.0	+100.0	+100.0	+100.0
C13	-10.0	+100.0	+100.0	+81.0	+81.0
C14	-10.0	+100.0	+100.0	+90.0	+90.0
C15	-10.0	+100.0	+100.0	+100.0	+100.0
C16	+100.0	+100.0	+100.0	+100.0	+100.0

Table 1: Comparison between the values of the state value function $V_{\pi}(s)$ for deterministic policies $(\pi_{\mathrm{A}}(s_t), \pi_{\mathrm{B}}(s_t), \pi_{\mathrm{C}}(s_t))$. Without the discount parameter ($\gamma = 1$), we can see that policy A is the worst but we cannot decide which is better between policies B and C. Including the discount parameter ($\gamma = 0.9$), we can finally say that policy C is the best of the three because it has a larger accumulated reward value for state (s_0 =C6), while for the rest of the states the values remain the same.

We can confirm that the worst policy is $\pi_{\mathrm{A}}(s_t)$ because it presents lower values for all states. However, we cannot decide between policies $\pi_{\mathrm{B}}(s_t), \pi_{\mathrm{C}}(s_t)$ because they have the same values at every state. Therefore, we need to include something that unties the result in this kind of situation. For this purpose, we will consider also the number of moves the miner needed for getting the reward. To do this we will apply a *discount*

rate, γ, to the cumulative reward of each state as

$$\begin{aligned} V_\pi(s_t) &= \sum_{k=0}^{T-1} \gamma^k r_{t+k+1}(s_{t+k}, \pi(s_{t+k})), \\ &= r_{t+1}(s_t, \pi(s_t)) + \gamma \sum_{k=0}^{T-2} \gamma^k r_{t+k+2}(s_{t+k+1}, \pi(s_{t+k+1})), \\ &= r_{t+1}(s_t, \pi(s_t)) + \gamma V(s_{t+1}). \end{aligned} \tag{6}$$

You can see that if $\gamma \in (0, 1)$, the value of the state at every time step will decrease, penalizing then the number of moves done by the miner before getting the reward. Calculating again the value of the state for policy B and C (see results in Table 1 under $\gamma = 0.9$), we can see that policy C is better because it has at least one value of the state bigger than policy B at the state C6, while all the others are the same.

The question that we face now is:

How can I know if the policy $\pi_C(s_t)$ is the best policy that I could find?

In other words:

Can I find an optimal policy $\pi^\star$ for which the cumulative reward at the end of the episode becomes maximum?

This question will be answered in the following sections.

2.1.2 Action Value Function

We will introduce another value function called the *action value function*, which indicates the value[6] of a specific action, α_t, given that the miner is at state s_t.

Definition 3 *The action value function, $Q_\pi(s_t, \alpha_t)$, at a state, s_t, corresponds to the immediate reward, $r_{t+1}(s_t, \alpha_t)$, that will be obtained performing one of the available actions, α_t, at such state, plus the expected value of the state according to a given policy starting from the state s_{t+1}, i.e.,*

$$\begin{aligned} Q_\pi(s_t, \alpha_t) &= r_{t+1}(s_t, \alpha_t) + \gamma \sum_{k=0}^{T-2} \gamma^k r_{t+k+2}(s_{k+t+1}, \pi(s_{k+t+1})), \\ &= r_{t+1}(s_t, \alpha_t) + \gamma V_\pi(s_{t+1}). \end{aligned} \tag{7}$$

From the definition of the value of the state, $V_\pi(s_t)$, in (6) and the action value function, $Q_\pi(s_t, \alpha_t)$, in (7), we can see that both expressions are the same at s_{t+1} and when $\alpha_t = \pi(s_{t+1})$, *i.e.*, $V_\pi(s_{t+1}) = Q_\pi(s_{t+1}, \pi(s_{t+1}))$. Therefore, we can get a recursive expression of the action value function as

$$Q_\pi(s_t, \alpha_t) = r_{t+1}(s_t, \alpha_t) + \gamma Q_\pi(s_{t+1}, \pi(s_{t+1})). \tag{8}$$

[6] also known as the quality

In the case of our miner, from the Policy B of Figure 5, we could calculate the action value for the state s_t = C6 at every possible action $\alpha_t = \{\rightarrow, \uparrow, \leftarrow, \downarrow\}$ (see Figure 6). We took again $\gamma = 0.9$ so we still get the same values of $V_\pi(s_t)$ shown in Table 1. The action value functions are then calculated as

$$
\begin{aligned}
Q_{\pi_B}(C6, \rightarrow) &= r_{t+1}(C6, \rightarrow) + 0.9V_{\pi_B}(C7) = 0 + 0.9(81) = 72, \\
Q_{\pi_B}(C6, \uparrow) &= r_{t+1}(C6, \uparrow) + 0.9V_{\pi_B}(C10) = 0 + 0.9(81) = 72, \\
Q_{\pi_B}(C6, \leftarrow) &= r_{t+1}(C6, \leftarrow) + 0.9V_{\pi_B}(C5) = 0 + 0.9(65) = 59, \\
Q_{\pi_B}(C6, \downarrow) &= r_{t+1}(C6, \downarrow) + 0.9V_{\pi_B}(C2) = 0 + 0.9(65) = 59.
\end{aligned} \tag{9}
$$

We can see that there are at least two actions $(\uparrow, \rightarrow)$ that give more cumulative reward than what the current policy says (we will obtain 59 if we take the action $\leftarrow$ of Policy B). In other words, there is room for improvement! This is called *Policy Improvement*:

Definition 4 *[SB18] Any policy π for which $V_\pi(s_t) \leq Q_\pi(s_t, \alpha_t)$, $\forall s_t \in \mathcal{S}$, is a policy that we can improve on. That is we can choose action α_t at state s_t, formulating a new policy π' in the process, for which $V_\pi(s_t) \leq V_{\pi'}(s_t)$.*

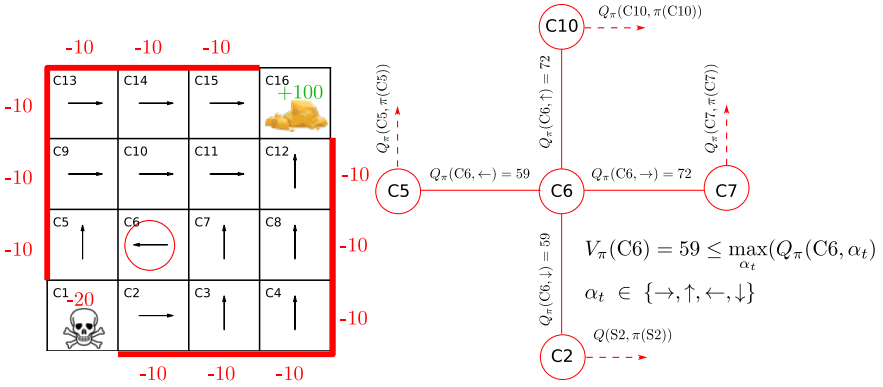


Figure 6: Evaluation of the action value function starting at state s_t = C6 with $\gamma = 0.9$. For every possible action in state C6 ($\alpha_t = \{\rightarrow, \uparrow, \leftarrow, \downarrow\}$), we evaluate $Q_{\pi_B}(\text{C6}, \alpha_t)$ as the immediate reward for performing this action and then moving according to policy $\pi_B(s_t)$.

The task now is to find a method to choose the right action at every state, s_t, obtaining the maximum accumulated reward at the end of every episode.

2.1.3 Bellman's Optimality conditions

As we saw before, from all the possible actions, α_t, that we can take in a state, s_t, there is one special action that maximises the immediate reward. Bellman showed that if we recursively take the best action at every state of the episode, then the expected accumulated reward for each state becomes maximum [Bel54]. In other words, the

policy that maximizes the accumulated reward at each state[7] is the optimal policy, $\pi^\star(s_t)$, that we were looking for, *i.e.*, $V_{\pi^\star}(s_t) = \sum_{k=0}^{T} r_{t+k+1}(s_{t=K}, \pi^\star(s_{t=k})) = \sum_{k=0}^{T} r_{t+k=1} \max_{\alpha_{t+k}} r(s_{t+k}, \alpha_{t+k})$. This means that the value of the state and the action value function will have the same value if we follow the optimal policy, *i.e.*,

$$V_{\pi^\star}(s_t) = \max_{\alpha_t} Q_{\pi^\star}(s_t, \alpha_t), \tag{10}$$

which, according to equation (7), leads to

$$V_{\pi^\star}(s_t) = \max_{\alpha_t} r_{t+1}(s_t, \alpha_t) + \gamma V_{\pi^\star}(s_{t+1}). \tag{11}$$

On the other hand, if we take the optimal policy in equation (7), it turns into

$$Q_{\pi^\star}(s_t, \alpha_t) = r_{t+1}(s_t, \alpha_t) + \gamma V_{\pi^\star}(s_{t+1}), \tag{12}$$

which together with equation (10) reads as

$$Q_{\pi^\star}(s_t, \alpha_t) = r_{t+1}(s_t, \alpha_t) + \gamma \max_{\alpha_{t+1}} Q_{\pi^\star}(s_{t+1}, \alpha_{t+1}). \tag{13}$$

Equations (11) and (13) are known as *Bellman's Optimality conditions* and they can be used iteratively to find the optimal state and action value functions. Moreover, the $\max$ operator on the right-hand side of the equations prevails over any policy π that is used for the evaluation of $V_{\pi^\star}(s_t), Q_{\pi^\star}(s_t, \alpha_t)$. As a result, the optimal values of the state and action, $V^\star(s_t), Q^\star(s_t, \alpha_t)$ are independent of the optimal policy! In othger words, there can be many policies $\pi^\star$ that lead to the optimal values of the state and action value functions $(V_{\pi^\star}(s_t), Q_{\pi^\star}(s_t, \alpha_t))$, but these values are unique and equal to $(V^\star(s_t), Q^\star(s_t, \alpha_t))$. We will present the basic algorithms for finding such optimal state and action value functions in the following.

2.1.4 Maximizing the reward: Model-based Methods

Based on the evaluation of the optimal policy and the *Bellman optimality conditions* we can discern two types of algorithms for maximizing the reward. These are the *Policy iteration algorithms* and the *Value iteration algorithms* (see Figure 7)

Both methods work excellently in deterministic or stochastic cases where the number of available states and actions is small. However, these methods require complete knowledge of the states, actions and rewards of the environment to work. This is why they are called model-based methods[8]. For our miner example, we need to know all possible states inside the mine and their corresponding rewards. The agent (miner) must *plans* its actions before they are taken.

We will now explain in more detail how both algorithms work.

[7] *i.e.* the value of the state $V_\pi(s_t)$

[8] We note that the notion of *model* in RL differs from that of supervised or unsupervised learning in the sense that the RL model is used to provide states and rewards to the agent and not to perform a regression task or find a pattern in the data (*i.e.*, in RL we train the agent, not the model). To describe the term model used in (supervised and unsupervised learning) the term *function approximator* is used instead (see [SB18] and Section 2.3).

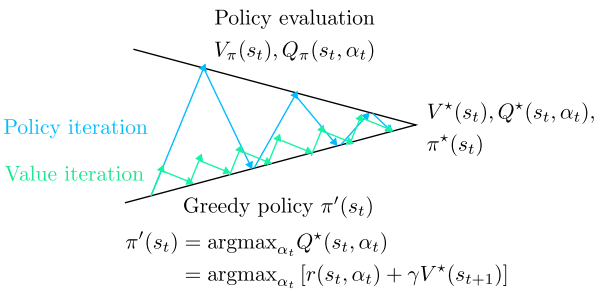


Figure 7: The basic learning algorithms for finding the unique optimal value and action functions, $V^\star(s_t), Q^\star(s_t, \alpha_t)$, and one optimal policy, $\pi^\star$, (of many): Policy Iteration (blue)) and Value iteration (green). The difference between the two lies in the update strategy. Policy iteration calculates a better policy for all states and then evaluates the value functions according to this policy. Value iteration updates only one action at a time based on the Bellman optimality condition.

Policy Iteration (PI) method

In this algorithm, we achieve the optimal action values and the respective optimal policy by modifying the policy $\pi(s_t)$. In particular, starting from a random initialization of $\pi(s_t)$, we evaluate the value function over each state $V_\pi(s_t)$. Then we calculate the action value function for each state-action pair $Q_\pi(s_t, \alpha_t)$.

Next, we compare between the different $Q_\pi(s_t, \alpha_t)$ and find the value of $\max_{\alpha_t} Q_\pi(s_t, \alpha_t)$. Then, we extract the action that maximises the action value function $Q_\pi(s_t, \alpha_t)$, *i.e.*, $\alpha'_t = \text{argmax}_{\alpha_t} Q_\pi(s_t, \alpha_t)$. Finally, we replace the action a_t in state s_t with the better action α'_t. This leads to a new policy $\pi'(s_t)$.

Once the policy has been updated, we repeat the evaluation of the value functions $V_{\pi'}(s_t), Q_{\pi'}(s_t)$. The next iterations for the next prediction of the optimal state and action value $V^{k+1}_{\pi^\star}(s_t), Q^{k+1}_{\pi^\star}(s_t, \alpha_t)$ functions are given by

$$V^{k+1}_{\pi^\star}(s_t) = r_{t+1}(s_t, \pi'(s_t)) + \gamma V^k_{\pi'}(s_t), \tag{14}$$

$$Q^{k+1}_{\pi^\star}(s_t, \alpha_t) = r_{t+1}(s_t, \alpha_t) + \gamma Q^k_{\pi'}(s_{t+1}, \pi'(s_{t+1})). \tag{15}$$

When the policy $\pi'(s_t)$ converges for all states, the PI algorithm has converged to an optimal policy $\pi^\star(s_t) = \pi'(s_t)$ and yields the optimal $V_{\pi^\star}(s_t),\ Q_{\pi^\star}(s_t, \alpha_t)$.

We emphasize that the optimal state value function, $V_{\pi^\star(s_t)}$, and the optimal action value function, $Q_{\pi^\star}(s_t, \alpha_t)$, are unique. However, the policies that yield the optimal reward are not, *i.e* $V^\star(s_t) = V_{\pi^\star}(s_t),\ Q^\star(s_t, \alpha_t) = Q_{\pi^\star}(s_t, \alpha_t)$. Moreover, we note that in this algorithm the evaluation of the future state-action value functions $(V^k_{\pi'}(s_{t+1}), Q^k_{\pi'}(s_{t+1}, \alpha_{t+1}))$ does not happen according to the initial policy $\pi(s_t)$

(see also section 2.1.1) but the new updated policy $\pi'(s_t)$ is used in each iteration (see also Figure 7).

Application to the miner example

We can apply the above algorithm for finding the optimal state and action-value functions, $V^\star(s_t), Q^\star(s_t, \alpha_t)$, in the example of the miner. In this case, we see in Figure 8, that the policy iteration algorithm converges after four iterations to the optimal values $V^\star(s_t)$ for each state s_t inside the mine (see Table 2).

(a) Initial random policy

(b) First policy iteration

(c) Second policy iteration

(d) Final optimal policy

Figure 8: Policy iteration example for the example of the miner. The algorithm evaluates according to the policy $\alpha_t = \pi_i(s_t)$ the accumulated reward starting from each state separately $V_{\pi_i}(s_t) = V_{\pi_i}(s_t, \pi^i(s_t))$ and compares with the accumulated reward for the different actions at each state $Q_{\pi_i}(s_t, \alpha_t)$. Then the algorithm updates the actions in order to optimise $Q_{\pi_i}(s_t, \alpha_t)$.

	$V_{\pi^0}(s_t)$	$V_{\pi^1}(s_t)$	$V_{\pi^2}(s_t)$	$V_{\pi^\star}(s_t)$
C1	-20	-20	-20	-20
C2	+65	+65	+65	+65
C3	+72	+72	+72	+72
C4	+81	+81	+81	+81
C5	+65	+65	+65	+65
C6	+59	+72	+72	+72
C7	+81	+81	+81	+81
C8	+90	+90	+90	+90
C9	+72	+72	+72	+72
C10	+81	+81	+81	+81
C11	+90	+90	+90	+90
C12	+100	+100	+100	+100
C13	+81	+81	+81	+81
C14	+90	+90	+90	+90
C15	+100	+100	+100	+100
C16	+100	+100	+100	+100

Table 2: Values of the state value function for the different iterations of the policy iteration algorithm for the example of the miner (the discount parameter was chosen $\gamma = 0.9$). The optimal policy is reached after the first iteration. The final optimal policy and the first iteration are both optimal policies because they lead to the optimal values of the state. The change in the actions happens because of the way the algorithm selects the maximum action value (the right action, $\rightarrow$, is selected first).

Generalised Policy Iteration methods (GPI)

In the previous algorithm (PI) we evaluated the state values and action values of the optimal policy $V_{\pi'}(s_t), Q_{\pi'}(s_t, \alpha_t)$ by iteratively changing the policy $\pi'(s_t)$ which is used for the evaluation of $V_{\pi'}(s_{t+1}), Q_{\pi'}(s_{t+1}, \alpha_{t+1})$. We will now use Bellman's optimality condition to perform the updates on the values of the state value and action value functions without performing a policy update. These learning algorithms are called GPI methods.

These methods stem from the property of the optimal state value, $V^\star(s_t)$, and action value, $Q^\star(s_t, \alpha_t)$, to be independent of the optimal policy $\pi^\star$. This property extends also to the updates of the GPI algorithm, therefore, we don't need to explicitly guess the optimal policy $\pi^\star$ in order to obtain the optimal $V^\star(s_t)$, $Q^\star(s_t, \alpha_t)$. We can instead apply the maximum operator at each state repeatedly, find the optimal values $V^\star(s_t)$, $Q^\star(s_t, \alpha_t)$ and then deduce the optimal policy $\pi^\star$. We present below the methods of the *Value iteration* and *Q-Value iteration*.

- **Value Iteration method**

The simplest method to use in order to find the optimal $V^\star(s_t)$ and a corresponding

optimal policy $\pi^\star$, is to randomly initialize the values of $V^0(s_t)$ and then iterate over each state, s_t, and update the value of the state value function $V^{k+1}(s_t)$ according to the rule,

$$V^{k+1}(s_t) = \max_{\alpha_t} \left[r_{t+1}(s_t, \alpha_t) + \gamma V^k(s_{t+1}) \right], \tag{16}$$

where $V^{k+1}(s_t), V^k(s_{t+1})$ are the next update of the state value function at the current state, s_t, and the current update of the state value function at the subsequent states, s_{t+1}, respectively.

After sufficient iterations over each state choosing the best action α_t at each state, the values of the state value function will converge to the optimal values, *i.e.*, $V^{k+1}(s_t) \rightarrow V^\star(s_t)$.

At this point, the optimal policy, $\pi^\star$, can be found by choosing to move along the path of optimal state values and therefore to higher cumulative reward, *i.e.*, $\pi^\star = \text{argmax}_{\alpha_t} \left[r_{t+1}(s_t, \alpha_t) + \gamma V^k_\pi(s_{t+1}) \right]$.

- **Q-Value Iteration\Greedy-Policy Improvement algorithm**

Due to the equivalence between the optimal state values and the optimal action values ($V^\star(s_t) = \max_{\alpha_t} Q^\star(s_t, \alpha_t)$) the above state value iteration algorithm can be converted to its action value form as

$$Q^{k+1}(s_t, \alpha_t) = r_{t+1}(s_t, \alpha_t) + \gamma \max_{\alpha_{t+1}} Q^k(s_{t+1}, \alpha_{t+1}), \tag{17}$$

where $Q^k(s_{t+1}, \alpha_{t+1})$ is the current update k of the future action state values (s_{t+1}, α_{t+1}) and $Q^{k+1}(s_t, \alpha_t)$ is the new update $k+1$ of the action value function for the present state action pair s_t, α_t.

After sufficient iterations over each state-action pair (s_t, α_t), choosing the best action α_{t+1} for each next state s_{t+1}, the values of the action value function will converge to the optimal values, *i.e.*, $Q^{k+1}(s_t, \alpha_t) \rightarrow Q^\star(s_t, \alpha_t)$.

At this point, the optimal policy $\pi^\star$ can be found by choosing the actions that have the highest action value and equal the higher cumulative reward, *i.e.*,
$\pi^\star = \text{argmax}_{\alpha_t} Q^\star(s_t, \alpha_t)$.

2.2 Probabilistic environment and Stochastic policies

2.2.1 Motivation

In the deterministic example described above, there are two subtle assumptions. First, we assumed that the environment does not influence directly the movement of the miner (*i.e.*, there are no *extraneous actions*[9] and that at each state the miner is moving exactly as the deterministic function of the policy $\pi(s_t)$ dictates. Moreover, each

[9]By extraneous actions we refer to actions that are outside the control of the agent and depend on the environment dynamics (see [SB18]).

state-action pair (s_t, α_t) leads to exactly one new state, s_{t+1}, and one corresponding reward $r_{t+1}(s_t, \alpha_t)$.

However, this is not always the case. In most practical applications there are *uncertainties* that influence both the environment dynamics *extraneous actions* and the movement of the agent. This means that the movement of the agent does not lead always to the intended new state and reward $s_{t+1}, r_{t+1}(s_t, \alpha_t)$.

For our miner example, we can take account of these uncertainties in the environment and the policy of the miner. More specifically, we assume that the mine is trembling (uncertainties in the dynamics of the environment) forcing the miner to miss a step and that the lamp of the miner is not working at all times (it flickers) so that the miner mixes its moves (see Figure 9).

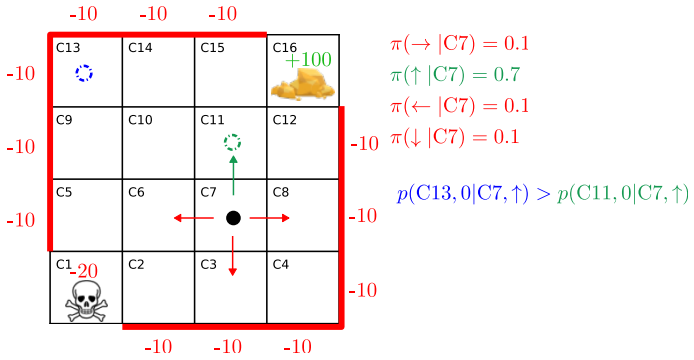


Figure 9: In the stochastic framework the move of the miner is decided based on the conditional probability of its current state. In this example the miner in state $s_t = \text{C7}$ selects more frequently the $\uparrow$ action. The next tile the miner will transition to (new state s_{t+1}), depends on the transition probabilities of the miner for its current state s_t and selected action α_t. In this example, we consider only two non-zero transition probabilities based on the state-action pair $(s_t, \alpha_t) = (\text{C7}, \uparrow)$, these are: $p(\text{C13}, 0|\text{C7}, \uparrow)$ and $p(\text{C11}, 0|\text{C7}, \uparrow)$. In both cases, the intermediate reward for the transition is zero ($r_{t+1} = 0$). We know that the environment favours the transition to $C13$ more, *i.e* it is more probable that the agent's next state will be s_{t+1} =C13 with a reward $r_{t+1} = 0$.

We implement these uncertainties in the policy of the agent and the dynamics of the environment by considering:

- The conditional probability the miner will perform an action, α_t, given the previous history of states and actions it performed in the past $s_t, \alpha_{t-1}, s_{t-1}, ..., \alpha_0, s_0$, *i.e.*, the *policy function* becomes a probability function of all previous states and actions, $\pi(\alpha_t|s_t, \alpha_{t-1}, s_{t-1}, ..., \alpha_0, s_0)$.
- The conditional probability the miner will visit a new state, s_{t+1}, and receive the reward, r_{t+1}, given the previous history of states actions and rewards

$s_t, \alpha_t, r_t, \alpha_{t-1}, s_{t-1}, ..., r_1, \alpha_0, s_0$. This is the next state and reward probability function $p(s_{t+1}, r_{t+1}|s_t, \alpha_t, r_t, \alpha_{t-1}, s_{t-1}, ..., r_1, \alpha_0, s_0)$.

The knowledge of the previous history significantly complicates the problem to be solved, but as in the deterministic case, we will consider a special type of history-independent process, called a *Markov Decision Process* [WIW89].

2.2.2 Markov Processes

A Markov process is a formal way to describe a problem that involves different states that can be observed by the agent and change sequentially[10]. At the time t during the process, the agent observes the features of a state, s_t, and then transitions to another state, s_{t+1}, receiving an immediate reward, r_{t+1}. The total reward the agent accumulates is the sum of all immediate rewards. This is called a *Markov Reward Process* (MRP).

The Dynamics of an MRP process are completely characterised by a *Markov transition matrix* $p(s_{t+1}, r_{t+1}|s_t)$ that shows the probability of the agent observing the next state, s_{t+1}, and receiving a reward, r_{t+1}, given the agent's current observation of state s_t.

In most applications when the agent lies in its current state s_t, it also chooses an action α_t that changes the probability of the agent's next state $s_{t+1},\ r_{t+1}$. This is called a *Markov Decision Process*. We note that if we assign the action, α_t, inside the state, s_t, observed by the agent, we obtain a state action pair. When we consider transitions between the different state action pairs, we retrieve again an MRP. The Markov Decision Process (MDP) is also described by a transition matrix between the agent's current state and action pair (s_t, α_t) and the next state and reward (s_{t+1}, r_{t+1}).

Considering that the process is history independent, the transition probability matrix implies that the current state, s_t, and action, α_t, contain all information needed for the transition to the next state and the next reward (this is called the *Markov property* [WIW89])[11].

Using the Markov property, the previous history doesn't matter and we can assert that

$$\pi(\alpha_t|s_t, \alpha_{t-1}, s_{t-1}, ..., \alpha_0, s_0) = \pi(\alpha_t|s_t), \tag{18}$$

$$p(s_{t+1}, r_{t+1}|s_t, \alpha_t, r_t, \alpha_{t-1}, s_{t-1}, ..., r_1, \alpha_0, s_0) = p(s_{t+1}, r_{t+1}|s_t, \alpha_t). \tag{19}$$

For the case of our miner, our analysis then simplifies significantly, because we need to only specify two functions for the DP framework to work. These are:

- The conditional probability that the miner will perform an action, α_t, given the current state, s_t, it is in, *i.e.*, the *policy function* becomes a probability function $\pi(\alpha_t|s_t)$.

[10] Sequentially means that the agent can only find itself in one state at each time. Moreover, only the past states influence the next state *i.e* in a Markov Process the future does not influence the past.

[11] In the case where the processes are history dependent and don't have the Markov property, we choose to enhance the *features* of the state by appending available information from previous states to the current state. The number of states will grow drastically but the Markov property will be restored.

- The conditional probability the miner will visit a new state, s_{t+1}, and receive the reward, r_{t+1}, given that in the current state, s_t, the miner performed an action, α_t. This is the next state and reward probability function $p(s_{t+1}, r_{t+1}|s_t, \alpha_t)$.

We present a general example of an MDP with four states $s_t \in \{\text{C1,C2,C3,C4}\}$ and three actions $\alpha_t \in \{\text{d1}{=}\leftarrow, \text{d2}{=}\uparrow, \text{d3}{=}\rightarrow\}$ in Figure 10. The transition probability matrix characterising this MDP is given in Table 3.

As we did in the deterministic case, we will focus on episodic processes. In order for episodic cases to be possible, the Markov process should contain at least one terminal state, in which the agent will remain with probability 1. In this case, the Markov process is called *absorbing*.

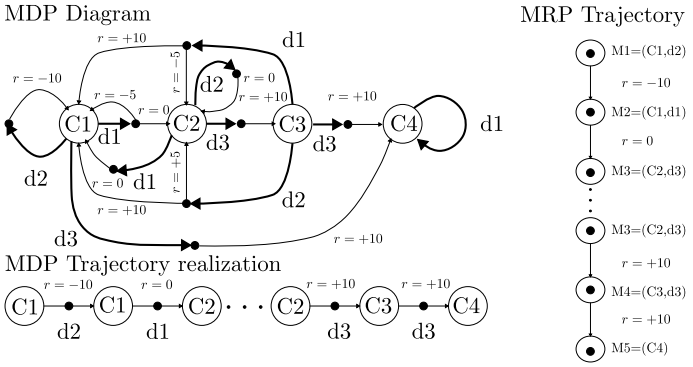


Figure 10: Left: A Markov Decision Process. At each state transition, $s_t \rightarrow s_{t+1}$, the agent performs an action, α_t, and receives a reward, r based on its final state s_{t+1}. The dynamics of the environment, *i.e.*, the rule that associates the rewards and the next states according to the current state and actions might be known (deterministic or stochastic) or unknown. We can also treat each state-action pair as a separate state. This is useful in model-free methods when the environment dynamics are not known a priori. In this case, we obtain A Markov Reward Process. For a specific trajectory realization, the state-action-reward of the MDP is shown together with its corresponding state action pair-reward MRP.

2.2.3 Value of the State and Action Value Function

In the general stochastic framework of DP, we seek to maximize the expected value of the accumulated reward starting from a given state s_t, given by

$$\begin{aligned} V_\pi(s_t) &= \mathbb{E}_\pi[G_t|s_t] \\ &= \mathbb{E}_\pi[r_{t+1}|s_t] + \gamma \mathbb{E}_\pi[G_{t+1}|s_t] \\ &= \sum_{\alpha_t \in \mathcal{A}} \pi(\alpha_t|s_t) \sum_{\substack{s_{t+1}\in\mathcal{S} \\ r_{t+1}\in\mathcal{R}}} p(s_{t+1}, r_{t+1}|s_t, \alpha_t) r_{t+1} + \gamma \mathbb{E}_\pi[G_{t+1}|s_t], \end{aligned} \tag{20}$$

$p(s_{t+1}, r_{t+1}\|s_t, \alpha_t)$	C1	C2	C3	C4
C1	[0.5,1.0,0.0]	[0.5,0.0,0.0]	[0.0,0.0,0.0]	[0.0,0.0,1.0]
C2	[1.0,0.0,0.0]	[0.0,1.0,0.0]	[0.0,0.0,1.0]	[0.0,0.0,0.0]
C3	[0.6,0.3,0.0]	[0.4,0.7,0.0]	[0.0,0.0,0.0]	[0.0,0.0,1.0]
C4	[0.0,0.0,0.0]	[0.0,0.0,0.0]	[0.0,0.0,0.0]	[1.0,0.0,0.0]

Table 3: *Markov Decision Process (MDP) Transition Matrix.* This probability matrix shows us the transition probabilities between the states in the MDP $p(s_{t+1}, r_{t+1}|s_t, \alpha_t)$ of Figure 10, i.e the probabilities of reaching a new state $s_{t+1} \in \{C1, C2, C3, C4\}$ from the current state $s_t \in \{C1, C2, C3, C4\}$ and receiving the next reward r_{t+1} given that we perform action $\alpha_t \in \{\leftarrow, \uparrow, \rightarrow\}$. Note that in each row summing over a specific action the total probability is 1.

where $r_{t+1} = r_{t+1}(s_{t+1}, \alpha_t, s_t)$, *i.e* the reward at next state s_{t+1} given the current state s_t and action α_t and $\mathbb{E}_\pi[\cdot]$ is an operator signifying the *mean* of the *expected conditional probability given the policy* π.

More precisely, the *expected value* of a discrete random variable X, $\mathbb{E}[X]$ is given by

$$\mathbb{E}[X] = \sum_{x_i \in X} x_i p(X = x_i). \tag{21}$$

where x_i are the values of the random variable and $p(X = x_i)$ are the corresponding probability of the random variable X taking the value x_i.

When the random variable is expressed in the form of conditional probabilities *e.g.* $p(X|s_t, \alpha_t)$, the average value is generalized to the *expected conditional probability*

$$g(s_t, \alpha_t) = \mathbb{E}[X|s_t, \alpha_t] = \sum_{x_i \in X} x_i p(X = x_i|s_t, \alpha_t) \tag{22}$$

. We can expand the above relation using the *general product rule* of probability theory to obtain

$$g(s_t) = \mathbb{E}[X|s_t, \alpha_t] = \sum_{x_i \in X} x_i p(X = x_i|s_t, \alpha_t)\pi(\alpha_t|s_t)p(s_t) \tag{23}$$

. Because s_t can be any state in the state space $\mathcal{S}$ we get that $p(s_t) = 1$.

We can then derive the *mean value of the expected conditional probability* as

$$\mathbb{E}_\pi[X|s_t] = \mathbb{E}[\mathbb{E}[X|s_t, \pi(\alpha_t|s_t)]] = \sum_{\alpha_t \in \mathcal{A}} \sum_{x_i \in X} x_i p(X = x_i|s_t, \alpha_t)\pi(\alpha_t|s_t). \tag{24}$$

In order to find a policy, $\pi^\star(\alpha_t|s_t)$, that maximizes the agent's expected cumulative reward, $\pi^\star = \text{argmax}_\pi \mathbb{E}_\pi[G_t|s_t]$, at the end of the episode, it is useful to evaluate the

value function of the state the agent finds itself in, $V_\pi(s_t)$, and the *value function of the available actions* the agent can take, $Q_\pi(s_t, \alpha_t)$, as we did for the deterministic case.

For a given policy, π, and starting from the state, s_t, the value function of the state can be evaluated using the Bellman equation [Bel54, BT95]

$$\begin{aligned} V_\pi(s_t) &= \sum_{\alpha_t \in \mathcal{A}} \pi(\alpha_t|s_t) \sum_{\substack{s_{t+1}\in\mathcal{S},\\ r_{t+1}\in\mathcal{R}}} p(s_{t+1}, r_{t+1}|s_t, \alpha_t) r_{t+1} + \gamma \mathbb{E}_\pi \left[\sum_{k=0}^{T-2} \gamma^k r_{t+k+2} | s_t \right] \\ &= \sum_{\alpha_t \in \mathcal{A}} \pi(\alpha_t|s_t) \sum_{\substack{s_{t+1}\in\mathcal{S},\\ r_{t+1}\in\mathcal{R}}} p(s_{t+1}, r_{t+1}|s_t, \alpha_t) r_{t+1} \\ &\quad + \gamma \sum_{\substack{s_{t+2},\dots s_T \in \mathcal{S} \\ \alpha_{t+1},\dots,\alpha_{T_1}\in\mathcal{A}, \\ r_{t+2},\dots,r_T\in\mathcal{R}}} p(Tr, s_{t+1}, \alpha_{t+1}, s_t, \alpha_t) \sum_{k=0}^{T-2} \gamma^k r_{t+k+2}, \end{aligned} \tag{25}$$

where $p(Tr, s_{t+1}, \alpha_{t+1}, s_t, \alpha_t) = p(s_T, r_T ... s_{t+2}, r_{t+2}, s_{t+1}, \alpha_{t+1}, s_t, \alpha_t)$ is the joint probability of the specific trajectory realization. Using again the *general product rule* from probability theory we get

$$p(Tr, s_{t+1}, \alpha_{t+1}, s_t, \alpha_t) = p(s_T, r_T ... s_{t+2}, r_{t+2} | s_{t+1}, \alpha_{t+1}, s_t, \alpha_t) p(s_{t+1}|s_t, \alpha_t) \pi(\alpha_t|s_t)$$

Replacing and grouping the terms we obtain,

$$\begin{aligned} V_\pi(s_t) = \sum_{\alpha_t \in \mathcal{A}} \pi(\alpha_t|s_t) \sum_{\substack{s_{t+1}\in\mathcal{S},\\ r_{t+1}\in\mathcal{R}}} p(s_{t+1}, r_{t+1}|s_t, \alpha_t) \\ \times \left[r_{t+1} + \gamma \sum_{\substack{s_{t+2},\dots s_T \in \mathcal{S} \\ \alpha_{t+1},\dots,\alpha_{T_1}\in\mathcal{A}, \\ r_{t+2},\dots,r_T\in\mathcal{R}}} p(T_r|s_{t+1}, \alpha_{t+1}, s_t, \alpha_t) \sum_{k=0}^{T-2} \gamma^k r_{t+k+2} \right]. \end{aligned} \tag{26}$$

Moreover, we note that

$$\gamma^k r_{t+k+2}|s_t, \alpha_t, r_{t+1}, s_{t+1} = \sum_{\substack{s_{t+2},\dots s_T \in \mathcal{S} \\ \alpha_{t+1},\dots,\alpha_{T_1} \in \mathcal{A}, \\ r_{t+2},\dots,r_T \in \mathcal{R}}} p(T_r|s_{t+1}, \alpha_{t+1}, s_t, \alpha_t)\gamma^k r_{t+k+2}$$

, which leads to

$$\begin{aligned} V_\pi(s_t) = &\sum_{\alpha_t \in \mathcal{A}} \pi(\alpha_t|s_t) \sum_{\substack{s_{t+1} \in \mathcal{S}, \\ r_{t+1} \in \mathcal{R}}} p(s_{t+1}, r_{t+1}|s_t, \alpha_t) \\ &\times \left[r_{t+1} + \gamma \sum_{k=0}^{T-2} \gamma^k r_{t+k+2}|s_t, \alpha_t, r_{t+1}, s_{t+1} \right], \end{aligned} \tag{27}$$

However, since the process is an MDP, the last sum (which denotes the expectation of the reward r_{t+2} for the next state) is only dependent on the next state s_{t+1}. This yields

$$\begin{aligned} V_\pi(s_t) &= \sum_{\alpha_t \in \mathcal{A}} \pi(\alpha_t|s_t) \sum_{\substack{s_{t+1} \in \mathcal{S} \\ r_{t+1} \in \mathcal{R}}} \left[r_{t+1}(s_t, \alpha_t, s_{t+1}) + \gamma \sum_{k=0}^{T-k-2} \gamma^k r_{t+k+2}|, s_{t+1} \right] \\ &= \sum_{\alpha_t \in \mathcal{A}} \pi(\alpha_t|s_t) \sum_{\substack{s_{t+1} \in \mathcal{S} \\ r_{t+1} \in \mathcal{R}}} \left[r_{t+1}(s_t, \alpha_t, s_{t+1}) + \gamma V_\pi(s_{t+1}) \right]. \end{aligned} \tag{28}$$

Similarly for the action value function $Q_\pi(s_t, \alpha_t)$ we derive

$$\begin{aligned} Q_\pi(s_t, \alpha_t) &= \mathbb{E}_\pi[G_t|s_t, \alpha_t] \\ &= \sum_{\substack{s_{t+1} \in \mathcal{S} \\ r_{t+1} \in \mathcal{R}}} p(s_{t+1}, r_{t+1}|s_t, \alpha_t) \\ &\quad \times \left[r_{t+1} + \gamma \sum_{\alpha_{t+1} \in \mathcal{A}} \pi(\alpha_{t+1}|s_{t+1}) \sum_{k=0}^{T-k-2} \gamma^k r_{t+k+2}|s_t, \alpha_t, r_{t+1}, s_{t+1}, \alpha_{t+1} \right] \\ &= \sum_{\substack{s_{t+1} \in \mathcal{S} \\ r_{t+1} \in \mathcal{R}}} p(s_{t+1}, r_{t+1}|s_t, \alpha_t) \left[r_{t+1} + \gamma \sum_{\alpha_{t+1} \in \mathcal{A}} \pi(\alpha_{t+1}|s_{t+1}) Q_\pi(s_{t+1}, \alpha_{t+1}) \right]. \end{aligned} \tag{29}$$

The relationship between $V_\pi(s_t),\ Q_\pi(s_t, \alpha_t)$ can be shown starting from equation 27 and replacing the initial definition of the action-value function 29, obtaining

$$V_\pi(s_t) = \sum_{\alpha_t \in \mathcal{A}} \pi(\alpha_t|s_t) Q_\pi(s_t, \alpha_t). \tag{30}$$

The above mathematical derivation for the stochastic Bellman equation can be summarized in Figure 11.

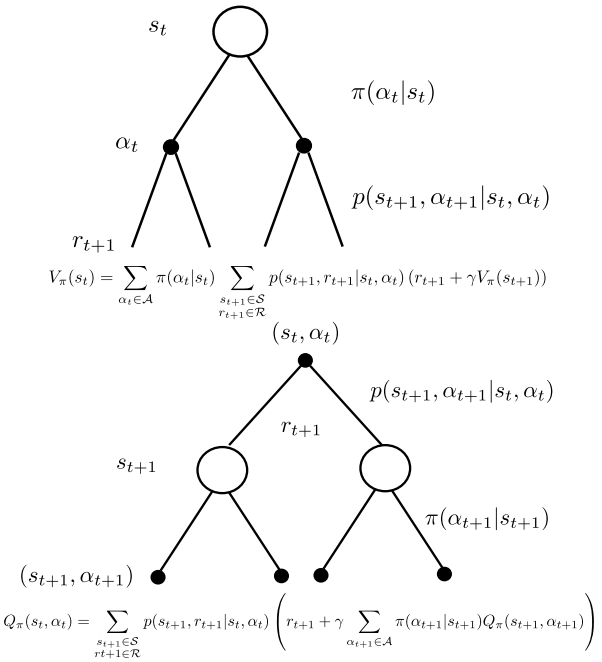


Figure 11: The schematic diagrams above indicate how the state value function, $V_\pi(s_t)$, and the action value function, $Q_\pi(s_t, \alpha_t)$, are evaluated. This is done through the definition of the average value $V(s_t) = \mathbb{E}_\pi[G_t|s = s_t]$, $Q(s_t, \alpha_t) = \mathbb{E}_\pi[G_t|s = s_t, \alpha = \alpha_t]$, the dynamics of the environment $p(s_{t+1}, r_{t+1}|s = s_t, \alpha = \alpha_t)$, and the policy of the agent $\pi(\alpha_t|s_t)$.

2.2.4 Bellman's Optimality conditions

The logic we presented in the deterministic case is also valid for the stochastic case considering the maximization of the expected value of the accumulated reward $\max_\pi \mathbb{E}_\pi[G_t]$. More specifically the Bellman optimality conditions and the policy improvement theorem also hold in the stochastic case [SB18]. The general formulae for the optimal state value function, $V^\star(s_t)$, and action-value function, $Q^\star(s_t, \alpha_t)$, are given by

$$V^\star(s_t) = \max_{\alpha_t} \sum_{\substack{s_{t+1}\in\mathcal{S} \\ r_{t+1}\in\mathcal{R}}} (s_{t+1}, r_{t+1}|s_t, \alpha_t) \left[r_{t+1} + \gamma V_{\pi^\star}(s_{t+1}) \right], \tag{31}$$

$$Q^\star(s_t, \alpha_t) = \sum_{\substack{s_{t+1}\in\mathcal{S} \\ r_{t+1}\in\mathcal{R}}} p(s_{t+1}, r_{t+1}|s_t, \alpha_t) r_{t+1} + \gamma \max_{\alpha_{t+1}} Q(s_{t+1}, \alpha_{t+1}), \tag{32}$$

where $r_{t+1} = r_{t+1}(s_t, \alpha_t, s_{t+1})$.

2.2.5 Policy Improvement Theorem

For the general stochastic case, the *Policy Improvement Theorem* [SB18] reads that any policy π' for which
$V_\pi(s_t) \leq \sum_{\alpha_t \in \mathcal{A}} \pi(s_t, \alpha_t) Q(s_t, \pi'(\alpha_t|s_t))$ is a policy that we can improve on.

We note that in the above theorem the action value, $Q_\pi(s_t, \pi'(\alpha_t|s_t))$, belongs to the set of action values evaluated with the initial policy, π, at each state, $s_t \in \mathcal{S}$, but it is selected based on policy $\pi' \alpha_t, s_t$. The above relation is equivalent to $V_\pi(s_t) \leq V'_\pi(s_t)$.

The Bellman equations applied to the framework of MDPs, lend themselves to a straightforward application of *Dynamic Programming*, which provides basic algorithms to maximize the state (Value iteration) and/or action values (Q-Value iteration) of the environment and consequently find the optimal policy for maximizing the expected reward of the agent.

Moreover, the notions of MDPs and the Bellman Optimality conditions play a central role in *Reinforcement Learning* (RL). In particular, they constitute the basis for many learning methods that seek to maximize the agent's expected reward.

Based on whether or not we have complete knowledge of the dynamics of the environment, $p(s_{t+1}, r_{t+1}|s_t, \alpha_t)$, we separate these learning methods into two categories namely model-based and model-free methods.

2.2.6 Model-based Methods

In the case of model-based methods we can find the optimal policy using a *Policy Iteration* algorithm or a *Generalised policy iteration algorithm* (see Figure 7). These methods are the equivalent of the deterministic model-based methods discussed in section 2.1.4. Consequently, they work well in ideal cases where the number of available states and actions are small and the MDP probability distribution function $p(s_{t+1}, r_{t+1}|s_t, \alpha_t)$ is known a priori. In this case the agent *plans* its actions before they are taken.

Policy Iteration (PI) method

Similar to what we explained in the deterministic case, In this algorithm, we achieve the optimal action values and the respective optimal policy by modifying the policy π.

At first, we evaluate the policy π. This evaluation can be done by direct inverting the Markov system. However, this takes a lot of time when the number of states increases. We can evaluate the solution of the system faster by using an iterative solution strategy.

In particular, using a *Fixed Point Iteration algorithm* we are able starting from a random initialization of $V^0(s_t), Q^0(s_t, \alpha_t)$ (which does not respect necessarily the Bellman equations (28),(29)) and applying iteratively the state value definition of equation (28), for a randomly generated policy π, to evaluate the values of the state value function $V_\pi^{k+1}(s)$ at each state of the environment.

Once the values of the state value function $V_\pi(s_t)$ (respectively state value function, $Q_\pi(s_t, \alpha_t)$, based on policy π, have converged for each state s_t (and action α_t), we retrieve the action α_t' that is responsible for the maximum value of the action value function i.e $\alpha_t' = \text{argmax}_{\alpha_t} Q(s_t, \alpha_t)$.

In the case of deterministic policies, if this action α_t' is different than the action calculated by the policy $\alpha_t = \pi(\alpha_t)$, *i.e.*, $\alpha_t' \neq \pi(s_t)$, we update the action of policy algorithm at the specific state s_t equal to α_t'.

Moreover, in the stochastic framework, we can also consider an $\varepsilon - greedy\ policy$. This policy can be applied according to the following rule

$$\pi^{k+1}(\alpha_t|s_t) = \begin{cases} 1 - \varepsilon - \frac{\varepsilon}{|\mathcal{A}|}, & \alpha_t = \alpha_t^\star \\ \frac{\varepsilon}{|\mathcal{A}|} & \alpha_t \neq \alpha_t^\star, \end{cases}$$
$$\alpha_t^\star = \text{argmax}_{\alpha_t} Q_\pi^{k+1}(s_t, \alpha_t), \tag{33}$$

where ε is a small parameter and $\mathcal{A}$ is the action space of the system.

Essentially, this policy update rule indicates that the action that had the highest action value will be sampled among all possible other actions with probability $1 - \varepsilon$. In contrast, the rest of the actions will be sampled with a probability of ϵ. This policy may only partially exploits the best action according to a specific state in the MDP, *i.e.*, it is not a purely greedy policy.

However, it has been shown to converge towards the optimal value and as a result to an optimal policy (see [SB18]). This stochastic policy is very useful in the case of model-free learning, where the agent does not know the transition probabilities of the MDP.

Once the policy π' has been updated, we evaluate the new updates of the state value and action-value functions $V_{\pi'}^{k+1}(s_t), Q_{\pi'}^{k+1}(s_t, \alpha_t)$ as

$$V_{\pi'}^{k+1}(s_t) = \sum_{\alpha_t \in \mathcal{A}} \pi'(\alpha_t|s_t) \sum_{\substack{s_{t+1} \in \mathcal{S} \\ r_{t+1} \in \mathcal{R}}} \left[r_{t+1} + \gamma V_{\pi'}^k(s_{t+1}) \right] \tag{34}$$

$$Q_{\pi'}^{k+1}(s_t, \alpha) = \sum_{\substack{s_{t+1} \in \mathcal{S} \\ r_{t+1} \in \mathcal{R}}} p(s_{t+1}, r_{t+1}|s_t, \alpha_t) \times \left(r_{t+1} + \gamma \sum_{\alpha_{t+1} \in \mathcal{A}} \pi'(\alpha_{t+1}|s_{t+1}) Q_{\pi'}^k(s_{t+1}, \alpha_{t+1}) \right). \tag{35}$$

When the values of the state-value function under policy $\pi'(\alpha_t|s_t)$ have converged in all states, we say that the PI algorithm has converged to an optimal policy $\pi^\star(s_t, \alpha_t) = \pi'(s_t, \alpha_t)$ and yields the optimal values $V^\star(s_t) = V_{\pi^\star}(s_t),\ Q^\star(s_t, \alpha_t) = Q_{\pi^\star}(s_t, \alpha_t)$.

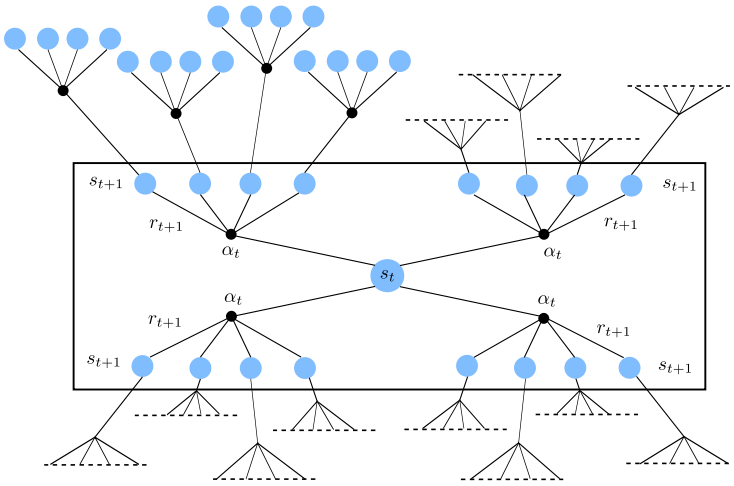


Figure 12: Stencil of the Policy iteration algorithm of Dynamic Programming, we present the case where the algorithm evaluates the values of the state value function $V^{\star}(s_t)$. The states and actions inside the rectangle are taken into account during the update of the value of the state function.

We present in Figure 12 the stencil for the model-based dynamic programming algorithm. We note that the updates involve only the current state-action pair, s_t, α_t, and the next state s_{t+1}.

We emphasize that the optimal state value function, $V^{\star}_{\pi}(s)$, and the optimal action value function, $Q^{\star}_{\pi}(s, \alpha)$, values are unique. However, the policies that yield the optimal reward are not. In this algorithm, the evaluation of the future state-action value functions $(V^{k}_{\pi'}(s_{t+1}), Q^{k}_{\pi'}(s_{t+1}, \alpha_{t+1}))$ does not happen according to the initial policy π, but the new updated policy $\pi'(\alpha_t|s_t)$ is used directly (see also Figure 7).

Generalised Policy Iteration methods (GPI)

Similar to what we discussed in the deterministic case, we can also use directly Bellman's optimality condition to perform the updates on the values of the state value and action value functions without performing a policy update. These learning algorithms are called GPI methods.

In these methods, we apply directly the nonlinear $\max$ operator for the Bellman optimality condition. As a result, the system that emerges is not linear and cannot be solved by a direct inversion method. We are only left with the iterative approach.

- **Value Iteration method**

The simplest method to use in order to find the optimal $V^{\star}(s_t)$ and the corresponding optimal policy $\pi^{\star}$, is to iterate over each state, s_t, using the $\max_{\alpha_t}$ operator, and

update the value of the state, $V^k(s_t)$, according to the rule

$$V^{k+1}(s_t) = \max_{\alpha_t} \sum_{\substack{s_{t+1}\in\mathcal{S}\\ r_{t+1}\in\mathcal{R}}} p(s_{t+1}, r_{t+1}|s_t, \alpha_t) \left[r_{t+1} + \gamma V^k(s_{t+1})\right], \tag{36}$$

where $V^k(s_t), V^k(s_{t+1})$ are the current and subsequent state values at iteration k and $V^{k+1}(s_t)$ is the new state value for the next iteration. We emphasize that these values are not evaluated based on a policy $\pi(\alpha_t|s_t)$ but by direct application of the maximum $\max_{\alpha_t}$ operator at each state. After sufficient iterations over each state choosing the best action, α_t, at each state, the values of the state value function will converge to the optimal values, $V^\star(s_t)$. At this point, the optimal policy, $\pi^\star$, can be found by choosing to move along the path of optimal state values and therefore to higher cumulative reward, *i.e.*, $\pi^\star = \text{argmax}_{\alpha_t} \sum_{\substack{s_{t+1}\in\mathcal{S}\\ r_{t+1}\in\mathcal{R}}} p(s_{t+1}, r_{t+1}|s_t, \alpha_t) \left[r_{t+1} + \gamma V^\star(s_{t+1})\right]$.

- **Q-Value Iteration$\diagdown$Greedy-Policy Improvement algorithm**

Because of the equivalence between the optimal state values and the optimal action values ($V^\star(s_t) = \max_\alpha Q^\star(s_t, \alpha)$) the above state value iteration algorithm can be converted to its action value form

$$Q^{k+1}(s_t, \alpha_t) = \sum_{\substack{s_{t+1}\in\mathcal{S}\\ r_{t+1}\in\mathcal{R}}} p(s_{t+1}, r_{t+1}|s_t, \alpha_t) \left(r_{t+1} + \gamma \max_{\alpha_{t+1}} Q^k(s_{t+1}, \alpha_{t+1})\right). \tag{37}$$

where $Q^k(s_{t+1}, \alpha_{t+1})$ are the future action state values at iteration k and $Q^{k+1}(s_t, \alpha_t)$ is the new update of the state action value function. At this point, the optimal policy, $\pi^\star$, can be found by choosing the actions that have the highest action value and equal the higher cumulative reward, *i.e.*, $\pi^\star = \text{argmax}_{\alpha_t} Q^\star(s_t, \alpha_t)$.

2.2.7 Model-free methods

These methods are used when the MDP probability distribution function, $p(s_{t+1}, r_{t+1}|s_t, \alpha_t)$, is not known. This is the case in most applications of practical geomechanical interest, where the number of states and actions and thus the dimensions of the *Markov transition matrix* can become prohibitively large for planning strategies to be practical.

Therefore, model-free algorithms that *learn* action to return associations are more useful. These methods can again be cast into the framework of value iteration methods including Monte Carlo Control, Temporal differences learning (SARSA), Q-learning, Deep Q Networks, Policy gradient methods, and Actor-Critic Networks, to name a few (see [BT95, SB18, Dee22]). Especially in the case of *Deep Learning*, different methods and networks can be combined together to yield better optimal policies and higher cumulative rewards (see also [Gér22, Fu16]).

The general idea of the model-free methods is that the agent does not know the probability function of the MDP, $p(s_{t+1}, r_{t+1|s_t,\alpha_t})$, that connects future states, s_{t+1}, and rewards, r_{t+1}, to current actions, α_t, and states s_t. For this reason, the action value of the state, s_t, according to the second Bellman equation (29) cannot be computed directly. Instead, the action value $Q_\pi(s_t, \alpha)$ at state s_t for any action, α, selected by the policy, π, will have to be *evaluated* through averaging the cumulative rewards of multiple episodes for different states of the MDP. This stage of learning is called *prediction*.

After a step of prediction, the new estimates of the state and action values, $Q_\pi(s_t, \alpha)$, are available. Then the algorithm will adjust its policy to π' in order to choose with higher probability the actions with the highest action value at each state $Q_{\pi'}(s_t, \alpha_t)$.

Monte Carlo method

Let us consider that the agent finds itself in an environment with a small space of available states, $s_t \in S$, and that the agent takes actions, $Q_\pi(s_t, \alpha_t)$, based on some random policy, π, The probability function of the MDP, $p(s_{t+1}, r_{t+1|s_t,\alpha})$, is unknown. For this reason the value function of a state, s_t, and the action value function of a state-action pair (*i.e.*, (s_t, α_t), $V_\pi(s_t), Q_\pi(s_t, \alpha_t)$) cannot be computed directly.

In the Monte Carlo (MC) method we choose to *estimate* the action value, $Q_\pi(s_t, \alpha_t)$, at state s_t for any action, α_t, selected by the policy, $\pi(\alpha_t|s_t)$, by using equation (29), *i.e.*, by running multiple episodes of numerical experiments, generating a sufficient number of trajectories and finally averaging over the expected reward for each state-action pair[12].

We do this procedure in batches and in each batch, m episodes are run. In order to average the expected reward at each state-action pair, we take note of the agent's trajectory in each episode of the batch. Then for each state-action pair, (s_t, α_t), in the agent's trajectory, we calculate the accumulated reward (see Figures 13 and 14).

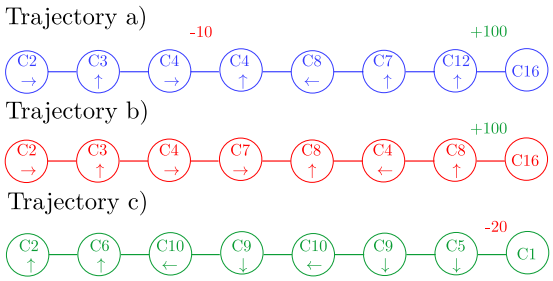
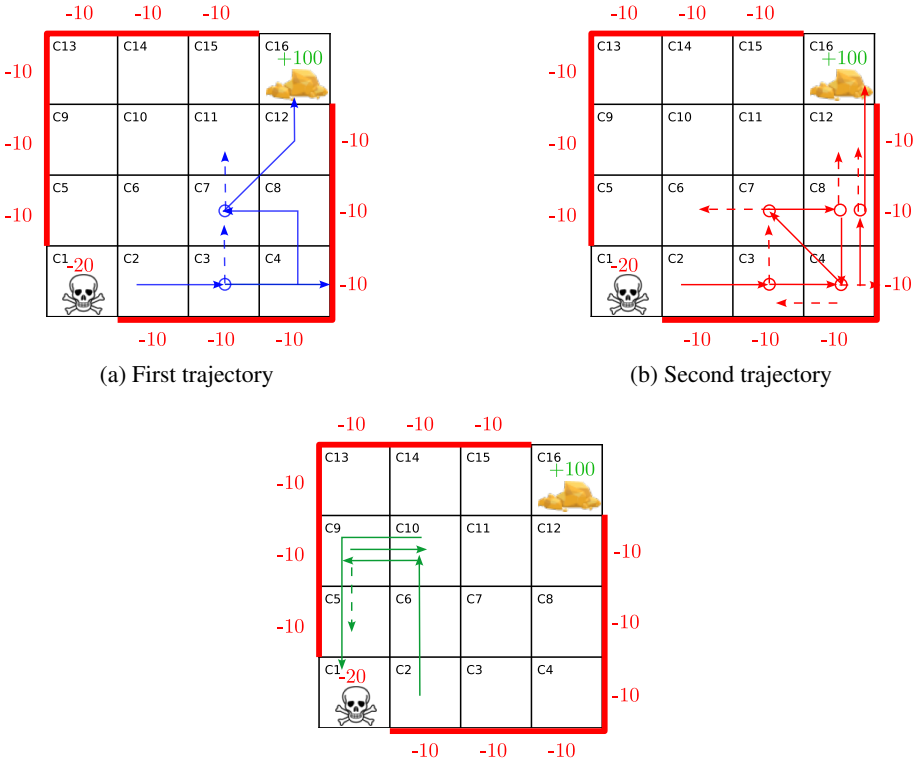


Figure 13: Three realizations of the miner's trajectory inside the mine

[12]This way we render the *Markov Decision Process* into a *Markov Reward Process*, where the agent visits different state-action pairs during an episode and then, its receives a reward r_{t+1}. Thus, each state-action pair is treated as a separate state leading to a different reward.



(a) First trajectory

(b) Second trajectory

(c) Third trajectory

Figure 14: Accumulated reward counting in multi-pass Monte Carlo. We examine the realization of the agent's trajectory of Figure 13. In a *Markov Reward Process* the states are the state-action pairs of the corresponding *Markov Decision Process*. Every time the state s_t is visited, the reward after each state is summed. Here states $(C8, \uparrow)$ and $(C9, \uparrow)$ were visited twice in the same episode during trajectories b, and c, respectively. And state $(C2, \rightarrow), (C3, \uparrow), (C4, \rightarrow)$ visited twice in different episodes. The trajectories have a degree of randomness due to the stochastic policy $\pi(\alpha_t|s_t)$ and the environment dynamics. The points where the action selected by the agent was not followed through are shown with a circle and the intended action is shown with a dotted line.

For each state-action pair, we add the accumulated reward over the different episodes of the batch and then we divide by the number of times the state-action pair was present in the batch. The action value then is given by

$$Q_\pi^{k+1}(s_t, \alpha_t) = \mathbb{E}_\pi[G_t|s_t, \alpha_t] = \frac{1}{C((s_t, \alpha_t))} \sum_{m=1}^{M} \sum_{t=0}^{T_m - 1} \mathbb{I}(s = s_t, \alpha_t) G_{t_m}, \quad (38)$$

where G_{t_m} is the accumulated reward at the end of the episode m starting counting at state-action pair (s_t, α_t), $C((s_t, \alpha_t))$ is the count of each state-action pair inside the batch of trajectories M and $\mathbb{I}(s = s_t, \alpha_t)$ is the assignment matrix that selects only the accumulated returns of the selected state-action pair (s_t, α_t) (see Table 4).

	$G, T_{r,a}$	$G, T_{r,b}$	$G, T_{r,c}$	$C((s_t, \alpha_t))$	$\mathbb{E}[G]$
(C2, $\rightarrow$)	+90	+100	0.0	2	+95
(C2, $\uparrow$)	0.0	0.0	-20	1	-20
(C3, $\uparrow$)	+90	+100	0	2	+95
(C4, $\rightarrow$)	+90	+100	0	2	+95
(C4, $\uparrow$)	+100	0.0	0.0	1	+100
(C4, $\leftarrow$)	+100	0.0	0.0	1	+100
(C5, $\downarrow$)	0.0	0.0	-20	1	-20
(C6, $\uparrow$)	0.0	0.0	-20	1	-20
(C7, $\rightarrow$)	0.0	+100	0.0	1	+100
(C7, $\uparrow$)	+100	0.0	0.0	1	+100
(C8, $\uparrow$)	0.0	+200	0.0	2	+100
(C8, $\leftarrow$)	+100	0.0	0.0	1	+100
(C9, $\downarrow$)	0.0	0.0	-40	2	-20
(C10, $\downarrow$)	0.0	0.0	-20	2	-20
(C12, $\uparrow$)	+100	0.0	0.0	1	+100

Table 4: Value of a state-action pair (Q-values) for three different trajectory realizations ($\gamma = 1.0$). $C((s_t, \alpha_t))$ is the number of times (count) the specific action-value pair was encountered in the batch of trajectories (in this example the batch size is 3). The expected value is given by summing the rewards following the specific state-action pair over the count of the state-action pair in the batch of trajectories.

Instead of evaluating the new average after each batch of trajectories, we could evaluate the running average of $Q_\pi(s_t, \alpha)$ after the agent realizes one trajectory. This reduces significantly the memory storage requirements of the algorithm. The above relation (38) can then be written as

$$Q_\pi^{m+1}(s_t, \alpha_t) = Q_\pi^m(s_t, \alpha_t) + \frac{1}{C((s_t, \alpha_t))} \left(G_{t_m} - Q_\pi^m(s_t, \alpha)\right). \tag{39}$$

The update parameter $\frac{1}{C((s_t, \alpha_t))}$ that based on the number of counts of the pair (s_t, α_t) on the batch samples, can be also replaced by a constant step-size learning hyperparameter a[13] and we obtain

$$Q_\pi^{m+1}(s_t, \alpha_t) = Q_\pi^m(s_t, \alpha_t) + a \left(G_t^m - Q_\pi^m(s_t, \alpha_t)\right). \tag{40}$$

We show in Figure 15 the stencil of the Monte-Carlo algorithm, during the policy evaluation.

[13]The action evaluation (prediction) method then is called "constant-a MC".

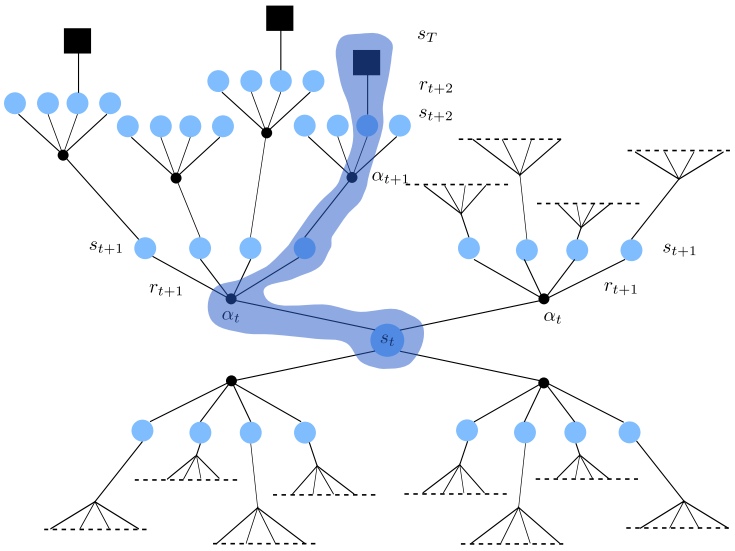


Figure 15: Stencil of the Monte-Carlo Learning algorithm. The black rectangles indicate terminal states. We note that a whole episode needs to be computed before any update can take place.

We note that the effect of the step-size parameter, a, in the convergence towards the optimal $\max_{\alpha_t} Q_\pi(s_t, \alpha_t)$ is similar to the effect of the gradient update parameter in the stochastic batch gradient descend algorithm used in supervised learning tasks. In essence, for large values of the hyperparameter a, the optimal action values oscillate without really converging to a value. *i.e.*, quick but noisy learning. On the contrary, smaller values of a will converge at the optimal value slowly but in an accurate manner (see Figure 16).

In order to perform the control task in RL, *i.e.*, reach the optimal values of the action value function $Q^\star(s_t, \alpha_t)$ at each state-action pair, we can use a default greedy policy that immediately selects the best action based on the updated values of the action value function.

Notice that the use of such greedy policy implies that another policy π has been used during the prediction of the action value function in each state-action pair (s_t, α_t). This type of learning where another policy generates the data and makes predictions and another policy controls the maximization of the action value function towards the optimal action values, $Q^\star(s_t, \alpha_t)$, is called *Off-policy learning* and requires the use of statistical techniques, like the so-called *Importance sampling*, in order to remove any bias from the learned data.

This second problem comes from the fact that after the first values of $Q_\pi(s_t, \alpha_t)$ have been evaluated, the greedy policy will always exploit the same actions that yield the

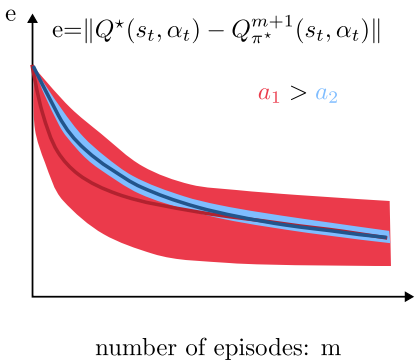


Figure 16: Representation of the convergence error estimate between updates of the optimal action value function $Q^{\star}(s_t,\alpha_t) - Q^{m+1}_{\pi^{\star}}(s_t,\alpha_t)$ for all state-action pairs (s_t,α_t). Red curve: higher values of the step hyperparameter a_1 make faster the convergence but higher the variance. Blue curve: lower values of the learning hyperparameter a_2 lead to slower convergence but less variance.

highest $Q_{\pi}(s_t,\alpha_t)$ at each s_t in order to maximise the expected accumulated reward[14]. Because $Q_{\pi}(s_t,\alpha_t)$ is not known for all state-action pairs, the greedy policy runs the risk of getting trapped in a local maximum of the expected reward and not reaching the global maximum. This problem is also known as the *Exploration vs Exploitation tradeoff*.

Alternatively, the agent can use an ε-greedy policy as in the case of the Q-value model-based method for the prediction and the action value function and the optimization of the reward (see equation (33)).

In particular, this policy considers that all possible actions are included during the action selection stage, *i.e.*, $\pi(\alpha|s) > 0, \forall s \in \mathcal{S},\ \forall\alpha \in \mathcal{A}(f)$[15]. Therefore, given enough tries by the agent, it is guaranteed to visit all possible states of the problem, spending more time on the interesting parts of the problem where higher expected cumulative rewards are present.

Thus, the Monte Carlo algorithm *learns to associate* actions to expected accumulated rewards achieving a balance between exploration and exploitation of the problem state space.

We note that for a larger number of states, the Monte Carlo Control method becomes impractical. This is because the action values are updated at the end of each episode which can take a long time to complete.

[14]Considering the example of the mine, a greedy miner won't explore the mine and will head straight for the exit, where the principal reward is, without ever searching for any hidden or secondary rewards that would increase the final cumulative reward.

[15]In this case the policy is characterized as *soft*.

On policy vs Off policy methods

In RL we make a distinction between the policy the agent uses in order to generate the data (behaviour policy $b(s_t|\alpha_t)$) and the policy we improve/evaluate in order to find the optimal policy (target policy $\pi(s_t|\alpha_t)$)) and maximize the expected cumulative reward. When we use the target policy to also generate the data in subsequent episodes, the RL method is called an *On policy* method, else, the method is called *Off policy*.

The need for *Off policy* methods becomes clear when we don't want to compromise the exploratory capabilities of the agent and more importantly when we want to expand our dataset with trajectory samples from third parties. This is common because RL applications take quite a lot to run and produce an adequate number of new trajectories for statistics applications. Therefore, the use of already available data is always welcome.

We can convert the action values of the initial dataset produced under the policy $b(\alpha_t|s_t)$, to action values obtained through the new policy $\pi(\alpha_t|s_t)$, by using *Importance sampling* provided that in each policy the available actions over each state are common, *i.e.*, if $\pi(s|\alpha) > 0$ then $b(s|\alpha) > 0$. The action values under policy π can be found using the formula

$$Q_\pi(s_t, \alpha_t) = \mathbb{E}_b[\rho G_t | s_t, \alpha_t], \tag{41}$$

where:

$$\rho = \Pi_{\tau=t+1}^{T-1} \frac{b(\alpha_\tau|s_\tau)}{\pi(\alpha_\tau|s_\tau)}. \tag{42}$$

Temporal Difference learning: SARSA

When Monte Carlo is used, the update to the new action values happens only after the agent has completed one episode. When the number of states becomes large, an episode may take a long time to complete and therefore, convergence to the maximum action values, $Q_{\pi^\star}(s_t, \alpha_t)$, and optimal policy, $\pi^\star(s_t, \alpha_t)$, becomes slow. Temporal Difference algorithm (TD) or State-Action-Reward-State-Action algorithm (SARSA), solves the update problem of the Monte Carlo control algorithm by updating the action values during the episode. The update happens every n-steps inside the episode m using the estimate of the expected accumulated reward. This is done assuming the agent acts optimally after a future state that is n-steps in front of the current state s_{t_m}. We consider as an optimal policy the ε-greedy policy (see equation (33)). We call this estimate a target value $G_{t_m}(n)$[16]. Inside an episode m, the update of the action value $Q_\pi^{k+1}(s_{t_m}, \alpha_{t_m})$ is given by

$$Q_\pi^{k+1}(s_{t_m}, \alpha_{t_m}) = Q_\pi^k(s_{t_m}, \alpha_{t_m}) + a\left(G_{t_m}(n) - Q_\pi^k(s_{t_m}, \alpha_{t_m})\right), \tag{43}$$

[16] A technique that uses estimates of state and/or action values in order to evaluate some target values is called *Bootstraping*.

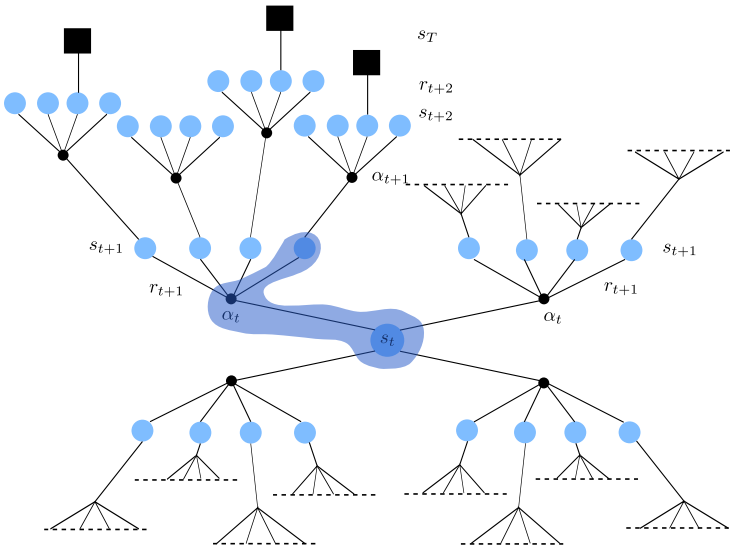


Figure 17: Stencil of the Temporal Difference Learning algorithm for 1 number of steps into the future (SARSA). The black rectangles indicate terminal states. The algorithm needs only the next state in order to perform the update of the current action value.

where:

$$G_{t_m}(n) = r_{t_m+1} + \gamma r_{t_m+2} + ...\gamma^{n-1} r_{t_m+n} + \gamma^n Q_\pi(s_{t_m+n}, \alpha_{t_m+n}). \tag{44}$$

We show in Figure 17 the stencil of the SARSA algorithm, for policy evaluation and policy iteration tasks.

We note that in equation (44) the update of the action value, $Q_\pi^{k+1}(s_{t_m}, \alpha_{t_m})$, does not coincide with the next episode $m+1$. The update happens every n steps inside the episode depending on the n-step evaluation of the target value $G_{t_m}(n)$. We also note two special cases in the TD method depending on the hyper-parameter n:

- $n = 1$: In this case, the action values are updated after the new state is reached. This allows for maximum learning during an episode.
- $n \to \infty$: In this case, the action values are updated only after the episode ends. This defaults to the usual Monte Carlo learning method of the previous section.

In the TD method, learning is affected by the pair of hyperparameters n and a (see Figure 18). In particular, we note that a low parameter n (*e.g.*, $n = 1$) yields faster learning rates when the higher values of the step size parameter a are considered. Moreover, with respect to the MC method, the convergence results are less noisy. This indicates a fundamental difference between the MC and TD learning methods.

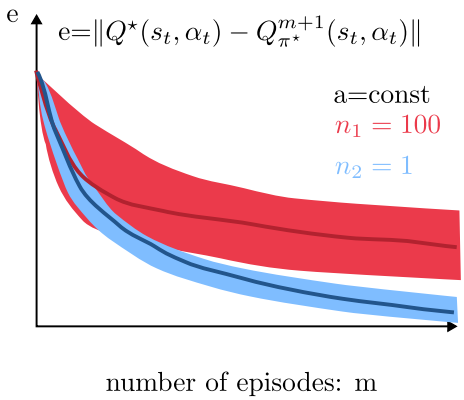


Figure 18: Comparison of the temporal difference algorithm for different values of the hyperparameter n based on their convergence estimates between updates of the action value function $Q^{\star}(s_t,\alpha_t) - Q^{m+1}(s_t,\alpha_t)$ for all s_t,α_t. Red curve $n_1 = 100$: The algorithm is equivalent to a Monte Carlo method. Blue curve $n_2 = 1$: The algorithm reaches a lower minimum for the convergence error. The two methods do not converge to the same optimal point because of the different estimates for the expected reward $\mathbb{E}[G_t]$.

Let us consider that we train the MC and TD algorithms on a batch of m episodes until convergence to the optimal state, $V^{\star}(s_t)$, and action values, $Q^{\star}(s_t,\alpha_t)$, for this batch. This way we eliminate the influence of the step size a. According to [SB18], the two methods will converge to different estimates of $Q^{\star}(s_t,\alpha_t)$! In particular, the MC method will converge towards values of the state $V_{\pi' \text{MC}}(s)$ that minimize the mean square error between the values of the state and the limit, while the TD method will tend towards the estimate that maximizes the likelihood of high expected reward. Therefore, TD is a better method for approximating the maximum expected accumulated reward $\max \mathbb{E}[G_t]$ when the assumptions of an MDP hold.

Q-learning

In the TD case, we used the estimate of a $\varepsilon - greedy$ policy for the accumulated reward, at step n inside the episode m. Another approach is to update the accumulated reward estimate using the maximum action value function of the n-step action value pair (s_{t+n},α_{t+n}). This leads to the following algorithm for learning the optimal action value function

$$Q^{k+1}_{\pi}(s_{t_m},\alpha_{t_m}) = Q^{k}_{\pi}(s_{t_m},\alpha_{t_m}) + a\left(G_{t_m}(n) - Q^{k}(s_{t_m},\alpha_{t_m})\right), \quad (45)$$

where:

$$G_{t_m}(n) = r_{t_m+1} + \gamma r_{t_m+2} + ...\gamma^{n-1} r_{t_m+n} + \gamma^{n} \max_{\alpha_{t_m+n}} Q^{k}(s_{t_m+n},\alpha_{t_m+n}). \quad (46)$$

We note that Q-learning is an off-policy method because the $\max_{\alpha_t} Q(s_t, \alpha_t)$ is not known a priori but deduced from the evaluation of the $Q_\pi(s_t, \alpha_t)$ values after n-steps in the trajectory.

Expected SARSA

Another variant of the Q-learning method can be made using the expected value of the action under policy $\pi(\alpha_t|s_t)$, *i.e.*,

$$Q_\pi^{k+1}(s_{t_m}, \alpha_{t_m}) = Q_\pi^k(s_{t_m}, \alpha_{t_m}) + a\left(G_{t_m}(n) - Q_\pi^k(s_{t_m}, \alpha_{t_m})\right), \tag{47}$$

where:

$$\begin{aligned} G_{t_m}(n) = r_{t_m+1} + \gamma r_{t_m+2} + ...\gamma^{n-1} r_{t_m+n} \\ + \gamma^n \sum_{\alpha_{t_m+n} \in \mathcal{A}} \pi(\alpha|s_{t_m+n}) Q^k(s_{t_m+n}, \alpha_{t_m+n}). \end{aligned} \tag{48}$$

We note that this can be an on-policy method because the action value $Q_\pi^k(s_t, \alpha_t)$ is evaluated the same way as its update $Q_\pi^{k+1}(s_t, \alpha_t)$.

2.3 Function approximation

When the state space is large (*e.g.*, a continuous system with infinite states) it is not practical -let alone feasible- to perform iterations over each state to find the optimal state and action values. We can move around this difficulty considering that the optimal state value function, $V_{\pi^\star}(s_t)$, and the optimal action value function, $Q_{\pi^\star}(s_t, \alpha_t)$, can be approximated by other functions $\hat{v}(s_t, \boldsymbol{w}) = \boldsymbol{w}\boldsymbol{x}$ and $\hat{q}(s_t, \alpha_t, \boldsymbol{w}) = \boldsymbol{w}\boldsymbol{\psi}$, respectively. Such functions are a linear combination of the vector of weight parameters, $\boldsymbol{w}$, and the vectors of data features $\boldsymbol{x}, \boldsymbol{\psi}$. By data features, we mean the data itself or patterns from the data that can arise from any classification method (see **Chapter 4: Classification Techniques in Machine Learning** for more details). Any number of weights and data features can be implemented and the task of state evaluation now is to minimize, $\overline{VE}$ (respectively $\overline{QE}$), which can be the square error (or the Huber norm) between the target function $V_{\pi^\star}(s_t)$ and its approximation $\hat{v}(s_t, \boldsymbol{w})$ (respectively, $Q_{\pi^\star}(s_t, \alpha_t)$ and its approximation $\hat{q}(s_t, \alpha_t, \boldsymbol{w})$) as

$$\overline{VE} = \sum_{s \in S} \mu(s) \left[V_{\pi^\star}(s) - \hat{v}_\pi(s, \boldsymbol{w})\right]^2, \tag{49}$$

$$\overline{QE} = \sum_{s \in S, \alpha \in \mathcal{A}} \mu(s, \alpha) \left[Q_{\pi^\star}(s_t, \alpha_t) - \hat{q}_\pi(s_t, \alpha_t, \boldsymbol{w})\right]^2, \tag{50}$$

where $\mu(s), \mu(s, \alpha)$ are measures of the frequency a state or state-action pair is visited by the agent.

Once the mean (square) error, $\overline{VE}, (\overline{QE})$, has been calculated we apply an optimisation algorithm (*e.g.*, batch gradient descent, stochastic gradient descent Adam or

Nesterov, see **Chapter 7 - Artificial Neural Networks: layer architectures, optimizers and automatic differentiation** for more details) to calculate the gradient of the error and update the weights.

In the framework of Gradient descent, an update rule for the weights can be given by

$$\boldsymbol{w}^{k+1} \leftarrow \boldsymbol{w}^k + a\left(U_t - \hat{v}(s_t, \boldsymbol{w}^k)\right) \nabla_{\boldsymbol{w}} \hat{v}(s_t, \boldsymbol{w}^k), \tag{51}$$

$$\boldsymbol{w}^{k+1} \leftarrow \boldsymbol{w}^k + a\left(U_t - \hat{q}(s_t, \alpha_t, \boldsymbol{w}^k)\right) \nabla_{\boldsymbol{w}} \hat{q}(s_t, \alpha_t, \boldsymbol{w}^k), \tag{52}$$

where the quantity U_t is called the target. The target U_t is an estimation of $V_{\pi^\star}(s)$, (or $Q_{\pi^\star}(s_t, \alpha_t)$). Typically, we can use as a target the accumulated reward at the end of the episode, G_t. We note that as long as the expected target value is equal to the state or action value[17], the update $\boldsymbol{w}^{k+1}$ leads to a local minimum of $\overline{VE}, \overline{QE}$.

In practice, since G_t is not known a priori, we use an estimate of the future rewards after the current state, s_t, by bootstrapping G_t as

$$G_t = r_{t+1} + \gamma \hat{v}(s_{t+1}, \boldsymbol{w}), \tag{53}$$

$$G_t = r_{t+1} + \gamma \hat{q}(s_{t+1}, \alpha_{t+1}, \boldsymbol{w}), \tag{54}$$

respectively. The weight updates can then be given by

$$\boldsymbol{w}^{k+1} \leftarrow \boldsymbol{w}^k + a\left(r_{t+1} + \gamma \hat{v}(s_{t+1}, \boldsymbol{w}) - \hat{v}(s_t, \boldsymbol{w}^k)\right) \nabla_{\boldsymbol{w}} \hat{v}(s_t, \boldsymbol{w}^k), \tag{55}$$

$$\boldsymbol{w}^{k+1} \leftarrow \boldsymbol{w}^k + a\left(r_{t+1} + \gamma \hat{q}(s_{t+1}, \alpha_{t+1}, \boldsymbol{w}) - \hat{q}(s_t, \alpha_t, \boldsymbol{w}^k)\right) \nabla_{\boldsymbol{w}} \hat{q}(s_t, \alpha_t, \boldsymbol{w}^k), \tag{56}$$

respectively. We note that these estimates are not unbiased because they contain the influence of $\hat{v}(s_t, \boldsymbol{w}), \hat{q}(s_t, \alpha_t, \boldsymbol{w})$, *i.e.*, they are dependent on $\boldsymbol{w}$. Therefore, there is no guarantee that the update with this estimate will help convergence towards a local minimum. However, in the case of a 1-step approximation of the target, the point that the algorithm converges is close to the local minimum [SB18].

Of course, this is a regression task that can be solved by many different supervised learning algorithms of ML (see **Chapter 2: Introduction to regression methods** for more details). Such a regression can be performed with the help of deep neural networks (DNNs). In the model-free approaches where the action values optimal $Q_{\pi^\star}(s_t, \alpha_t)$ are approximated, we call the DNN a Deep Q-Network (DQN).

Once the target action value function $Q_{\pi^\star}(s_t, \alpha_t)$ (respectively state value function $V^\star_\pi(s_t)$) have been successfully approximated using $\hat{q}^k(s_t, \alpha_t, \boldsymbol{w})$ (respectively action value $\hat{v}^k(s_t, \boldsymbol{w})$) we can apply any of the previously described learning methods to obtain an estimate of the new target optimal action value $Q^{k+1}_{\pi^\star}(s_t, \alpha_t)$ and or the state value $V^{k+1}_{\pi^\star}(s)$. Usually, the Q-learning method for $n = 1$ is used to update the target values as

$$Q^{k+1}_{\pi^\star}(s_{t_m}, \alpha_{t_m}) = r_{t_m+1} + \gamma \max_{\alpha_{t+1}} \hat{q}^k(s_{t+1}, \alpha_{t+1}, \boldsymbol{w}). \tag{57}$$

[17] Under this assumption the target is *unbiased*.

When a DQN is used for the function approximation of the action value $Q^k(s, \alpha)$ that will later be used by the RL learning algorithm (*e.g.*, Q-learning) for updating the target $Q^{k+1}(s, \alpha)$, is called *Deep Q-Learning*.

2.3.1 Policy gradients

Until now, we focused on learning the optimal action values, $Q(s_t, \alpha_t)$, that maximise the expected accumulated reward, $\max_{\alpha_t} \mathbb{E}[G_t]$. We then established that the optimal policy, $\pi^\star(\alpha_t|\alpha_t)$, simply selects among the actions with the highest action value $\alpha_t^\star = \text{argmax}_{\alpha_t} Q(s_t, \alpha_t)$. However, we have also seen cases where the policy can also be tuned to get higher rewards more frequently (*e.g.*, Monte Carlo control). Now, we advance one step further and introduce a special class of policy gradient algorithms, in which we assume that the optimal policy function can itself be approximated by a linear combination of features θ_i, which constitute a vector $\boldsymbol{\theta}$. Therefore, this is a method that uses function approximation, *i.e.*, $\pi(\alpha_t|s_t) \approx \hat{\pi}(\alpha_t|s_t, \boldsymbol{\theta})$.

We will describe the policy gradient algorithm (which is called *REINFORCE* algorithm, see [Wil92]), which provided a conceptually simple rule for updating the vector of policy parameters $\boldsymbol{\theta}$. This rule aims at improving the expected cumulative reward at each time step of the trajectory by modifying the parameter vector $\boldsymbol{\theta}$ of the policy $\hat{\pi}(\alpha_t|s_t, \boldsymbol{\theta})$.

More specifically, during a trajectory, the cumulative reward for each state-action pair $s_t - \alpha_t$ is given by $G_t(s_t, \alpha_t) = \sum_{k=0}^{T-1} \gamma^k r_{t+k-1}(s_{t+k}, \alpha_{t+k})$. For a stochastic policy $\pi(\alpha_{t+k}|s_{t+k})$ together with stochastic dynamics of the environment $p(s_{t+k}, \alpha_{t+k}, s_{t+k+1})$ the expected cumulative reward is equal to

$$\mathbb{E}_\pi \left[G_t|s_t, \alpha_t\right] = \sum_{k=0}^{T-1} p(s_{t+k+1}, ..., s_0)\gamma^k r_{t+k+1}(s_{t+k}, \alpha_{t+k}, s_{t+k+1}) \tag{58}$$

where $p(s_{t+k+1}, \alpha_{t+k}, s_{t+k}, ..., s_0)$ is the joint probability of a specific trajectory realization up to state action pair s_{t+k}, α_{t+k}.

We note that since the above process is a Markov process of independent events, the above joint probability of reaching the future state s_{t+1} following the trajectory $T_r = \{s_0, \alpha_0, r_1, s_1, \alpha_1, ..., r_{t+1}, s_{t+1}\}$ can be written as

$$\begin{aligned} p(s_{t+1}, r_{t+1}, &\alpha_t, s_t, ..., \alpha_0, s_0) = \\ &p(s_0)\pi(\alpha_0|s_0)p(s_1, r_1|s_0, \alpha_0)\pi(\alpha_1|s_1)p(s_2, r_2|s_1, \alpha_1) \\ &...\pi(\alpha_{t-1}|s_{t-1})p(s_t, r_t|s_{t-1}, \alpha_{t-1})\pi(\alpha_t|s_t)p(s_{t+1}, r_{t+1}|s_t, \alpha_t). \end{aligned}$$

This means that the joint probability of a trajectory reaching future state s_{t+1} and receiving reward r_{t+1} is the same as the product of conditional probabilities between the sequential states, (s_t, s_{t+1}), following the policy of the agent $\pi(\alpha_t|s_t)$ and the dynamics of the environment, $p(s_{t+1}, r_{t+1}|s_t, \alpha_t)$ *i.e.* finding state s_{t+1} after performing action α_t in state s_t. Inserting equation 59 into equation 58 we obtain

$$\mathbb{E}_{\pi}\left[G_t|s_t,\alpha_t\right]=\sum_{k=0}^{T-1}\gamma^k r_{t+k+1}(s_{t+k},\alpha_{t+k},s_{t+k+1})p(s_1,r_1|s_0,\alpha_0)... \\ \times\pi(\alpha_{t+k}|s_{t+k})p(s_{t+k+1},r_{t+k+1}|s_t,\alpha_t) \tag{59}$$

Normally because of the environment dynamics, this information is known only for model-based methods. We choose to approximate the agent's policy with a set of parameters $\boldsymbol{\theta}$, *i.e.*, $\pi(\alpha_t|s_t)\approx\pi_{\boldsymbol{\theta}}(\alpha_t|s_t,\boldsymbol{\theta})$, therefore equation (59) can be approximated as

$$\mathbb{E}_{\pi_{\boldsymbol{\theta}}}\left[G_t|s_t,\alpha_t\right]=\sum_{k=0}^{T-1}\gamma^k r_{t+k+1}(s_{t+k},\alpha_{t+k},s_{t+k+1})p(s_{t+k+1},r_{t+k+1}|\alpha_{t+k},s_{t+k})\pi_{\boldsymbol{\theta}}(\alpha_{t+k}|s_{t+k},\boldsymbol{\theta})... \\ \times p(s_1,r_1|\alpha_0,s_0)\pi_{\boldsymbol{\theta}}(\alpha_0|s_0\boldsymbol{\theta})p(s_0). \tag{60}$$

In order to update the parameters vector, $\boldsymbol{\theta}$, towards the policy that maximises the expected total accumulated reward at the end of the episode, we will consider the gradient of the approximated expected accumulated value of a trajectory with respect to this vector as

$$\nabla_{\boldsymbol{\theta}}\mathbb{E}_{\pi_{\boldsymbol{\theta}}}\left[G_t|s_t,\alpha_t\right]\approx\sum_{k=0}^{T-1}\gamma^k r_{t+k+1}(s_{t+k},\alpha_{t+k},s_{t+k+1})... \\ \times\nabla_{\boldsymbol{\theta}}\left[p(s_{t+k+1},r_{t+k+1}|\alpha_{t+k},s_{t+k})\pi_{\boldsymbol{\theta}}(\alpha_{t+k}|s_{t+k},\boldsymbol{\theta})...p(s_1,r_1|\alpha_0,s_0)\pi_{\boldsymbol{\theta}}(\alpha_0|s_0,\boldsymbol{\theta})p(s_0)\right]. \tag{61}$$

Using the logarithmic property $\nabla_\theta\left(\log(f(\theta))\right)=\frac{\nabla_\theta f(\theta)}{f(\theta)}$ for a given function $f(\theta)$, the last expression becomes

$$\begin{aligned}\nabla_{\boldsymbol{\theta}}\mathbb{E}_{\pi_{\boldsymbol{\theta}}}\left[G_t|s_t,\alpha_t\right]&\approx\sum_{k=0}^{T-1}\gamma^k r_{t+k+1}(s_{t+k},\alpha_{t+k},s_{t+k+1}) \\ &\times\left[p(s_{t+k+1},r_{t+k+1}|\alpha_{t+k},s_{t+k})\pi_{\boldsymbol{\theta}}(\alpha_{t+k}|s_{t+k},\boldsymbol{\theta})...p(s_1,r_1|\alpha_0,s_0)\pi_{\boldsymbol{\theta}}(\alpha_0|s_0\boldsymbol{\theta})p(s_0)\right] \\ &\times\nabla_{\boldsymbol{\theta}}\log\left[p(s_{t+k+1},r_{t+k+1}|\alpha_{t+k},s_{t+k})\pi_{\boldsymbol{\theta}}(\alpha_{t+k}|s_{t+k},\boldsymbol{\theta})...p(s_1,r_1|\alpha_0,s_0)\pi_{\boldsymbol{\theta}}(\alpha_0|s_0,\boldsymbol{\theta})p(s_0)\right], \\ &=\mathbb{E}_{\pi_{\boldsymbol{\theta}}}\left[G_t\nabla_{\boldsymbol{\theta}}\left(\log\pi_{\boldsymbol{\theta}}(\alpha_{t+k}|s_{t+k},\boldsymbol{\theta})+...+\log\pi_{\boldsymbol{\theta}}(\alpha_0|s_0,\boldsymbol{\theta})\right)\right], \\ &=\mathbb{E}_{\pi_{\boldsymbol{\theta}}}\left[G_t\nabla_{\boldsymbol{\theta}}\sum_{k=0}^{T-1}\log\pi_{\boldsymbol{\theta}}(\alpha_{t+k}|s_{t+k},\boldsymbol{\theta})\right].\end{aligned} \tag{62}$$

We note that in this form the gradient update does not require any knowledge of the extraneous environment dynamics, *i.e.*, $p(s_{t+1},r_{t+1}|\alpha_t,s_t,...,\alpha_0,s_0)$ is not needed to evaluate the gradient of the expected accumulated return.

In practice, we cannot compute the expectation directly and therefore we estimate the mean by summing over different trajectories. Thus we obtain

$$\nabla_{\boldsymbol{\theta}} \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left[G_t | s_t, \alpha_t \right] \approx \frac{1}{M} \sum_{m=1}^{M} \sum_{t_m=0}^{T_m} \gamma^{t_m} r(s_{t_m}, \alpha_{t_m}, s_{t_m+1}) \nabla_{\boldsymbol{\theta}} \log \left[\pi_{\boldsymbol{\theta}}(\alpha_{t_m} | s_{t_m}, \boldsymbol{\theta}) \right] \tag{63}$$

Now, we can update the values of the parameter vector $\boldsymbol{\theta}$ towards increasing the expected return $\boldsymbol{\theta} = \max_{\pi_{\boldsymbol{\theta}}} \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[G_t]$ as

$$\boldsymbol{\theta}^{k+1} \leftarrow \boldsymbol{\theta}^k + a \sum_{m=1}^{M} \sum_{t_m=0}^{T_m} \gamma^{t_m} r(s_{t_m}, \alpha_{t_m}, s_{t_m+1}) \nabla_{\boldsymbol{\theta}} \log \left[\pi_{\boldsymbol{\theta}}(\alpha_{t_m+1} | s_{t_m+1}, \boldsymbol{\theta}) \right], \tag{64}$$

where a is a hyperparamater (see MC method), t_m is the current time at episode m, T_m is the final time at episode m, and M is the batch size.

2.3.2 Baseline correction and Advantage function

The main problem with the *REINFORCE* algorithm presented above is the fact that the estimate of the gradient update contains a lot of variance due to the limited number of samples the method averages over. Moreover, the method inherently presents a problem when it averages over states that lead to high rewards because of *vanishing gradients*[18].

We can further increase the convergence rate of the policy gradient by reducing the variance of the gradient without changing its expected value. This can be done by adding a baseline function $b(S_t)$ into the function of the cumulative reward G. Since this function is not a function of the policy parameters θ_i, it does not influence the gradient. The addition of the baseline allows us to better differentiate between good and very good actions. A good baseline function is the use of the state value function $V(s_t)$. This function is independent of the actions α_t and consequently from the policy parameters θ_i.

Using the Policy Gradient Theorem and the state value function $V(s_t)$ we can replace the cumulative reward $\gamma^t r(s, \alpha)$ with the *Advantage function*,

$$A_{\pi_{\boldsymbol{\theta}}}(s_t, \alpha_t) = Q_{\pi_{\boldsymbol{\theta}}}(s_t, \alpha_t) - V_{\pi_{\boldsymbol{\theta}}}(s_t). \tag{65}$$

The values of the parameter vector $\boldsymbol{\theta}$ can then be updated using

$$\boldsymbol{\theta}^{k+1} \leftarrow \boldsymbol{\theta}^k + a \sum_{m=1}^{M} \sum_{t_m=0}^{T_m} \left(Q_{\pi_{\boldsymbol{\theta}}}(s_{t_m}, \alpha_{t_m}) - V_{\pi_{\boldsymbol{\theta}}}(s_{t_m}) \right) \nabla_{\boldsymbol{\theta}} \log \left[\pi_{\boldsymbol{\theta}}(\alpha_{t_m+1} | s_{t_m+1}, \boldsymbol{\theta}) \right]. \tag{66}$$

[18] In a region of the state-action space where all accumulated rewards are high, the gradient of the cumulative reward estimate between the different states vanishes.

Replacing the action value function $Q_{\pi_{\boldsymbol{\theta}}}(s_t, \alpha_t)$ we obtain,

$$\boldsymbol{\theta}^{k+1} \leftarrow \boldsymbol{\theta}^k + a \sum_{m=1}^{M} \sum_{t_m=0}^{T_m} (r(s_{t_m}, \alpha_{t_m}, s_{t_m+1}) + \gamma V_{\pi_{\boldsymbol{\theta}}}(s_{t_m+1}) - V_{\pi_{\boldsymbol{\theta}}}(s_{t_m})) \nabla_{\boldsymbol{\theta}} \log [\pi_{\boldsymbol{\theta}}(\alpha_{t_m}|s_{t_m}, \boldsymbol{\theta})], \tag{67}$$

and when the action value function is used, we obtain,

$$\boldsymbol{\theta}^{k+1} \leftarrow \boldsymbol{\theta}^k + a \sum_{m=1}^{M} \sum_{t_m=0}^{T_m} \delta(s_{t_m}, \alpha_{t_m}, r, s_{t_m+1}, \alpha_{t_m+1}) \nabla_{\boldsymbol{\theta}} \log [\pi_{\boldsymbol{\theta}}(\alpha_{t_m}|s_{t_m}, \boldsymbol{\theta})], \tag{68}$$

where

$$\delta(s_{t_m}, \alpha_{t_m}, r, s_{t_m+1}, \alpha_{t_m+1}) = r(s_{t_m}, \alpha_{t_m}, s_{t_m+1}) + \gamma Q_{\pi_{\boldsymbol{\theta}}}(s_{t_m}, \alpha_{t_m+1}) - Q_{\pi_{\boldsymbol{\theta}}}(s_{t_m}, \alpha_{t_m}).$$

We note that the quantity $\delta(s_{t_m}, \alpha_{t_m}, r, s_{t_m+1}, \alpha_{t_m+1})$ is the *TD-error* of the 1 step TD learning (SARSA) algorithm (see Section 2.2.7).

2.4 Summing up

In the configuration of section 2.3.1 all the tasks, *i.e.*, policy iteration (calculation of the gradients) and policy evaluation (prediction, calculation of the action values) are performed by the same algorithm. This is similar to the *policy iteration* scheme we visited in the classic DP approaches (see Section 2.1.4).

The difference between these approaches and the policy gradient method is that in order for the system to learn a policy of continuous actions we approximated the policy $\pi(\alpha_t|s_t) \approx \pi_{\boldsymbol{\theta}}(\alpha_t|s_t, \boldsymbol{\theta})$ with a set of policy interpolation weights $\boldsymbol{\theta}$.

The biggest problem in the above approach lies in calculating the values of the action value and state value functions ($Q_{\pi_{\boldsymbol{\theta}}}(\alpha_t, s_t), V_{\pi_{\boldsymbol{\theta}}}(s_t)$), respectively. These functions are not known for the whole state-action space, and performing the steps of policy evaluation and value iteration is inefficient.

Instead of calculating the exact $Q_{\pi_{\boldsymbol{\theta}}}$ we can use an approximation - *yet again* - of the action/state value function $Q_{\pi_{\boldsymbol{\theta}}}(s_t, \alpha_t) \approx \hat{q}(s_t, \alpha_t, \boldsymbol{w})$, $(V_{\pi_{\boldsymbol{\theta}}}(s_t) \approx \hat{v}(s_t, \boldsymbol{w}))$ (see Section 2.3). In this way, the maximum value of the action value function approximation $\max_{\alpha_t} Q_\pi \theta(s_t, \alpha_t) \approx \max_{\alpha_t} \hat{q}(s_t, \alpha_t, \boldsymbol{w})$ can be used for the estimates of the model-free algorithms (*e.g.*, Q-learning algorithm).

Notice that the above procedure introduces three problematic elements in comparison to the original methods of DP[19]. These are:

[19]These are also called *The deadly Triad*, see [SB18].

- *Bootstrapping*
- *Off policy learning (e.g. due to the application of the* $\max$ *operator)*
- *Function approximations* for the policy and value functions

Under these conditions, convergence is not assured and the algorithm might show signs of oscillations in the accumulated reward and even diverge. This general difficulty in RL applications leads to the need for large data sets and large training times. Moreover, specific regression architectures need to be put in place for the combination of the policy $\pi_{\boldsymbol{\theta}}(\alpha_t|s_t, \boldsymbol{\theta})$ and action function value $\hat{q}(s_t, \alpha_t, \boldsymbol{w})$.

2.5 Actor-Critic Network (AC)

This is the most common method for handling Policy iteration and Value approximation performed by the DNN. This involves an actor network that takes actions and based on those actions accumulates rewards. In essence the actor network initializes with a random policy $\boldsymbol{\theta}, \pi_{\boldsymbol{\theta}}(\alpha_t|s_t, \boldsymbol{\theta})$ and runs a set of trajectories inside the state-action space (s_t, α_t) of the problem, accumulating rewards, r_{t+1}, and visiting different future state-action pairs (s_{t+1}, α_{t+1}).

As we discussed in the policy gradients and in the summary sections (see Sections 2.3.1 and 2.4), the approximation of the predicted action values
$Q_{\pi_{\boldsymbol{\theta}}}(s_t, \alpha_t) \approx \hat{q}(s_t, \alpha_t, \boldsymbol{w})$ needs to be used. This is done thanks to the separate DQN which is called the critic.

The DQN is also initialized with random weights $\boldsymbol{w}$ and uses the next states, s_{t+1}, actions, α_{t+1}, and rewards, r_{t+1}, obtained by the actor network[20], to find the maximum action value estimates $\hat{q}^k(s_{t+1}, \alpha_{t+1}, \boldsymbol{w})$.

The DQN network then calculates the maximum target action values $\hat{q}^{k+1}(s_t, \alpha_t, \boldsymbol{w})$ based on one of the algorithms of RL (Control, e.g Q-Value iteration, MC-control, SARSA and Q-learning)[21]. The DQN uses and applies the loss function between the target values and the actual predicted values of the action value function
$L\left(\hat{q}^{k+1}_{\boldsymbol{w}}(s_t, \alpha_t) - \hat{q}^{k}_{\boldsymbol{w}}(s_{t+1}, \alpha_{t+1})\right)$, where $L = \|\cdot\|^p$ is a norm. Then it calculates the gradient of the loss to train the weights (see also **Chapter 7 - Artificial Neural Networks: layer architectures, optimizers and automatic differentiation** and **Chapter 8 - Artificial Neural Networks: advanced topics** for more details).

The converged values of the critic are then used to predict the values of action value function for the current state of the actor $\hat{q}^k_{\boldsymbol{w}}(s_t, \alpha_t)$ in the policy gradient estimation. The procedure repeats until the maximum reward is reached.

There are different variants of the AC network. For instance, instead of calculating the loss function of the actor with respect to the action value approximation, we can use the advantage function (see Section 2.3.2). The change between the actor-critic

[20] We emphasize that the actor uses the policy $\pi_{\boldsymbol{\theta}}(\alpha_t|s_t, \boldsymbol{\theta})$ to get $s_{t+1}, \alpha_{t+1}, r_{t+1}$.

[21] We emphasize that the policy used by the DQN is an ε-greedy policy when Q-Value iteration, MC-control or SARSA are used and a plain greedy policy when Q-learning is used.

(AC) and the Advantage actor-critic A2C is minimal if someone uses the SARSA algorithm in the critic network (see Section 2.2.7) and monitors the updates between $\hat{q}^{k+1}_{\boldsymbol{w}}(s_t, \alpha_t) - \hat{q}^{k}_{\boldsymbol{w}}(s_{t+1}, \alpha_{t+1})$. In the advantage actor-critic (A2C) the updates of the policy gradients are done using the error δ of the SARSA algorithm in the policy gradient. We will use this network in the next Section for controlling two different applications in Geomechanics.

3 Applications to Geomechanics

In this Section, we will present two interesting applications of earthquake prevention. In the first, a reduced model of earthquakes (the so-called *spring-slider* model) will be presented for avoiding a fast-slip behaviour. In the second, a geothermal reservoir is introduced to prevent seismicity rate while keeping energy production. In both examples, an RL method will be used to design a linear control and prevent earthquake-like behaviour. But first, we will start by giving a short introduction to Control Theory.

3.1 Control Theory: The basics

Control theory, a discipline within control engineering and applied mathematics, focuses on managing dynamical systems in engineered processes and machines. The primary goal of control theory is to establish models or algorithms that govern the application of system inputs, enabling the system to achieve a desired state or behaviour.

In this chapter, we will only talk about closed-loop control[22]. Such kind of control requires measuring an *output* from the system, calculating an error with respect to a desired reference, and then taking actions with an *input* depending on such signal, taking the states to a desired reference. The algorithm that gets the output as a *feedback* from the system and generates the signal for the input of the system is called a *controller*. This explains the name of *closed-loop* control (see Fig. 19). Therefore, in real applications, a control process requires elements to measure signals (sensors) and actuators to be the input of the system (*e.g.*, motors and valves).

Ktesibios of Alexandria is recognized as the inventor of float valves during the 3rd century BC. These valves were designed to regulate the water level of water clocks, marking a significant milestone as the earliest recorded instance of a controlled process in human history. In such an example, the water clock is the system to-be-controlled, the output is the position of the float valves, the reference is the desired water level, and the valve is the actuator.

Control theory can be classified depending on the system (linear and nonlinear), the number of inputs and outputs (Single-Input-Single-Output, Multiple-Input-Multiple-Output and its combinations), the design (classical using transfer functions or modern using state-space representation), or if there exist perturbations in the model (robust and non-robust), to name a few (see [Oga10, Kha02, SEFL14] for more details). We

[22] Open-loop control, in contrast, is the one that does not depend on the output of the system.

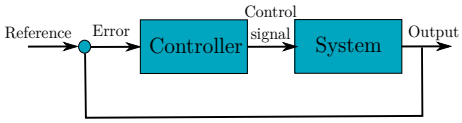


Figure 19: A typical control system diagram.

will only introduce the most known controller: the Proportional-Integral-Derivative (PID) control.

Consider the next (very simple) second-order linear system

$$\begin{aligned} \dot{x}_1 &= x_2, \\ \dot{x}_2 &= u, \\ y &= [x_1, x_2]^T \end{aligned} \tag{69}$$

where y is the output and u is the input of the system. Such a system can be seen as a mechanical system, where x_1 represents a mass position, x_2 its velocity and u represents an acceleration that we can control. The objective is to take the displacement to a **constant** reference, r, by designing correctly the control input u. A classical approach is to take the system to its error dynamics by introducing the next change of coordinates: $e_1 = x_1 - r$, $e_2 = x_2 - \dot{r} = x_2$, obtaining the next system

$$\begin{aligned} \dot{e}_1 &= e_2, \\ \dot{e}_2 &= u. \end{aligned} \tag{70}$$

System (69) and (70) seem to be exactly the same, but there is a fundamental difference: if we can design u in system (70) to take the states, e_1, e_2, to zero, we will solve the main goal of taking the original states, x_1, x_2, in system (69) to the desired reference r. This is due to the introduced change of coordinates. For that purpose, let us present the PID control as

$$\begin{aligned} u &= -k_1 e_1 - k_2 e_2 - k_3 \zeta, \\ \dot{\zeta} &= e_1, \end{aligned} \tag{71}$$

where k_1, k_2, k_3 are control gains to be designed. Note how the PID takes the output $y = [x_1, x_2]^T$ in its design. This linear control takes the *present* (the proportional part e_1), the *past* (the integral part ζ) and the *future* (the derivative part e_2) of the error to drive it to zero. The question is, how can I design the three gains? Let us obtain the *closed-loop system* by substituting the PID control (71) in system (70) as

$$\begin{aligned} \dot{e}_1 &= e_2, \\ \dot{e}_2 &= -k_1 e_1 - k_2 e_2 - k_3 \zeta, \\ \dot{\zeta} &= e_1. \end{aligned} \tag{72}$$

The first thing that we notice is that the PID added another state to the system due to the integral part. Using simple linear algebra, we can obtain the characteristic equation of the system as $\lambda^3 + \lambda^2 k_2 + \lambda k_1 + k_3 = 0$, where λ represents the characteristics values of the system. In order to have exponential stability, *i.e.*, the three states e_1, e_2, ζ will tend to zero exponentially, the characteristic values must have a negative real part. This can be solved, *e.g.* by the Routh-Hurwitz criterion [Oga10], obtaining the next design rules for the gains

$$k_1 > 0, \quad k_2 > 0, \quad k_3 < k_1 k_2. \tag{73}$$

Note how the selection of the gains to have exponential stability is very general and the final choice can be done by adding more requests for the stabilization, like choosing the exact characteristic values of the system by selecting desired time response (how fast we want to reach the origin) and the overshoot (maximum peak value of the response curve measured from the desired reference), to name a few.

In more realistic applications, the gain selection is not as easy to perform due to many factors, like nonlinearities, perturbations, unmodelled dynamics or noise. In the following, we will apply the PID control for two different applications of Geomechanics. Due to the challenging nature of such systems, we will use an RL algorithm to select the best gains for each application.

3.2 Reduced Model for Earthquakes: The spring-slider

The dynamics of earthquakes can be represented, in average/energetical sense, with the spring-slider analogue system (see [Ste19, Ste20, Sch02, KB04, TBS21, GOTSP23]) depicted in Fig. 20.

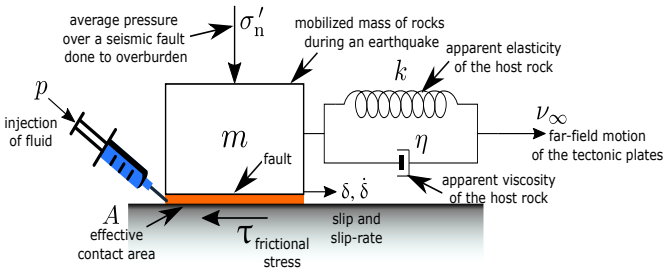


Figure 20: Reduced mechanical model for reproducing earthquake-like instabilities. Figure extracted from [GOTSP23].

This mechanical system consists of a mass, m, which slides on a frictional interface (equivalent to a seismic fault). The mass is connected to a Kelvin-Voigt configuration composed of a spring with stiffness k (equivalent to the apparent elasticity of the host rock) and a dashpot with damping coefficient η (equivalent to the apparent viscosity of the host rock). At the other extremity of the Kelvin-Voigt configuration a constant

velocity, ν_∞, is applied (equivalent to the far-field motion of the tectonic plates). It is assumed Coulomb friction with a friction coefficient $\mu(\delta, \dot{\delta})$ that depends on the slip δ (block's displacement) and the slip-rate $\dot{\delta}$ (block's velocity). According to Terzaghi's principle, the frictional stress τ takes the following form: $\tau = \mu(\delta, \dot{\delta})\,(\sigma_{\text{n}}' - p)$, where σ_{n}' is the constant/reference average effective stress (*e.g.*, the overburden due to the weight of the rocks and the interstitial fluid pressure) and p the fluid pressure developed due to injecting fluid. p is the input to the system for which the controllers will be designed and tested.

According to [Sch02] and [KB04], approximately a rock mass of volume L_{ac}^3 is mobilized during an earthquake event, where L_{ac} is equal to the length of the seismic fault. Therefore, the mobilized mass during an earthquake event is $m \approx \rho L_{\text{ac}}^3$, where ρ is the density of the surrounding seismic fault rocks. The fault length can be calculated as $L_{\text{ac}} = {}^{G}/_{\bar{k}}$, where G is the shear-modulus of the host rock and $\bar{k} = {}^{k}/_{L_{\text{ac}}^2}$, its apparent normalized elastic stiffness. The damping coefficient η is given by $\eta = 2\zeta m\omega_n$, where ζ is the damping ratio and $\omega_n = \sqrt{{}^{k}/_{m}}$, the natural frequency of the reduced system.

The system in Fig. 20 can be represented in a state representation by the following mathematical model [Ste19, Ste20, TBS21, GOTSP23]

$$\begin{aligned} \dot{x}_1 &= x_2, \\ \dot{x}_2 &= -[\mu(x_1, x_2) - \mu^*]\hat{N}\sigma_n' + \mu(x_1, x_2)\hat{N}p - \hat{k}x_1 - \hat{\eta}x_2, \end{aligned} \tag{74}$$

where $x_1 = \delta$ and $x_2 = \dot{\delta}$ are the state variables, $\hat{N} = {}^{A}/_{m}$, $\hat{k} = {}^{k}/_{m}$, $\hat{\eta} = {}^{\eta}/_{m}$, and $A \approx L_{\text{ac}}^2$ is the effective contact area (fault rupture area). The term μ^* represents an initial shifting in the frictional term which results in the system being on the verge of slip (it will start moving) if $\mu^* = \max \mu(x_1, x_2)$.

In this work, the friction coefficient $\mu(x_1, x_2)$ is defined as

$$\mu(x_1) = \mu_{res} - \Delta\mu \cdot e^{-{}^{x_1}/_{d_c}}, \tag{75}$$

with $\Delta\mu < 0$. Such function is defined as a slip-weakening friction law [KB04] and it evolves from an initial value $\mu_{max} = \mu_{res} - \Delta\mu$ (static friction coefficient), to a residual one μ_{res} (kinetic friction coefficient) in a characteristic slip d_c. This will define our term μ^* as $\mu^* = \mu_{max} = \mu_{res} - \Delta\mu$. Notice that such a choice of friction coefficient was made for this academic example. Indeed, the exact frictional rheology is not known in real applications, which creates the need for the design of robust controllers[23]. All the parameters of the reduced model for earthquakes (see eq. (74)) are given in Table 5.

Note that system (74) has an equilibrium point[24] at the origin $x_1^* = x_2^* = 0$, when $p = 0$. According to the nominal studies of [Ste19, Sch02, Die79, GOTSP23], a

[23] A controller that copes with some uncertainties, *i.e.*, unknown system parameters, or disturbances, *i.e.*, an external dynamics affecting the system, is called a robust controller.

[24] We call as equilibrium point to the point, x, in the state space where the derivatives are zero, *i.e.*, $\dot{x} = 0$.

Table 5: Mechanical and frictional properties adopted for the simulations

Parameter	Description	Value
ρ	density	2500 [kg/m^3]
G	shear modulus	30 [GPa]
η	damping coefficient	5×10^{14} [kg/s]
L_{ac}	activated fault length	5 [km]
σ'_n	effective normal stress	50 [MPa]
μ_{res}	residual friction	0.2353
$\Delta\mu$	friction drop	-0.1
d_c	characteristic slip distance	276.35 [mm]
d_{max}	maximum displacement	785 [mm]
t_{op}	operation time	600 [s]
T_s	sampling time	0.5 [s]
M_0	seismic moment	6.25×10^{17} [Nm]
M_w	seismic magnitude	5.8

dynamic instability (a fast-slip behaviour) will take place when the elastic unloading of the springs or the apparent viscosity of the host rock cannot be counterbalanced by friction. This is exactly what happens for system (74), (75), when there is no control ($p = 0$) and when the system parameters are chosen as in Table 5. Fig. 21 shows the fast-slip (earthquake-like) behaviour of such a system, corresponding to an earthquake of magnitude $M_w = 5.8$. The slip, x_1, evolves from an initial state to a maximum displacement, d_{max}, due to the high value of the slip rate ($x_2 \approx 0.075$ [m/s]). The simulation was performed in Python with an implicit solver (BDF) and saving values every $T_s = 0.5$ [s]. The prevention of such fast slips is the main goal in the following.

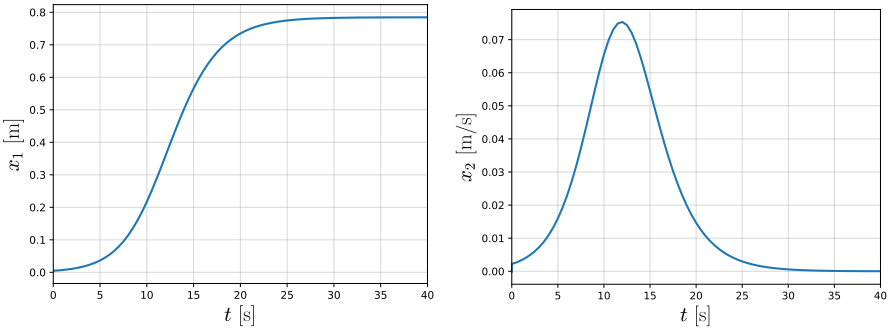


Figure 21: Earthquake-like behaviour in the spring-slider model. The elastic stored energy is released abruptly in the form of kinetic energy, creating a fast slip. Note that such behaviour lasts only a couple of seconds.

3.2.1 Control Objective

As shown in system (74) and Fig. 20, the fluid pressure p is the only input acting on the dynamics of the mechanical system. In a real-scale scenario, fluid injections in the earth's crust change the fluid pore pressure over seismic faults [CSC$^+$19]. As shown in [TBS21, GOTSP23], among others, this can destabilize the fault system and induce/trigger larger earthquakes. To prevent this, one could adequately adjust the fluid pressure (input p) by employing control techniques in order to stabilize the system (74) origin and/or track a reference input signal, releasing the stored elastic energy smoothly (x_1 evolving slowly), and not abruptly.

Therefore, the objective in the following is to design a control law p driving x_1 and x_2 in system (74) to follow some desired predefined references of slow slip rate, $r(t), \dot{r}(t)$, resulting in a slow-aseismic response, **without the knowledge of the system parameters or its dynamics**. The designed control will reduce the natural response time of the system slowing its energy dissipation and eliminating bursts of kinetic energy (earthquake phenomenon).

The desired reference for the output $y = x_1$ is a smooth function reading as

$$r(t) = d_{max} s^3(10 - 15s + 6s^2), \quad \dot{r}(t) = 3d_{max}\frac{s^2}{t_{op}}(10 - 20s + 10s^2), \qquad (76)$$

where $s = t/t_{op}$, d_{max} the target displacement and t_{op} the operational time of the tracking strategy. The constant d_{max} is the distance the fault slides dynamically in order to reach the next stable equilibrium point. Notice that the parameter t_{op} is free to be decided depending on the earthquake control scenario that one wants to apply. Nevertheless, t_{op} has to be sufficiently high with respect to the characteristic time of the earthquake phenomenon, but low enough to achieve aseismic slip with higher velocity than the far-field velocity (v_∞ in Fig. 20), for the control scenario to make sense. The chosen parameters of the reference can be seen in Table 5.

The choice of the reference output $y = x_1$ is motivated by the need to control the average slip over the fault. This average slip is directly connected with the magnitude of an earthquake through the seismic moment [KB04]. Therefore, by controlling the rate of the average slip, the system is forced to release its energy in a quasi-static way, *i.e.*, aseismically. See [Ste19, ST22, GOTSP23] for more details.

3.2.2 Control Design

As we did before, and following [GOTSP23], we will define the tracking error variables as

$$e_1 = x_1 - r, \quad e_2 = x_2 - \dot{r}. \qquad (77)$$

Note that if we design the input p in (74) such that e_1, e_2 tend to zero, we will do exactly what we want it, to force the slip and slip-rate to follow a reference that allows the slow dissipation of energy.

Let us study the error dynamics to see how the control p affects the new states e_1, e_2. Such dynamic is written as

$$\begin{aligned}\dot{e}_1 &= e_2,\\ \dot{e}_2 &= -[\mu(e_1+r)-\mu^*]\hat{N}\sigma_n' + \mu(e_1+r)\hat{N}p - \hat{k}(e_1+r) - \hat{\eta}(e_2+\dot{r}) - \ddot{r},\end{aligned} \tag{78}$$

which is practically the same as (74) except for the new terms due to the reference and its derivatives.

If the exact knowledge of the system parameters and the system dynamics would be available, all the known dynamics in $\dot{e}_2$ could be fought with the input

$$p = \frac{u + [\mu(e_1+r)-\mu^*]\hat{N}\sigma_n' + \hat{k}(e_1+r) + \hat{\eta}(e_2+\dot{r}) + \ddot{r}}{\mu(e_1+r)\hat{N}}, \tag{79}$$

to get the same (very simple) error system as in (70). For that case, we know that if we design the control input u as in (71), the errors would tend to zero by selecting the gains as in (73). This is obviously not the case in real applications where we can not know exactly the system parameters or its dynamics, making it impossible to implement the control (71),(79).

In [GOTSP23], a feedback control was designed with the knowledge of some nominal values of the system parameters and with some knowledge of the dynamics. However, the objective here is to design a controller, p, without any information on the system.

We will define our control, p, as the PID of equation (71), *i.e.*,

$$\begin{aligned}p &= -k_1 e_1 - k_2 e_2 - k_3 \zeta,\\ \dot{\zeta} &= e_1.\end{aligned} \tag{80}$$

As discussed before, such linear control has three parts in order to drive the errors to zero, **if** the reference signal to be followed is constant. This is clearly not our case since the reference was chosen as (76). Nevertheless, as we will show later, such control will be enough to keep the states, x_1, x_2, close to the references, $r, \dot{r}$, avoiding the earthquake-like behaviour.

Due to the fact that we do not know anything about the error dynamics (78), we can not design the gains of the control (80) in a conventional way (*e.g.*, pole location, Lyapunov-based or frequency-based [Oga10, Kha02]), For that purpose, we will use an RL approach in order to select the best possible gains to maximize a reward based on the error variables, dissipating slowly the stored energy of an earthquake. We will start by creating our environment, the Spring-Slider.

This application can be seen as the continuation of [PS21], where an RL approach was also used to avoid earthquake-like behaviour in the spring-slider, but without designing a control algorithm *a priori*, like the PID in (80).

3.2.3 Spring-Slider as environment

In order to train and implement our RL algorithm, an *OpenAI Gym*-based environment of the Spring-Slider will be constructed (see `https://www.gymlibrary.dev/content/environment_creation/` for more details on creating a custom environment).

For the spring-slider system (74) and the control input (80), we define as observations the states x_1, x_2 and the integral action ζ. The actions will be defined as the control gains k_1, k_2, k_3. Therefore, the first step is to define the space of the observations and actions.

```
#Action space will be the minimum and maximum values of the gains k1, k2 and k3
springslider_action_space = gym.spaces.Box(low=np.array([0, 0, 0]),
                            high=np.array([1e8, 1e8, 1e7]), dtype=np.float64)

#Observation space will be the minimum and maximum values of the states x1 (slip),
#x2 (slip-rate) and the integral action
springslider_obs_space = gym.spaces.Box(low=np.array([0, -1e-1, -1e2]),
                         high=np.array([800e-3, 1e-1, 1e2]), dtype=np.float64)
```

Note how we chose high values for the gains to try to compensate for all the unknown dynamics. Next, we define a function for the dynamics of the system, where we have used all the system parameters shown in Table 5:

```
def rhs(t,y,p):
    #System parameters
    rho = 2500 #[kg/m^3]
    G = 30e9 #[Pa]
    eta = 5e14 #[kg/s]
    Lac = 5e3 #[m]
    sigma = 50e6 #[Pa]
    mu_res = 0.2353 #[-]
    delta_mu = -0.1
    dc = 276.35e-3 #[m]
    A = Lac**2 #[m^2]
    m = rho*Lac**3 #[kg]
    k = Lac*G #[N/m]

    #Reference signal
    s = t/top
    ref = dmax*s**3*(10-15*s+6*s**2)

    #Spring-Slider states
    x1 = y[0]
    x2 = y[1]

    #Tracking error
    e1 = x1-ref

    #Friction coefficient
    mu = mu_res-delta_mu*np.exp(-np.abs(x1)/dc)

```

```
    #Spring-Slider dynamics
    x1d = x2
    x2d = -(mu-(mu_res-delta_mu))*A*sigma/m +mu*A*p/m - k*x1/m - eta*x2/m

    #Integral extension
    integrald = e1

    return np.hstack((x1d,x2d,integrald))
```

Then, we can define our class (environment) “SpringSlider” as

```
class SpringSlider(gym.Env):
    def __init__(self):
        self.action_space = springslider_action_space
        self.observation_space = springslider_obs_space

        self.dmax = dmax #[m]
        self.tau = tau #[s]
        self.top = top #[s]

    def observation(self):
        return np.array([self.state[o] for o in self.observations])

    def reset(self):
        self.state = np.array([np.random.uniform(0,5e-4),np.random.uniform(0,1e-4),0])
        self.t = 0
        return self.state

    def step(self, action):
        #States and gains
        x1, x2, integral = self.state
        k1, k2, k3 = action

        #Reference signals and errors
        s = self.t/self.top
        ref = self.dmax*s**3*(10-15*s+6*s**2)
        refd = 3*self.dmax*s**2*(10-20*s+10*s**2)/self.top

        #Tracking errors
        e1 = x1-ref
        e2 = x2-refd

        #PID control
        p = -k1*e1 - k2*e2 -k3*integral

        #Solution of the system dynamics at every step tau
        sol = solve_ivp(rhs, y0=np.hstack((x1,x2,integral)),
        t_span=[self.t, self.t+self.tau], args=[p], method="BDF", t_eval=[self.t+self.tau])
        x1,x2,integral=sol.y[:,-1]
        self.t += self.tau
        self.state = (x1, x2, integral)

        #Norm of the errors and its condition for the reward
        Gamma = np.sqrt(e1**2+e2**2)
```

```
        reward1 = np.exp(-Gamma)

        #Norm of the gains and its condition for the reward
        Gamma_k = np.sqrt(k1**2+k2**2+k3**2)
        max_k = np.sqrt(env.action_space.high[0]**2+env.action_space.high[1]**2
                +env.action_space.high[2]**2)
        reward2 = 1-Gamma_k/(max_k)

        #Total reward
        alpha = 0.5
        reward = (1-alpha)*reward1+alpha*reward2

        #Conditions to stop the simulation
        done = (self.t >= self.top or np.abs(x2)>3e-3)

        return np.squeeze(self.state), reward, done, {}

    def close(self):
        self.isopen = False
```

We defined the observation and the reset functions, where we choose random numbers for the slip and slip-rate, and zero for the integral action, as initial conditions. The solution of the spring-slider and the integral extension dynamics will be obtained at every step using an implicit solver ("BDF"). The reference and its derivative have been designed as (76) and the control p has been designed as (80). Finally, the reward system has been selected depending on the norm of the errors

$$\Gamma(t) = \sqrt{e_1^2 + e_2^2}, \tag{81}$$

and the norm of the control gains

$$\Gamma_k(t) = \sqrt{k_1^2 + k_2^2 + k_3^2}, \tag{82}$$

as

$$\text{reward} = (1-\alpha)e^{-\Gamma(t)} + \alpha\left(1 - \frac{\Gamma_k(t)}{max_{\Gamma_k}}\right), \quad \alpha \in [0,1]. \tag{83}$$

The first term gives rewards depending on how close the states are to the references, while the second gives rewards for using less control magnitude (smaller control gains). Furthermore, the conditions to finish the episode (the variable called "done") are when the time is equal to the operational time, t_{op}, or the velocity is bigger than a certain value. This last condition will force the ML algorithm to learn to avoid the fast-slip behaviour.

One can perform a simulation for the environment without control with the next code lines:

```
#Define the environment as the Spring Slider and check the initial state
env = SpringSlider()
obs = env.reset()
```

```
#Define the case without control, i.e., with the gains equal to zero
action=np.array([0,0,0])
nsteps=int(top/tau)

#Store the states and the time in a variable
x1sc = np.zeros(nsteps+1); x1sc[0] = obs[0]
x2sc = np.zeros(nsteps+1); x2sc[0] = obs[1]
integralsc = np.zeros(nsteps+1); integralsc[0] = obs[2]
tsc = np.arange(nsteps+1)*tau

#Simulation from t=0 to t=t_{op}
for step in range(nsteps):
    obs, reward, done, info = env.step(action)
    x1sc[step+1]=obs[0]
    x2sc[step+1]=obs[1]
    integralsc[step+1]=obs[2]
```

In this case, the states and the time are stored in the variables $x1sc, x2sc, tsc$, where plots can be obtained to check the earthquake-like behaviour of the system (Fig. 21 has been obtained with this procedure).

3.2.4 Deep Deterministic Policy Gradient algorithm

The DDPG [LHP+19] was chosen as RL to select the best control gain, k_1, k_2, k_3 in (80) to maximize the reward system described previously. Such algorithms present an Actor-Critic network in order to train a **continuous** action. This is our case because the PID gains must be chosen according to (73). As explained in Section 2.5, the NN that parametrizes the Q-function is called "the critic", but it exists also the NN of the policy, which is called "the actor". The policy is basically the agent behaviour, a mapping from state to action (in the case of deterministic policy) or a distribution of actions (in the case of stochastic policy).

The parameters of the policy network have to be updated in order to maximize the expected accumulated reward $\mathbb{E}[G_t]$ defined in the policy gradient theorem, while the parameters of the critic network are updated in order to minimize the temporal difference loss (see Section 2.2.7 for more details on TD).

The actor takes the state as input to give an action as output, while the critic takes both state and action as input to give as output the value of the Q function. The critic uses gradient temporal-difference learning while the actor parameters are learned following the policy gradient theorem. The main idea behind this architecture is that the policy network acts producing an action and the Q-network criticizes that action.

The DDPG has been trained for 200 episodes and implemented in the constructed environment. These simulations were made in Python using a sampling time of $T_s = 0.5$ [s]. The results are shown in Figs. 22-23. The slip and slip-rate are close to the reference and its derivative, maintaining the norm of the errors below a certain value. The slip-rate has been made slower (one order of magnitude smaller!) than the

earthquake-like behaviour shown in Fig. 21, obtaining a slow aseismic behaviour. The control signal generated is shown in Fig. 23 and it is the result of the RL algorithm. Although the states always present an error in the tracking (see norm of the error in Fig. 23), these results fulfilled the task of avoiding the fast slip of the original system, by designing a control without the knowledge of any of the system parameters or its nonlinear dynamics.

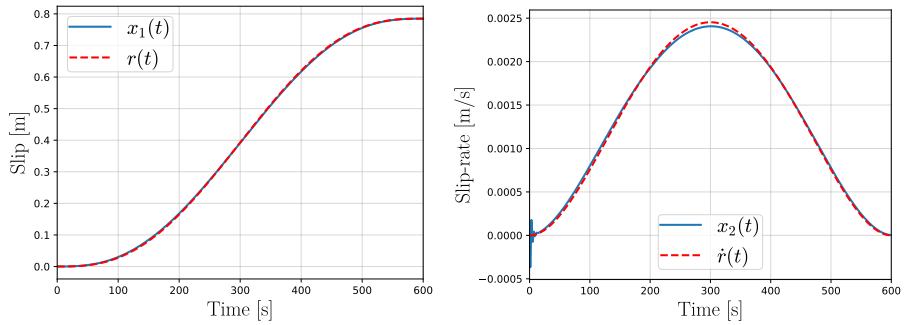


Figure 22: Output tracking of a slow reference due to the linear control and the reinforcement learning approach. Note how the slip and slip-rate follow the desired reference releasing the stored energy one order of magnitude slower than the fast slip behaviour of Fig. 21.

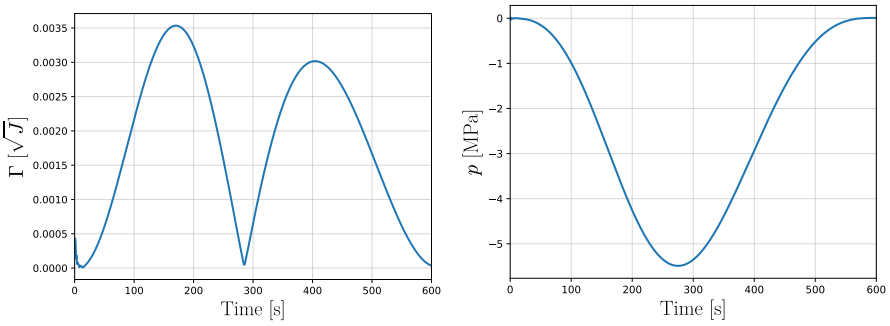


Figure 23: Norm of the tracking error, $\Gamma(t)$ and control signal generated by the PID controller. The control signal was generated by the selection of gains performed by the DDPG algorithm, keeping the norm of the tracking error very close to zero.

3.3 Controlling induced seismicity in a Geothermal Reservoir

In the context of an Enhanced Geothermal System (EGS) [Cor19], one of the objectives is to enhance the permeability between two wells by creating a network of small cracks. These cracks facilitate the circulation of fluids between the wells, promoting

efficient energy extraction. However, the creation of these cracks is often accompanied by localized microseismic activity in the vicinity of the wells. This microseismicity is considered desirable as long as it does not significantly impact the overall seismicity rate over the larger region of the reservoir.

Let's consider an underground reservoir situated approximately 4 [km] below the Earth's surface, as illustrated in Figure 24. This reservoir consists of a porous rock formation that allows the flow of fluids through its pores and fractures. In our example, the reservoir has a thickness of approximately 100 meters and extends horizontally across a square surface with dimensions of approximately 5 [km] by 5 [km]. Within the reservoir, there are various injection points where wells are used to inject and/or extract fluids, such as water, as depicted in Figure 24. For the sake of simplicity, the term "injection of fluids" will encompass both the injection and extraction of fluids from the reservoir.

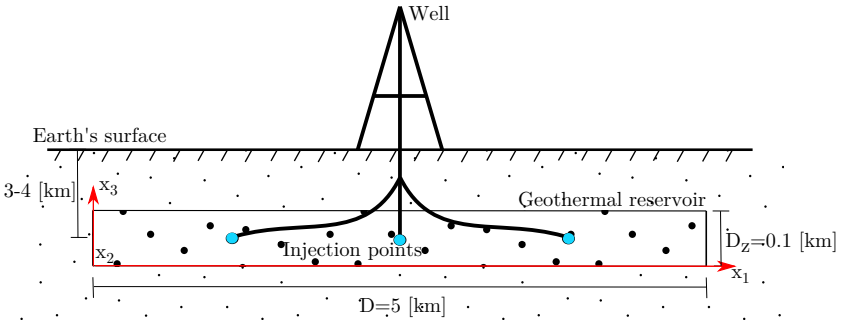


Figure 24: A diagram illustrating an underground reservoir with a thickness of approximately 100 meters and covering a square surface of dimensions approximately 5 [km] by 5 [km]. There are injection points marked within the reservoir where fluids are being injected and/or extracted using wells.

When fluids are pumped deep into the reservoir, it induces the circulation of fluids within the reservoir, resulting in the deformation of the surrounding porous rock. The hydro-mechanical behaviour of the reservoir, caused by the injection of fluids at depth, can be effectively described by Biot's theory [Bio41]. According to this theory, the diffusion of fluid and the deformation of the porous rock are dynamically coupled processes. However, under certain conditions, if the injection rates are sufficiently slow compared to the characteristic times of the system, considering the effects of inertia, and if the volumetric strain rate of the porous rock is negligible, the diffusion of fluid in the host rock due to fluid injections can be accurately approximated using a three-dimensional (3D) diffusion equation [ZCB80].

It is nowadays well established that injecting fluids into the Earth's crust can lead to the creation of new seismic faults and the reactivation of existing ones, resulting in significant earthquakes (refer to [RM15], [KSAC13], and [Zas19] for further information). The underlying physical mechanism behind these induced seismic events is

Table 6: Diffusion and Seismicity rate system parameters

Parameter	Description	Value and Units
c_{hy}	Hydraulic diffusivity	3.6×10^{-4} [km^2/hr]
D	Reservoir length	5 [km]
D_z	Reservoir depth	0.1 [km]
Q_{s_1}	Static flux	0.32 [m^3/hr]
β	Mixture compressibility	1.2×10^{-4} [1/MPa]
V_w	Volume of the well	0.1 [km^3]
μ	Friction coefficient	0.5 [-]
$\dot{\tau}_0$	Background stressing rate	1×10^{-6} [MPa/hr]
t_a	Characteristic decay time	500100 [hr]

closely related to changes in stress within the surrounding rock caused by the fluid injections. These changes can either increase the loading on existing faults or reduce the friction along these faults or newly formed discontinuities. In other words, fluid injections contribute to an elevated seismicity rate in a particular region, referring to the increased number of earthquakes occurring within a given time frame (for more detailed information on seismicity rate, refer to [SL15] and [Die94]).

To illustrate this mechanism, let us consider an injection of $Q_{s_1} = 0.32$ [m^3/hr][25] through a single injection well as shown in Figure 25. In this numerical example, we consider the parameters of Table 6. We then calculate the seismicity rate, R_1, R_2, over two distinct regions, one over the whole reservoir and one close to the injection point (see regions V_1, V_2, respectively, in Figure 25, bottom).

In Figure 25 (top) we plot the seismicity rate in both regions as a function of time. We observe that the maximum seismicity rate over V_1 is equal to $R_1 = 45.91$, which means that 45.91 more earthquakes of a given magnitude are expected over region V_1. The seismicity is even higher (but finite) close to the injection well.

3.3.1 Control Design

In this application, the control problem focuses on achieving a specific control objective: to deliberately increase the seismicity rate in a small region surrounding certain wells while maintaining a constant seismicity rate over the broader area of the reservoir. The aim is to actively manage and control the localized seismic activity, ensuring that it remains within acceptable limits while optimizing the circulation of fluids and energy production in the EGS.

Consider two new control wells, Q_{c_1}, Q_{c_2}, that will be the input of the system (see Fig. 26 for the localization of such control wells in the reservoir). We then perform a similar procedure as in the Spring-Slider application to tune two PID control (one

[25] We use the notation Q_s for a static or fixed flux input and Q_c for a controlled pressure flux input. Do not confuse them with the action value function $Q(s_t, \alpha_t)$.

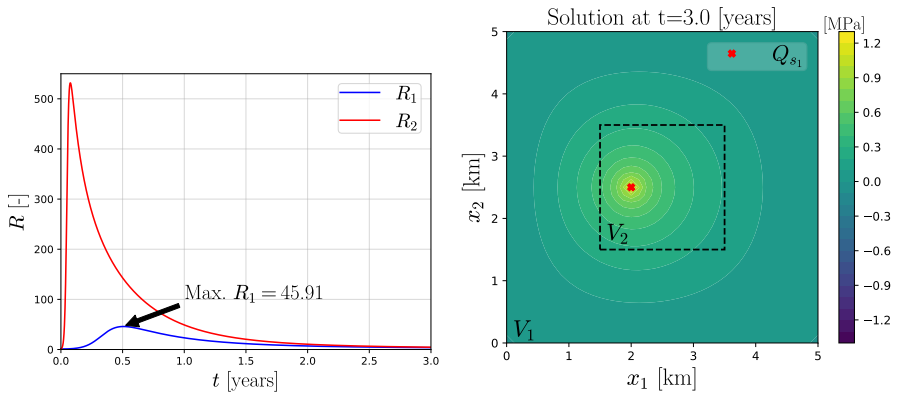


Figure 25: Seismicity rate in both regions, V_1, V_2 (left). Pressure's reservoir after 3 [years] (right). Both results were obtained with a constant injection rate, Q_{s_1}.

for each control input) with the DDPG algorithm to regulate the two seismicity rates of the previously detailed regions V_1, V_2 in two constant values $r_1 = 5, r_2 = 1$. The results are shown in Figs. 26–27. We achieve the goal of maintaining a constant seismicity rate of 5 in the small region V_1 while avoiding induced seismic events in the surrounding region V_2. These results are obtained much faster than the case without control, where the steady state is achieved in 3 [years] (36 [months]) and in this case, it is achieved in 30 [months] (see Fig. 25 for comparison). Therefore, our strategy is able to change the response time of the system! The control signals generated are the result of the DDPG as before, and the norm of the tracking errors is not zero but is maintained very close to the origin.

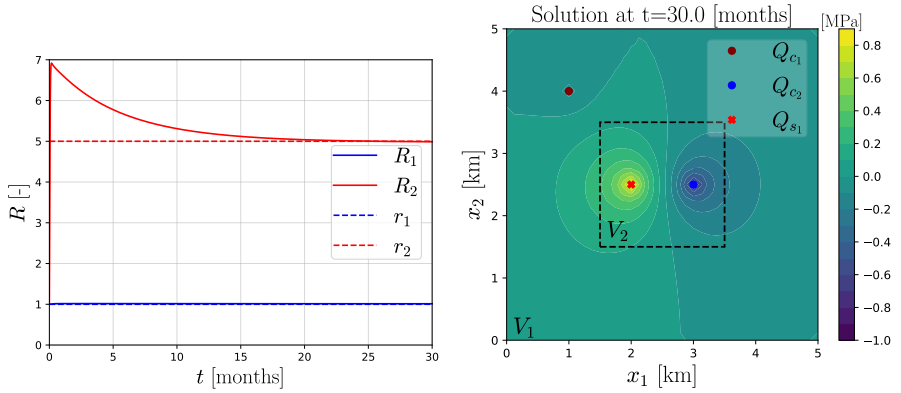


Figure 26: Seismicity rate in both regions, V_1, V_2 (left). Pressure's reservoir after 30 [months] (right). Both results were obtained by adding two control wells.

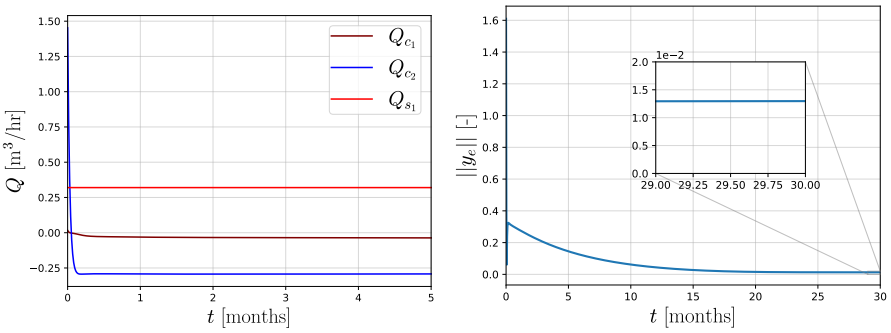


Figure 27: Static flux input Q_{s_1} and controlled flux inputs Q_{c_1}, Q_{c_2} (left). Norm of the tracking error, y_e (right).

This work is a preliminary result. All the details of the calculations and more interesting case studies of the geothermal reservoir are presented in [GOS23].

4 Conclusions

In this chapter, we have introduced the basic concepts of Reinforcement Learning (RL). Starting from the framework of Dynamic programming, we presented the *Bellman Equations*, the *Policy Improvement Theorem* and the *Bellman optimality conditions*. Finally, we presented the model-based methods of *policy iteration* and *value iteration* for finding the optimal state, action values and the determination of the optimal policy.

We expanded the presentation of Dynamic programming to the stochastic case and we introduced the framework of *Markov Decision processes* which generalises the notions of dynamic programming to the stochastic case.

When we don't have complete knowledge of the dynamics of the environment the *model based* methods of dynamic programming cannot be used. For this reason, we presented a second class of *model free* algorithms that can optimise the state and action value functions of the Dynamic Programming problem. Here we presented the algorithms of Monte-Carlo, Temporal differences and Q-learning.

When the number of states and actions in the problem at hand becomes large, the optimization of the values of the state and action value functions of the problem by the above model-free methods becomes intractable. For this reason, function approximation techniques are used to predict the values of the state and action value functions. These values are then used together with the model-free algorithms to improve the estimates of the state and action value functions.

Finally, based on the function approximation of the policy we presented the method of *policy gradients* that aims to improve the policy in order to maximize the expected

state and action values. The methods of direct function approximation of the optimal value functions and the policy gradient algorithm can be used together to provide a powerful actor-critic algorithm suitable for problems with continuous state and action spaces.

Based on an actor-critic network algorithm, we have presented two applications of geomechanics for preventing seismic events without the knowledge of the system dynamics or its parameters. In the first one, a reduced-order model for earthquakes is introduced and the designed control is able to release the stored energy in a slow manner, preventing the fast slip of an earthquake. In the second application, a geothermal reservoir is studied and controlled for preserving energy production, while preventing seismic events in the surrounding area.

The present chapter opens the possibility of more applications in the earthquake prevention field (presence of delays, noise in the sensors, uncertainties in the system parameters, to name a few), thanks to the ability to handle complex and unstructured environments of the RL algorithms.

Acknowledgment

The authors would like to acknowledge the European Research Council's (ERC) support under the European Union's Horizon 2020 research and innovation program (Grant agreement no. 757848 CoQuake). Furthermore, the authors would like to thank Dr Efthymios Papachristos for his support and feedback on the realization of this chapter.

References

[ABC+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, 2016.

[Bel54] Richard Bellman. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503–515, 1954.

[Bio41] M. A. Biot. General theory of three-dimensional consolidation. *Journal of Applied Physics*, 12(155):155–164, 1941.

[BT95] Dimitri P Bertsekas and John N Tsitsiklis. Neuro-dynamic programming: an overview. In *Proceedings of 1995 34th IEEE conference on decision and control*, volume 1, pages 560–564. IEEE, 1995.

[C+15] Francois Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[Cor19] François H. Cornet. The engineering of safe hydraulic stimulations for EGS development in hot crystalline rock masses. *Geomechanics for Energy and the Environment*, 1(1):1, October 2019.

[CSC+19] F. Cappa, M.M. Scuderi, C. Collettini, Y. Guglielmi, and J.-P. Avouac. Stabilization of fault slip by fluid injection in the laboratory and in situ. *Science Advances*, 5(3):eaau4065, 2019.

[Dee22] Google Deepmind. Deepmind x ucl rl lecture series - introduction to reinforcement learning. `https://www.youtube.com/watch?v=TCCjZe0y4Qc&list=PLqYmG7hTraZDVH599EItlEWsUOsJbAodm`, 2022.

[DFB+22] Jonas Degrave, Federico Felici, Jonas Buchli, Michael Neunert, Brendan Tracey, Francesco Carpanese, Timo Ewalds, Roland Hafner, Abbas Abdolmaleki, Diego de Las Casas, et al. Magnetic control of tokamak plasmas through deep reinforcement learning. *Nature*, 602(7897):414–419, 2022.

[Die79] J. H. Dieterich. Modeling of rock friction: 1. experimental results and constitutive equations. *Journal of Geophysical Research: Solid Earth*, 84(B5):2161–2168, 1979.

[Die94] James H. Dieterich. A constitutive law for rate of earthquake production and its application to earthquake clustering. *Journal of Geophysical Research*, 99(B2):2601–2618, 1994.

[Euc23] Jim Euchner. Generative ai. *Research-Technology Management*, 66(3):71–74, 2023.

[Fu16] Michael C Fu. AlphaGo and Monte Carlo tree search: the simulation optimization perspective. In *2016 Winter Simulation Conference (WSC)*, pages 659–670. IEEE, 2016.

[GBGM23] Roberto Gozalo-Brizuela and Eduardo C. Garrido-Merchan. ChatGPT is not all you need. A State of the Art Review of large Generative AI models, 2023.

[Gér22] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. " O'Reilly Media, Inc.", 2022.

[GOOSP23] Diego Gutiérrez-Oribio, Yury Orlov, Ioannis Stefanou, and Franck Plestan. Robust boundary tracking control of wave PDE: Insight on forcing slow-aseismic response. *Systems & Control Letters*, 178:105571, 2023.

[GOS23] Diego Gutiérrez-Oribio and Ioannis Stefanou. Robust Tracking for a 3D Diffusion Equation: Controlling Seismicity Rate in Geothermal Reservoirs. *Control Engineering Practice (submitted)*, 2023.

[GOSP22] D. Gutiérrez-Oribio, I. Stefanou, and F. Plestan. Passivity-based control of underactuated mechanical systems with Coulomb friction: Application to earthquake prevention. *arXiv:2207.07181*, 2022.

[GOTSP23] Diego Gutiérrez-Oribio, Georgios Tzortzopoulos, Ioannis Stefanou, and Franck Plestan. Earthquake control: An emerging application for robust control. theory and experimental tests. *IEEE Transactions on Control Systems Technology*, 31(4):1747–1761, 2023.

[KB04] H. Kanamori and E. E. Brodsky. The physics of earthquakes. *Reports on Progress in Physics*, 67(8):1429–1496, 2004.

[Kha02] H. Khalil. *Nonlinear Systems*. Prentice Hall, New Jersey, U.S.A., 2002.

[KSAC13] K. M. Keranen, H. M. Savage, G. A. Abers, and E. S. Cochran. Potentially induced earthquakes in Oklahoma, USA: Links between wastewater injection7 and the 2011 Mw 5.7 earthquake sequence. *Geology*, 41(6):1060–1067, 2013.

[Lap18] Maxim Lapan. *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more*. Packt Publishing Ltd, 2018.

[LHP+19] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv:1509.02971*, 2019.

[Oga10] Katsuhiko Ogata. *Modern Control Engineering*. Prentice Hall, New Jersey, USA, 2010.

[Ope23] OpenAI. Gpt-4 technical report. arXiv:2303.08774, 2023.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[Pic17] Moxie Pictures. AlphaGo - The Movie. `https://www.youtube.com/watch?v=WXuK6gekU1Y`, 2017.

[PS21] Efthymios Papachristos and Ioannis Stefanou. Controlling earthquake-like instabilities using artificial intelligence. *arXiv:2104.13180*, 2021.

[RM15] J. L. Rubinstein and A. B. Mahani. Myths and facts on wastewater injection, hydraulic fracturing, enhanced oil recovery, and induced seismicity. *Seismological Research Letters*, 86(4):1060–1067, 2015.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[SBW92] R.S. Sutton, A.G. Barto, and R.J. Williams. Reinforcement learning is direct adaptive optimal control. *IEEE Control Systems Magazine*, 12(2):19–22, 1992.

[Sch02] C. H. Scholz. *The Mechanics of Earthquakes and Faulting*. Cambridge University Press, USA, 2002.

[SEFL14] Y. Shtessel, C. Edwards, L. Fridman, and A. Levant. *Sliding Mode Control and Observation*. Intuitive theory of sliding mode control. Birkhauser, New York, USA, 2014.

[SHS+17] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. arXiv:1712.01815, 2017.

[SL15] Paul Segall and S Lu. Injection-induced seismicity: Poroelastic and earthquake nucleation effects. *Journal of Geophysical Research: Solid Earth*, 120(7):5082–5103, July 2015.

[SSZ+22] Md Tanzil Shahria, Md Samiul Haque Sunny, Md Ishrak Islam Zarif, Jawhar Ghommam, Sheikh Iqbal Ahamed, and Mohammad H Rahman. A comprehensive review of vision-based robotic applications: Current state, components, approaches, barriers, and potential solutions. *Robotics*, 11(6):139, 2022.

[ST22] Ioannis Stefanou and Georgios Tzortzopoulos. Preventing instabilities and inducing controlled, slow-slip in frictionally unstable systems. *Journal of Geophysical Research: Solid Earth*, 127(7):e2021JB023410, 2022.

[Ste19] I. Stefanou. Controlling anthropogenic and natural seismicity: Insights from active stabilization of the spring-slider model. *Journal of Geophysical Research: Solid Earth*, 124(8):8786–8802, 2019.

[Ste20] I. Stefanou. Control instabilities and incite slow-slip in generalized Burridge-Knopoff models. *arXiv:2008.03755*, 2020.

[Sut90] Richard S Sutton. Integrated modeling and control based on reinforcement learning and dynamic programming. In R.P. Lippmann, J. Moody, and D. Touretzky, editors, *Advances in Neural Information Processing Systems*, volume 3. Morgan-Kaufmann, 1990.

[Sut99] Richard S Sutton. Reinforcement learning: Past, present and future. In *Simulated Evolution and Learning: Second Asia-Pacific Conference on Simulated Evolution and Learning, SEAL'98 Canberra, Australia,*

November 24–27, 1998 Selected Papers 2, pages 195–197. Springer, 1999.

[TBS21] G. Tzortzopoulos, P. Braun, and I. Stefanou. Absorbent porous paper reveals how earthquakes could be mitigated. *Geophysical Research Letters*, 48(3):e2020GL090792, 2021.

[VEB+17] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. Starcraft ii: A new challenge for reinforcement learning, 2017.

[VSS+19] Ivan Vasilev, Daniel Slater, Gianmario Spacagna, Peter Roelants, and Valentino Zocca. *Python Deep Learning: Exploring deep learning techniques and neural network architectures with Pytorch, Keras, and TensorFlow*. Packt Publishing Ltd, 2019.

[Wil92] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Reinforcement learning*, pages 5–32, 1992.

[WIW89] Chelsea C White III and Douglas J White. Markov decision processes. *European Journal of Operational Research*, 39(1):1–16, 1989.

[Zas19] M. Zastrow. South Korea accepts geothermal plant probably caused destructive quake. *Nature*, 2019.

[ZCB80] O. C. Zienkiewicz, C. T. Chang, and P. Bettess. Drained, undrained, consolidating and dynamic behaviour assumptions in soils. *Geotechnique*, 30(4):385–395, 1980.