AN INTRODUCTION TO DEEP REINFORCEMENT LEARNING

Maxime Toquebiau
Sorbonne Université, CNRS, ISIR, F-75005 Paris, France
ECE Paris
maxime.toquebiau@gmail.com

Jae-Yun Jun
ECE Paris

Faïz Benamar
Sorbonne Université, CNRS, ISIR, F-75005 Paris, France

Nicolas Bredeche
Sorbonne Université, CNRS, ISIR, F-75005 Paris, France

ABSTRACT

This article presents an introduction to the field of deep reinforcement learning. It is primarily intended for students taking their first steps in this field. First, a general introduction to reinforcement learning is given, defining the approach and describing its origins and current trends. Then, the essential foundations of reinforcement learning algorithms — i.e., value, policy, and model — are described. Then the main model-free deep reinforcement learning algorithms are explained. Finally, current directions of research on deep reinforcement learning are introduced.

Contents

1 Introduction

1.1 Reinforcement learning paradigm

Reinforcement Learning (RL) is a machine learning paradigm that enables artificial agents to learn a strategy for behaving in order to maximise some reward signal. The core mechanic of RL is trial-and-error: the learner sequentially tries actions, obtains rewards, and updates their strategy based on their findings. Throughout many of these cycles, the learner progressively acquires knowledge about the task. From this knowledge, a strategy for solving the task can be devised.

This trial-and-error loop makes for the first crucial distinction with other machine learning paradigms: the RL agent is the primary source of its training data. In *supervised* and *unsupervised* learning, a fixed set of data points are given beforehand and used learn a mapping between inputs and outputs to optimise some objective over this data. In RL, there is no necessary pre-existing data[1], only some definition of the learning environment and agent. From there, the agent will generate its own training data by acting in the environment, learning from its experience, and so on.

Another key difference is in the objective optimised during training. In supervised and unsupervised learning, the learning task is defined by a mathematical cost function directly computed over the model's prediction and the data points — e.g., minimising the prediction error of an image recognition model. In RL, the learning task is defined by the rewards signal that should be maximised by the agents. While this reward is often refer to as a function, in practice, it is not a mathematical function of the learner's actions. The reward is a signal given by the environment to the agent: it may be the score in a video game, a note given by a human observer, or a win state in a tabletop game — i.e., 1 for "game won", -1 for "game lost". The goal of the RL algorithm is to shape the agent's strategy for selecting actions in order to maximise the received rewards. This requires the use of a series of mathematical tricks to translate the objective of "maximising the rewards" into a learnable objective linked to the agent's actions.

These two specific characteristics of the RL paradigm are both major sources of trouble. Because the agent generates its own training data, the way it behaves directly impacts the data distribution it will observe. Throughout training, its behaviour changes as a result of each learning updates, which in turn changes the training data distribution. Overall, this self-generated data collection mechanism is a great source of variance, making training more prone to the influence of all parameters involved — e.g., seed, hyperparameters, environment parameters. In addition, the lack of explicit mathematical link between the actions and the task performance makes updating the agent's behaviour particularly difficult, as it requires techniques to estimate the actions' impact on the observed performance and deduce a way to

[1] Even though it is possible to use RL techniques over pre-existing data points.

update the behaviour. This is, again, another source of variance as these estimations are often imperfect, thus introducing noise in learning.

For these reasons, RL is difficult to implement effectively. In practice, supervised learning is always preferred to RL if the context allows it. However, RL remains a great tool for artificial intelligence, as it allows to optimise any given numerical signal. Thus, in any task where it is not possible to compute a learning objective directly from the model's outputs — because the performance depends on external, possibly qualitative, measures —, supervised and unsupervised learning become useless, and RL can be used instead.

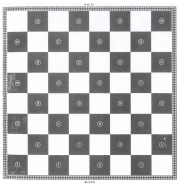
1.2 Trends and challenges

A particularly appealing aspect of RL lies in its way of emulating the way humans and other animals learn in the real world. In fact, while the term "reinforcement learning" now refers widely to a class of learning algorithms, the idea of learning through reinforcement has also been used for a long time beyond the scope of computer science. Many concepts of RL derive from neuroscience and are still used to describe the functioning of our brain (Schultz et al., 1997; Friston, 2010) or to study mental illness (Montague et al., 2012). Reinforcement experiments are used in biology and psychology to characterise intelligence and investigate how learning occurs in animals (Rescorla & Wagner, 1972; Gardner & Gardner, 1984; Brembs, 2010). RL is also closely tied to evolutionary biology, where evolution can be described as finding the best strategy to adapt to the environment through natural selection. This definition is at the core of "Evolutionary Algorithms" (Eiben & Smith, 2003) which shares common tools with RL.

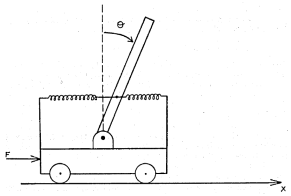
In computer science, RL was introduced during the second half of the 20th century, from various fields of research in neuroscience, optimisation, control theory, and electrical engineering (Sutton & Barto, 2018). Two main threads can be recognised as direct origins of RL theories. One stems from the ethologist study of learning by trial-and-error, and especially with reinforcing events (Thorndike, 1911; Rescorla & Wagner, 1972). These ideas influenced early works in artificial intelligence, later integrating trial-and-error learning into electrical engineering (Walter, 1950) and computer science (Minsky, 1961). The second branch is *dynamic programming* (Bellman, 1966), which investigated solving a complex problem by breaking it into easier sub-problems. This involved the definition of many mathematical tools that became the foundations of RL algorithms (see Section 2).

For practical reasons, the development of RL techniques has tackled many different recreational games, starting with simple ones like checkers (see Figure 2.1.a; Samuel, 1959) and tic-tac-toe (Michie & Chambers, 1968). Such games offer a convenient setting for reinforcement learning: the rules are easy to translate into a reward signal — e.g., 1 if the game is won, 0 otherwise — and the game is simple enough to be simulated in computers. This last point is crucial to allow running fast experiments in simulation. Indeed, RL tends to be *sample inefficient*, i.e., it requires a large number of experiences to find a good strategy. Thus, simulated games allow acquiring these experiences extremely fast, sometimes even playing multiple games in parallel. With more powerful computers, increasingly difficult games have been tackled with RL methods. Notable instances are TD-Gammon for backgammon (Tesauro, 1994) and AlphaGo (see Figure 2.1.f; Silver et al., 2016) which demonstrated how RL techniques could be used to achieve superhuman performance in the challenging game of Go. Following tabletop games, video games have recently been used as playgrounds for RL research. They offer similar advantages while providing increased complexity in the variety of tasks and environments. The Atari benchmark (see Figure 2.1.e; Bellemare et al., 2013) has been broadly adopted for its wide range of different games with agents playing by looking at the pixel images, just like humans do. Recently, the popular video game Minecraft has been invested as a new challenge for artificial intelligence research, thanks to its extremely diverse environment and vast set of incremental skills (see Figure 2.1.h; Oh et al., 2016; Guss et al., 2019), showing the great potential of modern video games for artificial intelligence research.

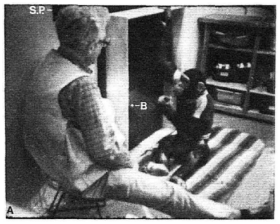
Concurrently to games, RL has also been employed for controlling robots. An early example of this is the classical pole-balancing problem (Michie & Chambers, 1968) that features a pole balancing on a moving cart (see Figure 2.1.h): the cart can move right or left and the reward is made only of a failure signal when the pole falls or the cart reaches the end of the track. While controlling robots is a major drive of RL research, it remains a great challenge. Robotics problems feature high-dimensional, continuous states and actions, which contrast with the discrete, finite state and action spaces of tabletop games. To overcome this, solutions include partitioning the space of states or actions to reduce complexity using linear function approximation (Busoniu et al., 2017), or using neural networks to extract features from continuous states automatically (Bishop, 2006) (as we will see in Section 4). Another attribute of robotic environments is partial observability, with robots having only incomplete information about the current state of the environment. This has been tackled by learning a model of the environment's dynamics (Bagnell & Schneider, 2001) (see Section 3.3) or by providing the learner with memory to keep track of past information (Wierstra et al., 2007). Another challenge in robotic settings is the design of a reward function. This requires translating the robot's objective into a numerical signal, which can be tedious. Intuitively, a robot could be rewarded positively when it completes its task. But this would mean having only one reward signal for the whole episode. Such a sparse reward signal is an issue for RL as it



(a) Checkers
(Samuel, 1959)

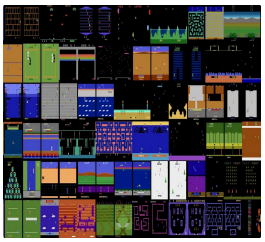


(b) Pole-balancing problem
(Michie & Chambers, 1968)

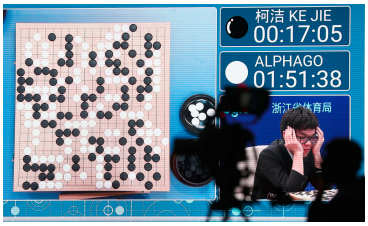


(c) Teaching word to a chimpazee
(Gardner & Gardner, 1984)

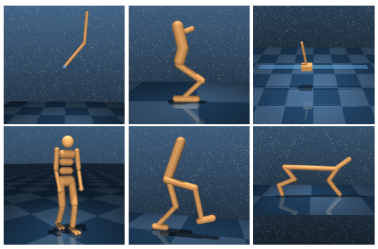
(d) Flying an helicopter
(Bagnell & Schneider, 2001)



(e) Atari benchmark
(Bellemare et al., 2013)

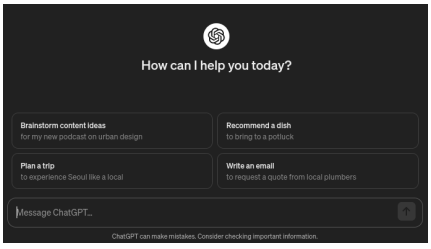


(f) AlphaGo vs. best human
(Silver et al., 2016)
(Photograph by Wu Hong, in The Guardian)



(g) Deepmind Control Suite in MuJoCo
(Tassa et al., 2018)

(h) Learning skills in Minecraft
(Guss et al., 2019)



(i) ChatGPT (chat.openai.com)
(Ouyang et al., 2022)

Figure 1: Reinforcement learning environments in various domains of research.

provides very few guidance for knowing what actions are contributed towards succeeding — or failing — in the task. To fix this, reward shaping methods (Ng et al., 1999; Laud, 2004) construct reward functions to consistently guide the learner towards the objective in a limited number of steps. Finally, a great challenge for learning robotic control with RL is the cost of experimenting with robots. Learning by trial-and-error and sample inefficiency make experimenting with physical hardware expensive and time-consuming. For these reasons, a preferred approach is often to learn a model from demonstrations of wanted behaviour, imitating the actions of human tutors (Hester et al., 2018), which can then fine-tuned with RL (Abbeel et al., 2006; Vecerik et al., 2017). Learning from demonstrated sequences of actions simplifies the learning problem by recasting it as a supervised learning problem. Another solution is use simulated environments like ROS (Macenski et al., 2022) or MuJoCo (see Figure 2.1.g; Todorov et al., 2012). However, a model learnt in simulation will perform poorly in the real world due to the numerous inconsistencies between the two settings. To tackle this *reality-gap* challenge, researchers have explored ways to make the RL agents more robust to the real world transfer by adding noise and variety in the simulated environment (Tobin et al., 2017; Peng et al., 2018).

While control has been a main focus of RL, it has also been studied in other domains of artificial intelligence research. A recent success has been the use of Reinforcement Learning from Human Feedback (RLHF; Christiano et al., 2017; Stiennon et al., 2020) in language modelling. Language modelling is the task of learning to predict the next word in a given sequence. In this context, RLHF fine-tunes initially supervised language models to better fit human preferences. This technique is made possible by using reward modelling (Leike et al., 2018), where a model learns to predict the rewards given by humans and is then used to train the RL model on a large number of generated experiences. This allowed the development of conversational agents like the now widely used ChatGPT (Ouyang et al., 2022). The success of RLHF approaches demonstrates the great potential of RL for learning to fit human requirements.

All these examples show the tremendous potential of RL to shape artificial intelligence research further. This is largely due to the advent of deep reinforcement learning that sparked an ongoing revolution in the field during the last decade. Today, RL is widely considered an important block to building more advanced forms of artificial intelligence (Silver et al., 2021). However, this view is not shared by all (Mitchell, 2021) and RL still suffers from many limitations. We have already mentioned the problems of sample inefficiency, safety, and the high cost of RL training. They all generally hinder performance and prevent RL algorithms from being used extensively in robotics (Sünderhauf et al., 2018; Ibarz et al., 2021) or autonomous driving (Kiran et al., 2022; Chen et al., 2023), with supervised alternatives like imitation learning being far superior. These issues motivate various practical solutions (Ibarz et al., 2021), like the aforementioned reward shaping strategy (Laud, 2004), as well as whole lines of research such as safe RL that explicitly constrain RL to learn from safe states (García et al., 2015), and curriculum learning that focuses on designing a schedule of increasingly difficult tasks to improve training efficiency (Bengio et al., 2009; Uchendu et al., 2023). RL also suffers from more practical issues inherent to its definitions and the tools it uses, such as issues with reproducibility, high variance, and intricacy of the algorithms and their implementations (Henderson et al., 2018). As with the rest of methods based on deep neural networks, there is also an issue with explainability and interpretability of deep RL models. Deep neural networks act as black boxes with no explicit mean for interpreting their reasoning and, thus, explaining their results (Rudin, 2019; Samek et al., 2021). With deep RL agents, interpreting the learnt behaviours and understanding how training shaped them this way is often difficult.

2 Elements of Reinforcement Learning

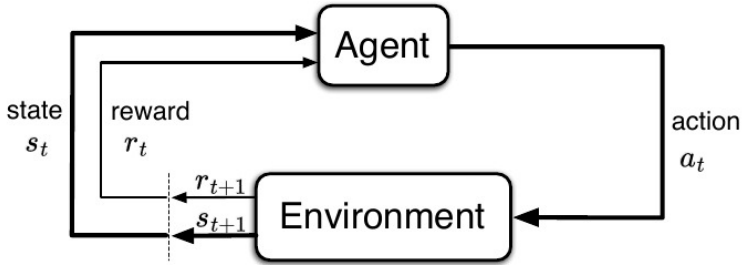


Figure 2: Diagram of the reinforcement learning interaction between the agent and its environment, from Sutton and Barto (2018).

Reinforcement learning is the process of learning how to complete a particular **task** from trial-and-error. In this process, we refer to the learner as an **agent**. The agent interacts with its **environment** over a sequence of discrete time steps $t = \{0, 1, 2, ...\}$. Each interaction consists in the agent receiving information on its **state** s_t from the environment and choosing an **action** a_t to perform. Following the agent's action, the environment produces a **reward** r_{t+1} and a new

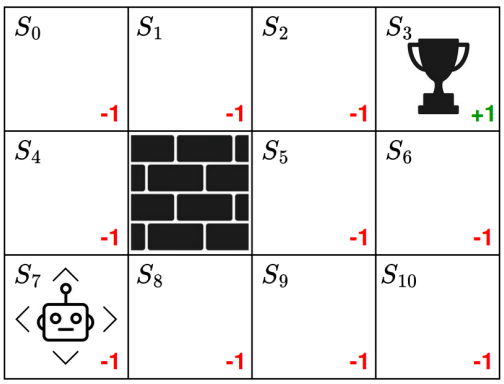


Figure 3: Simple reinforcement learning environment where the goal is for a robot to navigate in the grid and find the trophy cell. The robot can move in all four directions. The episode ends if the robot reaches the trophy or if the maximum number of steps T is reached. States are defined by the position of the robot in the grid. Rewards for entering each state are indicated in the bottom-right corner of each cell.

state s_{t+1}. This process, illustrated in Figure 2, repeats indefinitely or stops after a finite number of steps T. In the latter case, we call this finite sequence of steps an **episode**. In the context of RL, the agent's goal is to pick the right actions to maximise the sum of future rewards.

The agent refers to the central entity of the experiment. It is often considered *"intelligent"* for being: (i) **reactive** to its environment, perceiving information from its surroundings and acting in response to these signals; (ii) **proactive**, shaping its behaviour to satisfy its goals; and (iii) able to **learn** from its experience (Wooldridge & Jennings, 1995). The goal of the agent is defined primarily by the task. It can be as straightforward as "picking up an object", or be a more abstract objective like "give this person what they need". The environment refers to the setting where this task is conducted, encompassing all elements except for the agent. The frontier between the agent and the environment is not a physical boundary but rather a conceptual one: everything that cannot be changed arbitrarily by the agent is considered part of the environment. For example, sensors, motors and mechanical joints of a robot should be considered parts of the environment as they are subject to forces independent to the agent, which in this case refers to the program controlling the robot. The states and actions depend on the agent's capabilities in the environment. Like the goal, they can take different forms, with various levels of abstraction. For example, the state of a robot could be concrete information about its surroundings coming from its sensors (e.g., camera, lidar, inertial measurement unit), or more symbolic information like the state of being in a particular room or not. Similarly, actions can be as concrete as "turning 4.5 degrees left", or more high-level like "flip the pancake". The RL framework accounts for all these levels of abstraction, allowing its use in many different settings.

2.1 Markov Decision Processes

The universally accepted framework for building RL algorithms is the **Markov Decision Process** (MDP; Howard, 1960). The MDP is a mathematical formalisation of the RL problem. It defines this problem as a tuple $\langle \mathbf{S}, \mathbf{A}, \mathcal{T}, \mathcal{R} \rangle$. In this tuple, $\mathbf{S}$ represents the set of all possible states in the environment: $\mathbf{S} = \{S_0, S_1, S_2, ...\}$. Similarly, $\mathbf{A}$ is the set of all possible actions: $\mathbf{A} = \{A_0, A_1, A_2, ...\}$. In the example of Figure 3, $\mathbf{S}$ is the set of all eleven possible positions in the grid, numbered from 0 to 10, and $\mathbf{A}$ contains four possible actions: $\mathbf{A} = \{\texttt{move\_up}, \texttt{move\_right}, \texttt{move\_down}, \texttt{move\_left}\}$. This setup describes the particular case of a *finite* MDP, where the number of possible states and actions are finite. However, an MDP could also allow infinite states and actions if needed. For example, an autonomous driving system could have its state defined as a continuous GPS position and its action as the continuously defined rotation angle to apply to the steering wheel.

The **transition function** $\mathcal{T}$ defines the probability of transitioning from state s to state s' by taking action a:

$$\mathcal{T}(s' \mid s, a) \coloneqq \Pr\{s_{t+1} = s' \mid s_t = s, a_t = a\}. \tag{1}$$

It dictates how the environment changes after performing an action. The transition function translates the agent's move in terms of probability: for example, in the setup of Figure 3, $\mathcal{T}(S_1|S_0, \texttt{move\_right}) = 1$ and $\mathcal{T}(S_4|S_0, \texttt{move\_rigth}) = 0$. These probabilities describe a *deterministic* environment, where a given action performed in a particular state always leads to the same outcome. On the other hand, in *stochastic* environments, an action may have multiple possible outcomes. For example, an action may fail with a probability of 0.01: $\mathcal{T}(S_1|S_0, \texttt{move\_rigth}) = 0.99$ and

$\mathcal{T}(S_0|S_0, \texttt{move\_rigth}) = 0.01$. In our example synthetic environment, we control the probability of each transition. However, in more realistic settings, the transition function is unknown.

Finally, the **reward function** $\mathcal{R}$ defines the reward $r(s, s')$ given by the environment for the transition from a state s to a state s'. From this, we can compute the expected value of the reward obtained during a transition from state s to s' with action a:

$$\begin{aligned}\mathcal{R}(s, a, s') &\coloneqq \mathbb{E}[r_{t+1} = r(s, s') \mid s_t = s, a_t = a, s_{t+1} = s'] \\ &= \sum_{s' \in \mathbf{S}} \mathcal{T}(s' \mid s, a) r(s, s')\end{aligned} \tag{2}$$

In Figure 3, the reward function is made of two elements: a positive reward signal if the agent reaches the goal and a negative signal for all other states. The former is straightforward: it indicates that the task is complete. The latter penalises the agent for entering any state that is not the terminal state. This kind of penalty is used often to urge the agent to complete the task as fast as possible: the more steps are taken, the larger the accumulated penalty. The reward depicted in Figure 3 can be described as *sparse* because there are very few positive reward signals to guide the agent towards the objective. In this simple environment, this should not be an issue. But, in more complex settings with a larger number of possible states, sparse rewards can become problematic.

The transition probability and reward function are often referred to as the **environment dynamics**, corresponding the probability:

$$p(s', r \mid s, a) \coloneqq \Pr\{s_{t+1} = s', r_{t+1} = r \mid s_t = s, a_t = a\}, \tag{3}$$

which is the probability of being in state $s' \in \mathbf{S}$ and receiving reward $r \in \mathbb{R}$, after performing action $a \in \mathbf{A}$ in state $s \in \mathbf{S}$.

2.2 Modeling the Agent

In the context of an MDP, the agent is modelled as a **policy** that selects actions depending on the current state of the environment. This policy can be either *deterministic* or *stochastic*. A deterministic policy is a function π that directly maps the current state s to an action a to perform: $\pi : \mathbf{S} \to \mathbf{A}$, $\pi(s) = a$. A stochastic policy π is defined as a function that maps states to probabilities over possible actions: the probability of choosing action $a \in \mathbf{A}$ in state s is $\pi(a|s) \in [0, 1]$, with $\sum_{a \in \mathbf{A}} \pi(a|s) = 1$. In some algorithms, the policy is explicitly learnt during training. But the policy may also be derived from other learnt elements, or even random or unknown.

The agent uses the policy function to act in the environment, which results in a sequence of states, actions, and rewards that is called a **trajectory**, denoted τ:

$$\tau = (s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, ...). \tag{4}$$

The goal of an RL algorithm is to find a policy that maximises the cumulative rewards in the generated trajectories. To that end, the policy must choose actions to maximise the sum of future rewards which is called the **return** G_t, starting from any step t:

$$G_t \coloneqq r_{t+1} + r_{t+2} + r_{t+3} + ... + r_T, \tag{5}$$

with the episode ending at step T. In the case where T is infinite (i.e., the episode never ends) this formulation is problematic as it can result in an infinite return. This motivates another formulation of the return where future rewards are discounted by their distance in time to the current step. This **discounted return** is defined as:

$$G_t \coloneqq r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \tag{6}$$

with $\gamma \in [0, 1]$ the *discount factor*. The discounted return favours rewards that are closer in time. The value of the discount factor controls how far in time the reward should affect the decision of the agent: a γ close to zero favours immediate rewards, while a γ close to one favours long-term returns. In practice, almost all RL methods use the discounted return, even if the episode ends in a finite number of steps T.

Many RL algorithms use *value functions* that compute or estimate the expected return from any point in the trajectory. The **state-value function** associated with policy π, is defined as the expected return starting from any state s and following policy π:

$$V_\pi(s) \coloneqq \mathbb{E}_\pi[G_t \mid s_t = s], \tag{7}$$

where $\mathbb{E}_\pi[.]$ is the expected value of any random variable given that the agent always follows the policy π. Similarly, the **action-value function** associated with π computes the expected outcome of a particular action in a particular state, following π:

$$Q_\pi(s, a) \coloneqq \mathbb{E}_\pi[G_t \mid s_t = s, a_t = a]. \tag{8}$$

As we will see in further sections, these value functions can be computed given the environment dynamics and a policy, or learnt from experience and then used to choose actions accordingly or to do planning over multiple time steps.

An important property of the discounted return defined in Equation 6 is its successiveness: the return at step t can be expressed in function of the return of the following steps:

$$\begin{aligned} G_t &\coloneqq r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + ... \\ &= r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + \gamma^2 r_{t+4} + ... \\ &= r_{t+1} + \gamma G_{t+1}. \end{aligned} \tag{9}$$

The same property can therefore be found for the value functions, here with the state-value:

$$\begin{aligned} V_\pi(s) &\coloneqq \mathbb{E}_\pi[G_t \mid s_t = s] \\ &= \mathbb{E}_\pi[r_{t+1} + \gamma G_{t+1} \mid s_t = s] \\ &= \mathbb{E}_\pi[r_{t+1} + \gamma V_\pi(s') \mid s_t = s, s_{t+1} = s']. \end{aligned} \tag{10}$$

The *optimal policy* is defined as the policy whose expected return is greater than or equal to that of all other possible policies, for all possible states. In other words, a policy π is the optimal policy if $V_\pi(s) \geq V_{\pi'}(s)$, for all $s \in \mathbf{S}$ and all possible other policies π'. The optimal policy, denoted π^* is associated with the optimal value functions V^* and Q^*. By definition, the optimal policy always chooses the action with the largest expected return, which can be noted as:

$$\pi^*(s) = \arg\max_{a \in \mathbf{A}} Q^*(s, a). \tag{11}$$

Following this, the optimal state-value function of any state s is equal to the maximum value of Q^* in this state:

$$\begin{aligned} V^*(s) &= \max_{a \in \mathbf{A}} Q^*(s, a) \\ &= \max_{a \in \mathbf{A}} \mathbb{E}_{\pi^*}[G_t \mid s_t = s, a_t = a] \\ &= \max_{a \in \mathbf{A}} \mathbb{E}_{\pi^*}[r_{t+1} + \gamma V^*(s') \mid s_t = s, a_t = a, s_{t+1} = s']. \end{aligned} \tag{12}$$

This last result is the *Bellman optimality equation* (Bellman, 1957) for the state-value function. By unfolding the expected value, we get:

$$V^*(s) = \max_{a \in \mathbf{A}} \sum_{s' \in \mathbf{S}} \mathcal{T}(s' \mid s, a) \left[\mathcal{R}(s, a, s') + \gamma V^*(s')\right]. \tag{13}$$

It can also be expressed for the optimal action-value function as:

$$\begin{aligned} Q^*(s) &= \mathbb{E}_{\pi^*}[r_{t+1} + \gamma \max_{a' \in \mathbf{A}} Q^*(s_{t+1}, a') \mid s_t = s, a_t = a] \\ &= \sum_{s' \in \mathbf{S}} \mathcal{T}(s' \mid s, a) \left[\mathcal{R}(s, a, s') + \gamma \max_{a' \in \mathbf{A}} Q^*(s', a')\right]. \end{aligned} \tag{14}$$

Bellman optimality equations are the basis of most RL algorithms, as they allow to gradually learn the optimal value functions through an iterative process described in Section 3.1.1. In finite MDPs, the optimal value functions are proven to exist and be unique. But, that is not the case for more complex environments. Therefore, RL algorithms usually learn to estimate these functions through various techniques.

Finally, another major technique used in RL is a **model** of the environment. A model predicts the outcomes of the agent's action in terms of changes in the environment and resulting rewards. Concretely, a model learns to approximate the environment dynamics p defined in Equation 3. Thus, the model is a function $\widehat{p}$ that maps the current state and action to the next state and reward. It can output a direct prediction, with $\widehat{p} : S \times A \to S \times \mathbb{R}$, or a probability distribution over states and rewards (like in Equation 3). These two forms of models can be used in different ways for doing *planning* over one or multiple time steps. As we will see in Section 3.3, planning usually involves looking ahead by predicting future outcomes and then using these predictions to compute or improve value function predictions, which, in turn, can be used to select actions that maximise the expected return.

3 Foundational RL Algorithms

After introducing the core elements used in RL, we present different approaches to learning from reinforcement. These approaches rely on learning value functions, policies, models, or a combination of the three, to devise an efficient strategy of actions. In this section, we present the foundational RL algorithms that serve as a basis for most deep RL algorithms.

3.1 Value Estimation

As presented in the last section, value functions are a fundamental component of RL as they aim to predict the most important thing for the agent: the expected return of its actions. In RL algorithms, value functions are used for two different purposes:

- First, to evaluate a given policy, which may be unknown, by estimating the expected return obtained following this policy.
- Second, to explicitly learn a strategy to control the agent, in which case the agent's policy will be derived from the learnt value function.

In both cases, the algorithms presented learn estimates of the value functions defined in the last section (see Equations 7 and 8).

3.1.1 Value Iteration

A value function estimate can be learnt with the *value iteration* technique: an iterative process of gathering experience with a given policy and improving the estimated value function. Value iteration turns the Bellman optimality equations (see Equation 13) into an update rule. That is, given a current value estimate V, we compute a better estimation based on the result of the Bellman equation:

$$V(s) \leftarrow \max_{a \in \mathbf{A}} \sum_{s' \in \mathbf{S}} \mathcal{T}(s' \mid s, a) \left[\mathcal{R}(s, a, s') + \gamma V(s')\right], \tag{15}$$

for each state $s \in \mathbf{S}$. The arrow indicates that we replace the current value $V(s)$, typically stored in a table with one value for each state, with the new one resulting from the calculation. In finite MDPs, executing this one time yields a new value function that is necessarily closer to the optimal value function. Thus, starting with an arbitrary initial value function (e.g., $V(s) = 0$ for all $s \in \mathbf{S}$), repeating the value iteration operation ensures that V converges towards V^*. In Figure 4, value iteration is executed on the simple RL environment previously introduced. This example shows how value iteration can learn the optimal value function quickly in a simple environment with a limited number of states.

Note that here we omitted the subscript π to V as it is often done when learning value functions. But V still measures the value of a given policy, which in this case is dictated by the Bellman equation. By taking the maximum value of all possible actions, the value estimates the return following the policy that always takes the action that goes to the state with the best expected return. This policy is said to be *greedy* with regards to the learnt value function.

An interesting property of such algorithms is the fact that they use the current estimate to compute a new one. This is referred to as *bootstrapping*. As we will see, this feature is at the core of many value estimation methods. Bootstrapping can greatly improve the convergence speed of these algorithms: in Figure 4, using the current estimate V allows each iteration to have more impact (depending on the order in which states are processed). But, this can also induce difficulties: in more complex settings, using imperfect estimates can sometimes produce even worse new estimates, thus compromising the learning process.

3.1.2 Monte Carlo Methods

A crucial drawback of the value iteration algorithm is that it assumes complete knowledge of the environment dynamics. Only because we know the probabilities of all transitions and their resulting reward, are we able to compute the expected value in the update rule 15. In most RL settings, the environment dynamics are unknown. Therefore, it is impossible to compute the exact value iteration update rule. What is often possible, however, is to run simulated experiments to sample experienced trajectories. These sampled trajectories can then be used to compute approximations of what we want to predict. For example, given a trajectory τ generated using the sampling policy, we can compute the return from any state $s \in \tau$ (according to Equation 6) and use the result as an estimation of the expected return from state s, i.e., $V(s)$. This is the core idea behind **Monte Carlo** methods: estimating the value function of a given policy by averaging sampled returns. This can be performed iteratively, by keeping a running estimate of the value function and updating it after each episode with:

$$V(s_t) \leftarrow V(s_t) + \alpha \left[G_t - V(s_t)\right], \tag{16}$$

for all $t < T$, with G_t the experienced return from state s_t and $\alpha \in]0, 1]$ a constant parameter called the **learning rate**. This update modifies the value of $V(s_t)$ by "moving it" towards the experienced return. The learning rate α controls the magnitude of this modification: $\alpha = 1$ would mean the new value of $V(s_t)$ is G_t, $\alpha < 1$ yields a new value between the old $V(s_t)$ and G_t. Using a learning rate is crucial as one sampled return is a poor approximation of the actual state-value function. Thus, repeating this update many times will allow to gradually move towards a better estimate of $V(s_t)$.

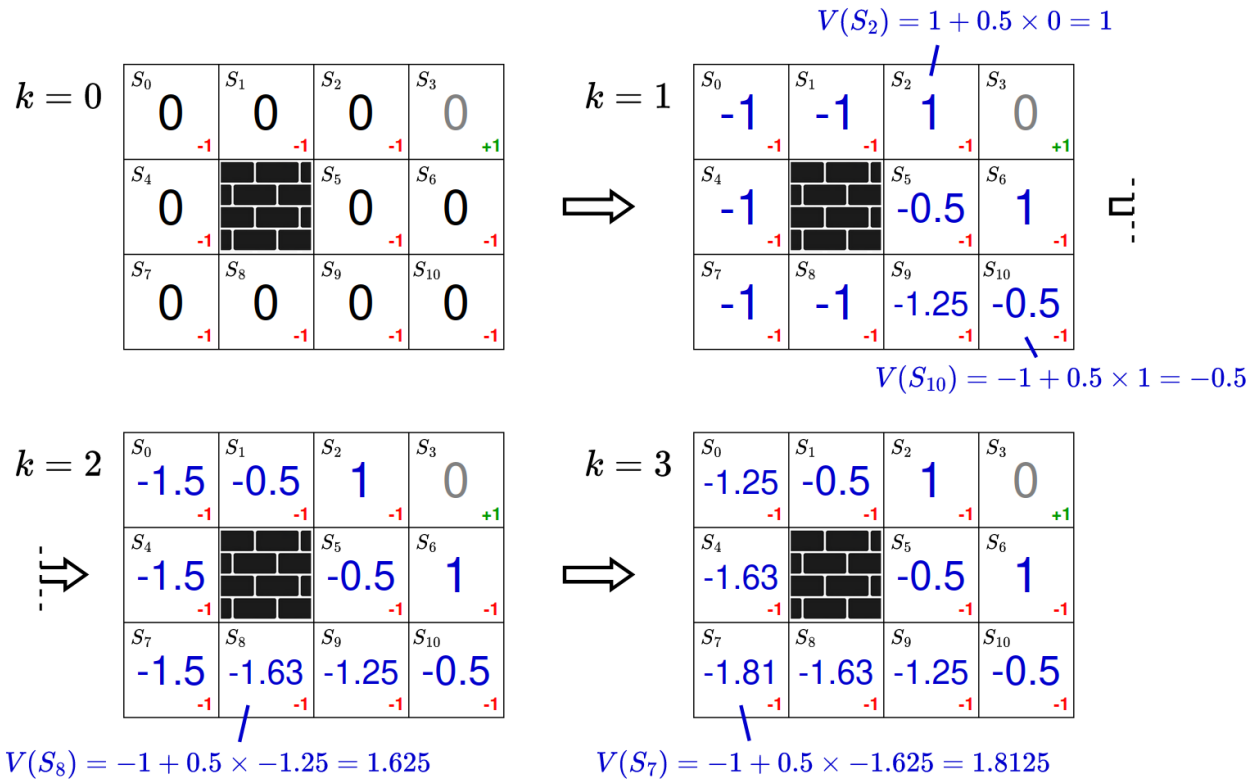


Figure 4: Value iteration algorithm executed on the example of Figure 3. We start with an initial value function $V(s) = 0$ for all $s \in \mathbf{S}$. Then, each iteration k applies the update rule of Equation 15, with $\gamma = 0.5$, with the result $V(S_t)$ shown in the corresponding cell. As state S_3 is terminal, its value is considered to be 0. In three iterations, we converge to the optimal value function. Note that, because we compute the new V for each state successively, the order of the states affects the result of each iteration and the number of iterations required for converging.

3.1.3 Temporal Difference Learning

Monte Carlo methods are limited as they require waiting for the end of each episode to compute the return of all states. In some environments, the episode might last for several hours or even never end, making this approach impractical. A solution for this is to use **Temporal-Difference** (TD) learning (Sutton, 1988). TD methods also learn an estimate of the value function, but using only a limited number of transitions. The simplest TD method, called *TD(0)*, updates the value after only one time step:

$$V(s_t) \leftarrow V(s_t) + \alpha \left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \right]. \tag{17}$$

The return in the Monte Carlo update rule (Equation 16) is replaced by its estimation using the current value function. The expression inside the brackets is often referred to as the *TD-error* or *Bellman error* as it is derived from the Bellman equation 13. It measures the difference between the current estimated value of s and the better estimate $r_{t+1} + \gamma V(s_{t+1})$, called the *TD-target*. In TD(0), the TD-target is based on a single transition that has been observed by the agent, but it can be extended to use the experience of multiple time steps (Sutton, 1988), with the *n-step TD-target* being:

$$\hat{G}_{t:t+n} = r_{t+1} + \gamma r_{t+2} + ... + \gamma^n V(s_{t+n}). \tag{18}$$

Using observed experience from multiple successive steps helps computing a better estimation of the actual return by reducing the impact of the bootstrapped value estimate.

3.1.4 SARSA and Q-learning

The TD learning approach can be applied for learning action-value functions. In the **SARSA** algorithm (Rummery & Niranjan, 1994), a tuple $\langle s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1} \rangle$ — giving its name to the algorithm — is used to compute the TD update rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]. \tag{19}$$

This requires the experience of two successive steps. The *Q-learning* algorithm (Watkins, 1989) uses only one transition, using the learnt action-value to select an action for the next step in the TD-target:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]. \tag{20}$$

This update rule uses the current estimate Q as an approximation of the optimal Q^* to compute the Bellman error. Thus, like in value iteration, Q-learning learns the action-value function of the *greedy* policy with respect to Q: $\pi(s) = \arg\max_a Q(s, a)$.

For Q-learning to effectively learn a good approximation of Q^*, the agent must visit all states of the environment enough times. Concretely, this means having a sampling policy that allows exploring all parts of the state space. One such exploration policy can be to select actions randomly, allowing the agent to try actions regardless of their current estimated value and hopefully explore the whole state space uniformly. But, in most settings, the vast majority of states and transitions are not actually valuable for solving the task. So a fully random exploration will spend a lot of time in irrelevant parts of the state space, slowing down the training. Instead, we might want to focus on the states and actions that have more potential for solving the task. This is a common dilemma in RL algorithms, where we need to balance a trade-off between *exploration* and *exploitation*:

- We want to explore different behaviours to better understand the environment and improve our estimations.
- But, at the same time, we want to exploit previously acquired knowledge to find the optimal strategy as fast as possible.

In the case of Q-learning, this exploration-exploitation trade-off is tackle with the *ϵ-greedy* policy. At each time step, ϵ-greedy decides between choosing an action randomly or following the greedy policy. It uses a parameter ϵ between 0 and 1 that determines the probability of choosing the action randomly. Typically, this parameter starts at 1 and decreases throughout training towards 0. This allows a progressive shift between random exploration at the start when the estimations are bad, and exploitation when estimations start getting better. While this is far from being a perfect exploration strategy, it offers a simple solution for exploration in Q-learning. The following algorithm describes Q-learning with ϵ-greedy:

Algorithm 1 Q-learning with ϵ-greedy policy

Initialize $Q(s, a)$ arbitrarily
Initialize $\epsilon = 1$
Initialize α (learning rate), γ (discount factor)
for each episode **do**
  Initialize state s_0
  for t=0,...,T **do**
    Execute ϵ-greedy policy:
      With probability ϵ, choose a random action a_t
      Otherwise, choose $a_t = \arg\max_a Q(s_t, a)$
    Perform action a_t, observe reward r_{t+1} and new state s_{t+1}
    $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$
    $s_t \leftarrow s_{t+1}$
    $\epsilon \leftarrow \max(\epsilon - 0.001, 0)$
  end for
end for

The original Q-learning algorithm represents the learnt Q function as a table that keeps track of the current value estimates. In this Q-table, rows correspond to states of the environment and columns to actions, with each cell (s, a) containing the current value $Q(s, a)$. Such "tabular" RL algorithms have a clear drawback: they are limited in the number of single values they can keep track of. To learn the optimal action-value function, Q-learning needs to try all possible state-action couples several times. But, this is impossible in rich environments with very large, if not continuous, state and action spaces. Thus, tabular value functions are limited to environments where they can realistically explore all transitions enough times to build good value estimates. To tackle continuous states and actions spaces, such algorithms must be adapted with specific techniques. One possibility is to discretise the spaces of states and actions, retrieving discrete spaces that can fit into a table but losing the fine-grained detail present in continuous spaces. The other option is to use function approximation techniques — e.g., linear function approximation or artificial neural networks —, representing the learnt value function as a parameterised function of states instead of a table (see Section 4). This allows for learning a function able to estimate the value of states that were never seen before, hopefully generalising from similar encountered states.

3.1.5 On-Policy vs. Off-Policy Learning

At this point, an important distinction between SARSA and Q-learning can be made. In SARSA, the update rule 19 depends on the policy used to generate the experience, which is referred to as the *behaviour* policy. This is because the TD-target is computed using the value of the action a_{t+1} selected by the behaviour policy. For this reason, SARSA is categorised as an **on-policy** algorithm. On the other hand, the update rule for Q-learning (20) does not depend on the behaviour policy as it uses the learnt action-value to select the action in the TD-target. Thus, Q-learning is categorised as **off-policy**.

Behind this distinction is an important difference between these two algorithms. Because SARSA follows the behaviour policy for computing the update, it actually learns the action-value associated with this behaviour policy. On the other hand, Q-learning computes the TD-target following the greedy policy with respect to the learnt action-value function. Thus, it learns the value associated with this policy, not with the behaviour policy. This illustrates that off-policy algorithms can learn values or policies based on data generated by other, totally separate processes. For example, Q-learning could learn the optimal action-value function based on experiences gathered by a random policy, by a another agent, or even by human experts. This particular feature allows using **experience replay** (L.-J. Lin, 1992), where, during training, experiences are stored in a memory, called the *replay buffer*, and are then reused any number of times in later updates. Updates are computed on samples of past experiences drawn randomly from the replay buffer. It also improves the sample efficiency of these algorithms, as each experience can be re-used several times throughout training.

3.2 Policy Gradients

3.2.1 Learning a Parameterised Policy Function

In Section 2.2, we said that an RL agent is primarily defined by its policy function π, which maps states to actions. We saw in the last section that value functions can be used to evaluate an unknown policy or to learn a policy by being greedy with respect to an action-value function. Another possible approach is to explicitly learn the policy function. This can be done by representing the policy function as a parameterised mathematical function of states which outputs either a probability over possible actions, in the case of a stochastic policy, or the action to perform, for a deterministic policy:

$$\pi(a \mid s, \theta) = Pr\{a_t = a \mid s_t = s, \theta_t = \theta\}, \tag{21}$$

or

$$\pi(s; \theta) = a. \tag{22}$$

In both cases, $\theta \in \mathbb{R}^{d^\theta}$ is the set of learnt parameters of the policy function. We also denote such policy π_θ, as the policy defined by parameters θ.

To maximise future rewards, the objective is to find the set of values for θ that produces the best sequences of actions. To do so, we can sample experiences from the environment and accordingly modify θ to improve the policy. This can be done using **gradient ascent**, which gradually modifies the parameters to maximise a given measure of performance $J(\theta)$:

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t), \tag{23}$$

where $\alpha \in]0, 1]$ is a small learning rate and $\nabla J(\theta_t)$ is the gradient of the objective $J(\theta_t)$ with regards to the policy parameters θ at time step t. It is a vector that shows in which direction in the parameter space to move θ to maximise the objective. Thus, Equation 23 replaces the current parameters θ_t with a new set of values θ_{t+1} that yields better values of J.

3.2.2 The Policy Gradient Theorem

In RL, the objective to maximise is the sum of future rewards. Thus, we can define the objective using the true value function of our policy: $J(\theta) = V_{\pi_\theta}(s_0)$, with s_0 being the initial state of the episode. Given this, for a discrete MDP and a given stochastic policy π_θ, we can devise the **policy gradient theorem** that yields:

$$\nabla J(\theta) = \sum_s \rho_{\pi_\theta}(s) \sum_a Q_{\pi_\theta}(s, a) \nabla \pi_\theta(a \mid s), \tag{24}$$

with Q_{π_θ} the action-value function of policy π_θ and $\rho_{\pi_\theta}(s)$ the probability of experiencing state s when following policy π_θ.

If policy π_θ is followed, then the theorem can be formulated with an expected value:

$$\nabla J(\theta) = \mathbb{E}_{s\sim\rho_{\pi_\theta}}\left[\sum_a Q_{\pi_\theta}(s,a)\nabla\pi_\theta(a \mid s)\right] \tag{25}$$

$$= \mathbb{E}_{s\sim\rho_{\pi_\theta}}\left[\sum_a \pi_\theta(a \mid s)Q_{\pi_\theta}(s,a)\frac{\nabla\pi_\theta(a \mid s)}{\pi_\theta(a \mid s)}\right]$$

$$= \mathbb{E}_{s\sim\rho_{\pi_\theta},a\sim\pi_\theta}\left[Q_{\pi_\theta}(s,a)\frac{\nabla\pi_\theta(a \mid s)}{\pi_\theta(a \mid s)}\right]. \tag{26}$$

The expected value in Equation 25 is obtained by sampling $s \sim \rho_{\pi_\theta}$. Likewise, in Equation 26, the sum is absorbed in the expectation by sampling $a \sim \pi_\theta$. Because $Q_{\pi_\theta}(s,a) = \mathbb{E}_{\pi_\theta}[G_{s,a}|s,a]$, by definition, we can rewrite the policy gradient as:

$$\nabla J(\theta) = \mathbb{E}_{\pi_\theta}\left[G_{s,a}\frac{\nabla\pi_\theta(a \mid s)}{\pi_\theta(a \mid s)}\right]. \tag{27}$$

The algorithm REINFORCE (Williams, 1992) approximates this expectation by sampling many trajectories, updating the parameters θ with:

$$\theta_{t+1} \leftarrow \theta_t + \alpha G_t\frac{\nabla\pi_\theta(a_t \mid s_t)}{\pi_\theta(a_t \mid s_t)}, \tag{28}$$

for any state s_t and action a_t from which the agent has experienced the return G_t. In this update, the gradient of the policy indicates the direction, in parameter space, that most increases the probability of choosing action a_t. By taking the product of this gradient with the return G_t, this update increases the probability of actions depending on the returns they generate. Positive returns will result in an increase in the corresponding action probabilities, with the increase being proportional to the return, and conversely for negative returns. Then, dividing by the probability of the action ensures that actions that are selected more frequently are not over-estimated by getting more updates of the same magnitude.

3.2.3 The Actor-Critic Architecture

As it uses the complete return G_t, REINFORCE is a Monte Carlo method for learning a parameterised policy function. Like the Monte Carlo approach presented in the last section, REINFORCE requires waiting for the end of the episode to update the policy. But, again, we can use the principle of TD learning (see Section 3.1.3) to solve this issue, by replacing the return by its estimation with a value function:

$$\theta_{t+1} \leftarrow \theta_t + \alpha\left(r_{t+1} + \gamma V(s_{t+1})\right)\frac{\nabla\pi_\theta(a_t \mid s_t)}{\pi_\theta(a_t \mid s_t)}, \tag{29}$$

or,

$$\theta_{t+1} \leftarrow \theta_t + \alpha Q(s_t,a_t)\frac{\nabla\pi_\theta(a_t \mid s_t)}{\pi_\theta(a_t \mid s_t)}, \tag{30}$$

where V and Q would be learnt value functions that estimate the true value functions of the current policy π. This kind of approach is referred to as **actor-critic** methods (Barto et al., 1983; Sutton, 1984), where the policy is the actor who learns to select actions, and the value function is the critic who evaluates the policy's actions. In this case, the value function is used only to guide the learning process of the policy.

Actor-critic architectures provide many algorithmic advantages. One example of this is the possibility of adapting the Q-learning algorithm to work with continuous actions. Continuous action spaces are impossible to handle with classical Q-learning, as finding a greedy policy with respect to a continuous action-value function — which is required for the update — would be an optimisation problem in itself. But, this can be solved using an actor-critic architecture. By extending the policy gradient theorem defined in Equation 25 to continuous actions and a deterministic policy, we can find that

$$\nabla J(\theta) = \mathbb{E}_{s\sim\rho_{\pi_\theta}}\left[\nabla_\theta\pi_\theta(s)\nabla_a Q_\varphi(s,a) \mid a = \pi_\theta(s)\right], \tag{31}$$

given policy π_θ and an action-value function Q_φ parameterised by $\varphi \in \mathbb{R}^{d^\varphi}$. This the **deterministic policy gradient theorem** (DPG), as defined by Silver et al. (2014), which gives the following update for the parameters θ:

$$\theta_{t+1} \leftarrow \theta_t + \alpha_\theta\nabla_\theta\pi_\theta(s_t)\nabla_a Q_\varphi(s_t,a_t). \tag{32}$$

The parameters of the action-value function φ are updated as well to minimise the Bellman error:

$$\delta_t = r_t + \gamma Q_\varphi(s_{t+1},a_{t+1}) - Q_\varphi(s_t,a_t) \tag{33}$$

$$\varphi_{t+1} \leftarrow \varphi_t + \alpha_\varphi\delta_t\nabla_\varphi Q_\varphi(s_t,a_t). \tag{34}$$

Intuitively, DPG extends Q-learning for continuous actions, where the ϵ-greedy policy is replaced by a learnt policy trained to select the actions that maximise Q_φ.

3.3 Planning with Models

The two previous sections presented algorithms that learn a policy, a value, or both. These methods are often called **model-free** because they do not use a model of the environment dynamics. That is in contrast with **model-based** methods that rely primarily on a model to learn an efficient strategy. Having a model that predicts future outcomes allows to simulate experiences and use these to devise a strategy or improve other learnt elements.

3.3.1 Learning the Model

Learning the model is a supervised task where the agent learns to approximate the environment dynamics using its own experienced transitions as training examples. The goal is to learn a good approximation $\widehat{p}$ to use for planning. As $\widehat{p}$ learns two distinct elements, the transition probability $\mathcal{T}$ and the reward function $\mathcal{R}$, the prediction task is often separated in two as well, with $\widehat{\mathcal{T}}$ and $\widehat{\mathcal{R}}$.

The simplest way to learn these functions is by learning a *tabular model* (Sutton, 1991). In a discrete MDP, a table can store the number of occurrences of all transitions $n(s, a, s')$ and use it to output a probability for each possible next state:

$$\widehat{\mathcal{T}}(s' \mid s, a) = \frac{n(s, a, s')}{\sum_{s" \in \mathbf{S}} n(s, a, s")}. \tag{35}$$

Similarly, for the reward, we can store the rewards obtained in each transition $r(s, a, s')$ and use them to approximate the expected reward of each transition by computing a mean over sampled transitions:

$$\widehat{\mathcal{R}}(s, a, s') = \frac{1}{N} \sum_{k=0}^{N} r_k(s, a, s'), \tag{36}$$

with $N = n(s, a, s')$. This tabular solution can be effective in small discrete MDPs. However, it does not scale well to environments with many possible transitions.

To tackle this issue, another approach is to define the model as a parametric function $\widehat{p}(s, a, \psi)$ with parameters $\psi \in \mathbb{R}^{d^\psi}$. Like for value and policy functions, this parameterised function can take many different forms: linear regression (Sutton et al., 2008), random forest (Hester & Stone, 2012), neural networks (Narendra & Parthasarathy, 1990; Oh et al., 2015).

3.3.2 Using the Model

We introduced models as a tool to simulate experience. This can serve different purposes. One approach is to use the model to learn better approximations of value and policy functions. Instead of gathering experiences in the environment, a model can be queried to simulate transitions. These simulated transitions can then be used for computing the learning objectives of any algorithm. For example, in Dyna (Sutton, 1991), a tabular model is used to perform many Q-learning updates between each environment step. This improves sample efficiency by increasing the number of updates for each call to the environment. Other learning algorithms require the environment dynamics to perform their update, like value iteration (see Section 3.1.1) or guided policy search (Levine & Koltun, 2013). In such cases, learning a model enables using these algorithms in complex environments with unknown dynamics (Abbeel et al., 2006; Levine & Abbeel, 2014).

A second approach uses the model for planning future sequences of states and actions, and ultimately choose the path leading to the best return. This can be done in an exhaustive manner, by simulating all possible paths until the completion of the episode. Then, the value of each observed state can be computed as a mean of all returns obtained from this state (Tesauro & Galperin, 1996), similarly to Monte Carlo methods (see Section 3.1.2). Planning in this way is particularly effective, especially in environments where a perfect model is given like Chess or Go. But, it requires a great amount of computation to simulate all trajectories and evaluate each state and action. Doing an exhaustive search of all possible outcomes is unrealistic in environments with a large state space. In the game of Go, the number of possible states is beyond any computing limitation. To handle this, the Monte Carlo Tree Search algorithm (MCTS; Coulom, 2006) greatly optimises the search by parallelising the computation of different paths of actions, excluding states that are evaluated as bad, and learning both an action-value and a policy function. This efficient planning strategy was crucial for building human-level algorithms in complicated games such as Go (Coulom, 2006; Finnsson & Björnsson, 2008; Silver et al., 2016; Schrittwieser et al., 2019).

Overall, model-based methods have many advantages. Intuitively, learning the environment dynamics seems logical, as the model learns general knowledge about the environment that should be applicable to all parts of the environment, which is not necessarily the case for value and policy functions. Model-based approaches are often more sample efficient than model-free ones. By using the model's prediction to train policies and values, they require less experience

in the real environment to converge to a good strategy. However, models still have their limitations. Learning the model is not straightforward in realistic settings where the environment is complex and dynamic. Also, relying on planning for selecting actions is very computationally intensive, which makes it tough to implement in real-time applications. For a recent review of literature on model-based RL, see Moerland et al. (2023).

4 Model-Free Deep Reinforcement Learning

As in other branches of machine learning, deep learning has been a revolutionary technique for RL. Using deep neural networks for function approximation has enabled extending existing RL algorithms to operate in much more complex settings and has facilitated the development of new approaches for learning models, policies, and value functions. In this section, we will present the foundational works in model-free deep RL published in the last decade, which form the basis for the majority of approaches using deep RL.

4.1 Value-Based Methods

4.1.1 Deep Q-Learning

In Section 3.1.4, we presented the Q-learning algorithm for learning a control policy using only an action-value function. We introduced the main limitations of this algorithm being scalability and generalisation. These make Q-learning difficult to use in settings where the number of possible states is large or infinite.

The **Deep Q-Network** (DQN) algorithm, proposed by Mnih et al. (2013), tackle these issues by using a deep neural network in place of the Q-table. This allows for successfully learning an action-value function from visual data. To do so, it models the action-value as a convolutional neural network (CNN; LeCun et al., 1989) that takes as input frames from a video game and outputs the values of possible actions, as illustrated in Figure 5a. Note that, while this first iteration of the DQN architecture uses a CNN, the term DQN is now used to describe any deep neural network architecture that learns an action-value function.

The DQN is learnt completely end-to-end from the Q-learning objective. Concretely, the parameters of the network θ are optimised to minimise the following squared Bellman error:

$$L(\theta) = \left(r_{t+1} + \gamma \max_a Q_{\theta^-}(s_{t+1}, a) - Q_\theta(s_t, a_t)\right)^2, \tag{37}$$

for any transition $\langle s_t, a_t, r_{t+1}, s_{t+1}\rangle$ and with the discount factor γ. Notice the TD-target is computed using another network Q_{θ^-}, named the *target network*. The parameters θ^- are initialised as a copy of the main parameters θ. Then, the target is fixed for a pre-defined number of training iterations, after which it is copied from the main network again, and so on. Using a slightly older, fixed version of Q_θ helps to stabilise training by giving a fixed target to aim for when minimising the Bellman error, instead of having it modified after each training iteration (Mnih et al., 2015).

Another important feature of the DQN is its use of experience replay (L.-J. Lin, 1992). Because deep Q-learning is off-policy, it can capitalise on all previously acquired experiences to train its deep neural network. The deep Q-learning algorithm thus alternates between acquiring experiences by interacting with the environment (typically using the ϵ-greedy policy), storing each experience in the replay buffer $\mathcal{B}$, and later training the DQN on a batch of samples randomly drawn from $\mathcal{B}$. Therefore, the DQN is trained to minimise the Mean Squared Bellman Error (MSBE) on the full batch:

$$L(\theta) = \mathbb{E}_{\langle s,a,r,s'\rangle\sim\mathcal{B}}\left[\left(r + \gamma \max_{a'} Q_{\theta^-}(s', a') - Q_\theta(s, a)\right)^2\right]. \tag{38}$$

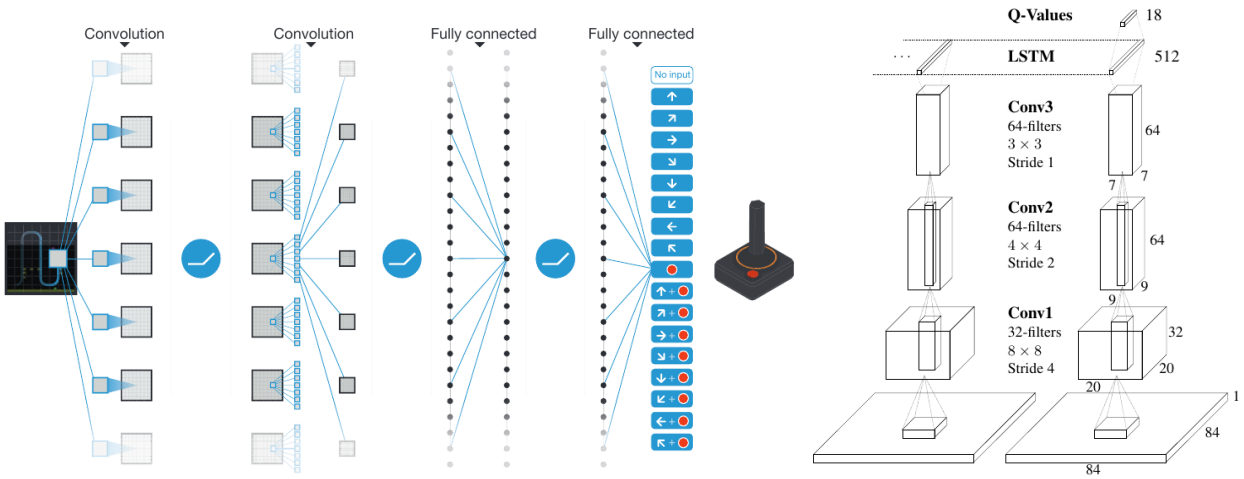
This batched loss is crucial for ensuring efficient training of a deep neural network. Plus, experience replay allows reusing each gathered experience multiple times and having more diverse batches, thus improving sample efficiency and avoiding overfitting the current policy.

4.1.2 DQN Extensions

Following the introduction of the DQN, many works focused on improving this architecture using both old and new techniques. In their paper, Hessel et al. (2018) summarise the most important extensions and combine them into their method called *Rainbow*. Some of these improvements concern the core functioning of the Q-learning algorithm:

- *Double Q-learning* (van Hasselt, 2010) modifies Equation 38 to decouple the selection of the action to its evaluation when computing the Bellman error:

$$L(\theta) = \left(r_{t+1} + \gamma Q_{\theta^-}(s_{t+1}, \arg\max_{a'} Q_\theta(s_{t+1}, a')) - Q_\theta(s_t, a_t)\right)^2. \tag{39}$$



(a) Deep Q-Network (b) Deep Recurrent Q-Network

Figure 5: DQN architectures. (a) The original DQN (Mnih et al., 2015), with a series of convolution layers that analyse the input frame, followed by fully connected layers that output the value of each action. (b) Recurrent version of a DQN (Hausknecht & Stone, 2015), that adds an LSTM unit after the convolutions to allow some information to carry to future steps.

In the TD-target, the action is selected by the main network Q_θ and evaluated by the target network Q_{θ^-}. van Hasselt et al. (2016) apply this idea to the DQN to improve its performance.

- The *Dueling DQN* (Wang et al., 2016) proposes to decompose the action-value of an action $Q(s, a)$ into two separate predictions: the value of the current state $V(s)$ and the *advantage* of the action defined as $A(s, a) = Q(s, a) - V(s)$. They use a neural network architecture with two outputs, the state-value and advantage, which are then combined into the action-value. This allows to learn a state-value and an action-value function in a single learning process with a deep Q-learning algorithm.
- *Distributional value learning* (Bellemare et al., 2017) proposes to learn a probability distribution over possible future returns. Doing this allows to better account for multimodality in value distributions and generally improves learning stability.

Other techniques focus on algorithmic and architectural improvements to DQN:

- *Prioritised experience replay*, proposed by Schaul et al. (2016), improves on experience replay by sampling experiences more often if they lead to highly incorrect value predictions (i.e., with high Bellman errors) during training. This allows prioritising samples that the network fails to evaluate well, thus greatly improving sample efficiency.
- *Noisy DQN* (Fortunato et al., 2018) tackles the problem of exploration by adding noise to the parameters of the DQN. Note that Plappert et al. (2018) concurrently proposed a similar approach working with other deep RL algorithms.

Rainbow shows that combining all these extensions greatly improves the performance and sample efficiency of DQN in a large variety of video games (Hessel et al., 2018).

4.1.3 Deep Recurrent Q-Network

A remaining limitation of DQN is its lack of memory of previous events. This becomes a problem in partially observable environments where the agent does not get the full state of the environment, but rather a local observation describing its surroundings. In such settings, a memory-equipped agent could explore its surroundings to build a better understanding of the current state of the environment. To achieve this, Hausknecht and Stone (2015) augment the DQN architecture with an *Long Short-Term Memory* (LSTM; Hochreiter and Schmidhuber, 1997) recurrent neural network to learn to memorise important information for future steps (see Figure 5b). This makes agents more robust to losses in their observations. This approach has been combined with distributed training by Kapturowski et al. (2019), largely surpassing previous works on some Atari games. Deep recurrent Q-Networks have also been widely adopted in partially observable environments.

4.2 Policy-Based Methods

4.2.1 Deep Deterministic Policy Gradients

Following the introduction of the DQN, similar ideas were implemented to extend actor-critics with deep learning. The **Deep Deterministic Policy Gradient** (DDPG; Lillicrap et al., 2015) was proposed to extend the DQN to work with continuous actions. Like in the original DPG (see Section 3.2.3), a deterministic policy function π_θ is learnt to select actions in a continuous space and an action-value function Q_ϕ is used to evaluate the selected actions. Here, both of these are modelled using deep neural networks. As in Double Q-learning (see Section 4.1.2), both the policy and value functions are doubled with target functions π_{θ^-} and Q_{ϕ^-}, respectively. The targets are used to compute the TD-target for computing the MSBE loss:

$$L(\Theta) = \mathbb{E}_{\langle s,a,r,s'\rangle\sim\mathcal{B}}\left[\left(r + \gamma Q_{\phi^-}(s', \pi_{\theta^-}(s')) - Q_\phi(s,a)\right)^2\right]. \tag{40}$$

The target parameters $\Theta^- = \{\theta^-, \phi^-\}$ are initialised as copies of the main parameters Θ and later slowly moved towards the main networks with a "soft" update: $\Theta^- \leftarrow \tau\Theta^- + (1-\tau)\Theta$, with $0 < \tau \ll 1$.

Fujimoto et al. (2018) pointed out that DDPG, as all Q-learning methods, suffers from an overestimation of action-values induced by using a bad value estimate for computing the optimisation objective. While Double Q-learning helps address this issue, it does not completely solve it. They propose an extension of DDPG, named *Twin Delayed DDPG* (TD3), that better mitigates the overestimation problem with three tricks:

- First (*Twin*), TD3 learns two action-value functions instead of one and uses the smaller values of the two when computing the TD-target.
- Second (*Delayed*), the policy function is trained less often than the action-value functions (one policy update for every two value updates) to make the policy less prone to value errors.
- Third, the value estimates are smoothed by adding a small clipped noise on the target policy actions when computing the TD-target. This prevents having high spikes of value on some particular actions that the deterministic policy would then overfit to.

Therefore, TD3 computes the TD-target by taking the minimum value given by the twin DQNs and adding the clipped noise to the policy's action:

$$y(r,s') = r + \gamma \min_{i=1,2} Q_{\phi_i^-}(s', \pi_{\theta^-}(s') + \epsilon), \tag{41}$$

$$\epsilon \sim \text{clip}(\mathcal{N}(0,\sigma), -c, c), \tag{42}$$

for transition $\langle s,a,r,s'\rangle$, with hyperparameters σ and c controlling the width of the smoothing noise. The twin DQNs are both trained to minimise their MSBE:

$$L(\phi_i) = \mathbb{E}_{\langle s,a,r,s'\rangle\sim\mathcal{B}}\left[\left(y(r,s') - Q_{\phi_i}(s,a)\right)^2\right], \text{ with } i \in \{1,2\}. \tag{43}$$

These modifications are shown to improve the stability and performance of DDPG.

4.2.2 Trust Region Policy Updates

In Section 3.2.2, we presented the policy gradient method used in most policy-based algorithms, which aims at maximising the following objective for learning a stochastic policy:

$$L^{PG}(\theta) = \mathbb{E}_\pi\left[\log \pi_\theta(a|s) Q_{\pi_\theta}(s,a)\right].^2 \tag{44}$$

This objective has two important limitations. First, the size of the learning step is difficult to define. In the context of learning a policy that generates the data it is trained on, a too big learning step could be catastrophic if it produces a bad policy. In supervised learning, a bad learning step can be recovered later because the training data is still good. Here, a bad policy would change the distribution of samples and could then be very hard to recover from. Second, this objective theoretically limits the algorithm to use each gathered experience only one time — Schulman et al. (2017) shows doing multiple updates damages performance —, which makes policy gradient algorithms less sample efficient.

Schulman et al. (2015) propose to modify the policy gradient objective to tackle these issues. The objective is replaced by the following surrogate objective:

$$L^{surr}(\theta) = \mathbb{E}_{\pi_\theta}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} A_{\pi_\theta}(s,a)\right], \tag{45}$$

[2] If we compute the gradient of this objective, we find the formulation of Equation 26 as, by chain rule, $\nabla_x \log f(x) = \frac{\nabla_x f(x)}{f(x)}$.

where $\pi_{\theta_{old}}$ is another policy and A_{π_θ} is the advantage function defined as $A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s)$ (Baird, 1995). Using the advantage is equivalent to the action-value but reduces the variance of the value estimate. This objective is shown to be a good approximation of the policy gradient one, only if $\pi_{\theta_{old}}$ is close to π_θ. Thus, for this to work, a constraint must be added to ensure that π_θ does not diverge too much from the old version:

$$\mathbb{E}_{\pi_\theta}\left[KL(\pi_{\theta_{old}}(\cdot|s), \pi_\theta(\cdot|s))\right] \leq \delta, \tag{46}$$

with KL measuring the Kullback-Leibler divergence between the two distributions and δ a hyperparameter for controlling the amount of allowed divergence. Maximising L^{surr} while respecting the constraint of 46 ensures that the update stays in a "trust region". The resulting **Trust Region Policy Optimisation** (TRPO) algorithm can do multiple updates on one sampled experience, with $\pi_{\theta_{old}}$ being the policy used when gathering the experience, to improve π_θ as much as possible for each interaction with the environment.

While the approach TRPO has great advantages, its implementation is difficult, mainly because of the constraint on KL divergence requiring second-order optimisation. To simplify this, **Proximal Policy Optimisation** (PPO; Schulman et al., 2017) removes the constraint and instead clips the objective to limit the magnitude of the updates. The new, clipped objective is defined as follows:

$$L^{clip}(\theta) = \mathbb{E}_{\pi_\theta}\left[\min(r(\theta)A_{\pi_\theta}(s,a), \text{clip}(r(\theta), 1-\epsilon, 1+\epsilon)A_{\pi_\theta}(s,a)\right], \tag{47}$$

with $r(\theta)$ the probability ratio from TRPO, $r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}$, and ϵ a hyperparameter typically close to 0. This objective first clips the ratio to be close to 1 to ensure that the new policy does not diverge too much from the old one. Then, it takes the minimum of the clipped and unclipped objectives to make a lower bound of the unclipped objective. This objective is simpler to compute and optimise, making PPO simpler to implement. The full objective maximised by PPO is the following:

$$L^{PPO}(\theta, \phi) = \mathbb{E}_{\pi_\theta}\left[L^{clip}(\theta) - c_1 L^{VF}(\phi) + c_2 H(\pi_\theta)\right] \tag{48}$$

where c_1 and c_2 are coefficients for the two secondary losses. L^{VF} is the MSBE loss for learning the value function used to compute the advantage in (47), using a technique called Generalised Advantage Estimation (Schulman et al., 2016). The third term uses the entropy of the policy, defined as $H(\pi) = \mathbb{E}_\pi[-\log \pi(a|s)]$, to induce exploration. Maximising entropy means increasing the randomness of the policy. This bonus ensures random exploration of new policies.

PPO is considered one of, if not the best existing deep RL algorithm. It has been deeply studied and improved in various ways since its first release. Many implementation tricks have been identified to be crucial to the performance of the algorithm (Henderson et al., 2018; Engstrom et al., 2020).

4.2.3 Soft Actor-Critic

Finally, we present the Soft Actor-Critic (SAC) algorithm (Haarnoja et al., 2018) that tackles the poor sample efficiency of on-policy algorithms and the brittleness of off-policy algorithms with regard to their hyperparameters. To do so, it builds an off-policy actor-critic algorithm with a stochastic policy and a "soft" action-value function. The policy is trained on samples drawn from the replay memory $\mathcal{B}$ to maximise the expected future rewards and an entropy bonus for exploration, as in PPO:

$$L^\pi(\theta) = \mathbb{E}_{s\sim\mathcal{B}, a\sim\pi_\theta}\left[Q_\phi(s,a) + \alpha H(\pi_\theta)\right], \tag{49}$$

with $alpha$ controlling the importance of the entropy bonus. As in DDPG, the action is sampled from the policy and evaluated by the value function. Maximising this objective operates a trade-off between generating actions with high action-values and inducing diverse behaviours.

To learn the action-value function, the authors take inspiration from soft Q-learning (Haarnoja et al., 2017), where entropy maximisation is included in the value learning algorithm. The soft action-value is trained to minimise the soft MSBE for any transition drawn from the buffer:

$$L^Q(\phi) = \mathbb{E}_{\langle s,a,r,s'\rangle\sim\mathcal{B}}\left[\frac{1}{2}\left(Q_\phi(s_t,a_t) - y^{soft}(r,s')\right)^2\right], \tag{50}$$

where

$$y^{soft}(r,s') = r + \gamma\mathbb{E}_{a'\sim\pi_\theta}[Q_\phi(s',a') - \alpha\log\pi_\theta(a'|s')]. \tag{51}$$

The target y^{soft} also has a bonus on the entropy of the policy, which makes the action-value function favour actions where the policy is uncertain. This is shown to greatly improve the stability and robustness of their algorithm.

5 Going Further

In this article, we have presented an introduction to the field and techniques of deep reinforcement learning. While not an exhaustive course or review of existing works, it should provide the required material to understand the motivations, concepts, and algorithms used in RL research and applications. As the RL approach gets more and more accepted as a central component of advanced, interactive AI systems, the domain of RL research spreads and develops new directions for improving the capabilities of agents. Here, we briefly introduce a few of these fields that present interesting potential for future research.

Model-based deep RL has immensely benefitted from deep learning to learn models of complex environments, which may be crucial for building agents capable of reasoning in real-world settings. So called *"world models"* learn latent representations of the environment with supervised autoencoders which can then be used to "imagine" future outcomes — i.e., plan in the latent space — (Ha & Schmidhuber, 2018; Hafner et al., 2018, 2020), helping for driving exploration (Sekar et al., 2020; Mendonca et al., 2021; E. S. Hu et al., 2023) and solving tasks requiring long sequences of actions (Hafner et al., 2022). Like previous model-based approaches, world models are much more sample-efficient than model-free baselines (Samsami et al., 2024; Hafner et al., 2025). The added deep latent representation learning allows tackling high-dimensional and diverse environments (Hansen et al., 2024; Hafner et al., 2025). The combination of these two advantages makes these approaches great candidates for tackling robotic tasks.

Hierarchical RL decomposes the policy learning into multiple levels of actions (Pateria et al., 2021). Low-level actions are the ones performed by the agent at each time-step of the environment, while more high-level actions refer to macro-actions typically requiring multiple steps to perform (Sutton et al., 1999; Amato et al., 2019). This allows disentangling the multiple levels of reasoning in the action-selection process, with a high-level policy focused on selecting sub-goals required to complete the task and a low-level policy — or multiple ones — for completing the sub-goals (Tessler et al., 2017; Hafner et al., 2022; J. Hu et al., 2024).

Intrinsic motivation takes inspiration from real biological systems that are not only motivated by solving a given task, but also, and perhaps primarily, intrinsically pushed to try new things and explore their environment (Berlyne, 1966). In RL, an intrinsically motivated agent computes an additional reward based on its observations and internal state (Oudeyer & Kaplan, 2007). Maximising this reward signal, in addition to the reward given by the environment, allows inducing some desired behaviour in the agent's policy. A typical example is curiosity which can be induced by rewarding the agent for finding "novel" states (Schmidhuber, 1991), with novelty being any measure of how new a state is compared to previously experienced states (Pathak et al., 2017; Burda et al., 2019). Such active exploration approaches are crucial to solve hard exploration problems (Badia et al., 2020) and may be central to developing open-ended learning algorithms (Colas et al., 2022; Jiang et al., 2023).

Multi-agent RL is a large field focused on applying RL in environments with multiple agents interacting and learning (Wooldridge, 2009). Having multiple agents makes all problems of RL even more challenging and introduces new ones such as modelling the reasoning of other agents (Albrecht & Stone, 2018) or communicating to share knowledge and needs (C. Zhu et al., 2024). Studying these problems is crucial for allowing RL agents to handle social interactions which are such an important component of human intelligence. While one solution would be to stick to single-agent RL, considering other agents as parts of the environment, multi-agent RL research has proposed many ways of adapting classical RL techniques to factor in the specificities of multi-agent systems (Albrecht et al., 2024).

Multi-objective RL tackles the important issue of optimising multiple separate objectives at the same time, which is often required in practical applications (Roijers et al., 2013; Hayes et al., 2022). Classical RL approaches would combined these multiple reward signals with a linear transformation to form a single signal to maximise. But this inevitably loses some information about the different objectives which can be detrimental, especially when these objectives are conflicting. Research on multi-objective RL develop specific approaches to learn policies that compromise between objectives (Abels et al., 2019; Rădulescu et al., 2019; Yang et al., 2019; Xu et al., 2020).

Episodic RL takes inspiration from humans being able to recall specific past events to better handle a related situation (Tulving, 2002). In the classical RL approach, knowledge of the past is embedded in the learnt elements, but there is no explicit memory process allowing the agent to recall experience acquired in previous episodes to inform its present strategy. Episodic RL fixes this to improve the sample efficiency of deep RL algorithms (Z. Lin et al., 2018; G. Zhu et al., 2020; Freire et al., 2024).

Anyone planning on implementing RL algorithms should summon their patience and determination. RL is famously hard to implement successfully thanks to the numerous sources of variance and uncertainty that make debugging tedious. However, there are many resources online to understand the algorithms, implement them, and debug[3]. One particularly valuable is Andy Jones's guide to debugging RL, which everyone should read and implement in their practices.

[3]Summarized here: github.com/andyljones/reinforcement-learning-discord-wiki/wiki.

References

Abbeel, P., Coates, A., Quigley, M., & Ng, A. (2006). *An application of reinforcement learning to aerobatic helicopter flight* (B. Schölkopf, J. Platt, & T. Hoffman, Eds.).

Abels, A., Roijers, D., Lenaerts, T., Nowé, A., & Steckelmacher, D. (2019). *Dynamic weights in multi-objective deep reinforcement learning*. In K. Chaudhuri & R. Salakhutdinov (Eds.), *Proceedings of the 36th international conference on machine learning* (pp. 11–20, Vol. 97). PMLR.

Albrecht, S. V., Christianos, F., & Schäfer, L. (2024). *Multi-agent reinforcement learning: Foundations and modern approaches*. MIT Press.

Albrecht, S. V., & Stone, P. (2018). *Autonomous agents modelling other agents: A comprehensive survey and open problems*. *Artificial Intelligence*, *258*, 66–95.

Amato, C., Konidaris, G., Kaelbling, L. P., & How, J. P. (2019). *Modeling and planning with macro-actions in decentralized pomdps*. *Journal of Artificial Intelligence Research*, *64*, 817–859.

Badia, A. P., Piot, B., Kapturowski, S., Sprechmann, P., Vitvitskyi, A., Guo, Z. D., & Blundell, C. (2020). *Agent57: Outperforming the Atari human benchmark*. In H. D. III & A. Singh (Eds.), *Proceedings of the 37th international conference on machine learning* (pp. 507–517, Vol. 119). PMLR.

Bagnell, J., & Schneider, J. (2001). *Autonomous helicopter control using reinforcement learning policy search methods*. *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, *2*, 1615–1620 vol.2.

Baird, L. (1995). *Residual algorithms: Reinforcement learning with function approximation*. *Machine Learning Proceedings 1995*, 30–37.

Barto, A. G., Sutton, R. S., & Anderson, C. W. (1983). *Neuronlike adaptive elements that can solve difficult learning control problems*. *IEEE Transactions on Systems, Man, and Cybernetics*, *SMC-13*(5), 834–846.

Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2013). *The arcade learning environment: An evaluation platform for general agents*. *Journal of Artificial Intelligence Research*, *47*, 253–279.

Bellemare, M. G., Dabney, W., & Munos, R. (2017). *A distributional perspective on reinforcement learning*. In D. Precup & Y. W. Teh (Eds.), *Proceedings of the 34th international conference on machine learning* (pp. 449–458, Vol. 70). PMLR.

Bellman, R. (1957). *A markovian decision process*. *Journal of Mathematics and Mechanics*, *6*(5), 679–684.

Bellman, R. (1966). *Dynamic programming*. *Science*, *153*(3731), 34–37.

Bengio, Y., Louradour, J., Collobert, R., & Weston, J. (2009). *Curriculum learning*. *Proceedings of the 26th Annual International Conference on Machine Learning*.

Berlyne, D. E. (1966). *Curiosity and exploration: Animals spend much of their time seeking stimuli whose significance raises problems for psychology.* *Science*, *153*(3731), 25–33.

Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer New York, NY.

Brembs, B. (2010). *Towards a scientific concept of free will as a biological trait: Spontaneous actions and decision-making in invertebrates*. *Proceedings of the Royal Society B: Biological Sciences*, *278*(1707), 930–939.

Burda, Y., Edwards, H., Storkey, A., & Klimov, O. (2019). *Exploration by random network distillation*. *7th International Conference on Learning Representations*.

Busoniu, L., Babuska, R., De Schutter, B., & Ernst, D. (2017). *Reinforcement learning and dynamic programming using function approximators*. CRC press.

Chen, L., Wu, P., Chitta, K., Jaeger, B., Geiger, A., & Li, H. (2023). *End-to-end autonomous driving: Challenges and frontiers* [arxiv:2306.16927].

Christiano, P. F., Leike, J., Brown, T., Martic, M., Legg, S., & Amodei, D. (2017). *Deep reinforcement learning from human preferences*. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, & R. Garnett (Eds.), *Advances in neural information processing systems* (Vol. 30). Curran Associates, Inc.

Colas, C., Karch, T., Sigaud, O., & Oudeyer, P.-Y. (2022). *Autotelic agents with intrinsically motivated goal-conditioned reinforcement learning: A short survey*. *Journal of Artificial Intelligence Research*, *74*, 1159–1199.

Coulom, R. (2006). *Efficient selectivity and backup operators in monte-carlo tree search*. In H. J. van den Herik, P. Ciancarini, & H. H. L. M. (Donkers (Eds.), *Computers and games* (pp. 72–83). Springer Berlin Heidelberg.

Eiben, A. E., & Smith, J. E. (2003). *Introduction to evolutionary computing*. Springer Berlin Heidelberg.

Engstrom, L., Ilyas, A., Santurkar, S., Tsipras, D., Janoos, F., Rudolph, L., & Madry, A. (2020). *Implementation matters in deep policy gradients: A case study on ppo and trpo*. *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*.

Finnsson, H., & Björnsson, Y. (2008). *Simulation-based approach to general game playing*. *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 1*, 259–264.

Fortunato, M., Azar, M. G., Piot, B., Menick, J., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., Blundell, C., & Legg, S. (2018). *Noisy networks for exploration*. *International Conference on Learning Representations*.

Freire, I. T., Amil, A. F., & Verschure, P. F. M. J. (2024). *Sequential memory improves sample and memory efficiency in episodic control*. *Nature Machine Intelligence*, *7*(1), 43–55.
Friston, K. (2010). *The free-energy principle: A unified brain theory?* *Nature Reviews Neuroscience*, *11*(2), 127–138.
Fujimoto, S., van Hoof, H., & Meger, D. (2018). *Addressing function approximation error in actor-critic methods*. In J. Dy & A. Krause (Eds.), *Proceedings of the 35th international conference on machine learning* (pp. 1587–1596, Vol. 80). PMLR.
García, J., Fern, & o Fernández. (2015). *A comprehensive survey on safe reinforcement learning*. *Journal of Machine Learning Research*, *16*(42), 1437–1480.
Gardner, R. A., & Gardner, B. T. (1984). *A vocabulary test for chimpanzees (pan troglodytes).* *Journal of Comparative Psychology*, *98*(4), 381–404.
Guss, W. H., Codel, C., Hofmann, K., Houghton, B., Kuno, N., Milani, S., Mohanty, S., Liebana, D. P., Salakhutdinov, R., Topin, N., Veloso, M., & Wang, P. (2019). *The minerl 2019 competition on sample efficient reinforcement learning using human priors* [arxiv:1904.10079].
Ha, D., & Schmidhuber, J. (2018). *World models*.
Haarnoja, T., Tang, H., Abbeel, P., & Levine, S. (2017). *Reinforcement learning with deep energy-based policies*. In D. Precup & Y. W. Teh (Eds.), *Proceedings of the 34th international conference on machine learning* (pp. 1352–1361, Vol. 70). PMLR.
Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018). *Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor*. In J. Dy & A. Krause (Eds.), *Proceedings of the 35th international conference on machine learning* (pp. 1861–1870, Vol. 80).
Hafner, D., Lee, K.-H., Fischer, I., & Abbeel, P. (2022). *Deep hierarchical planning from pixels* (S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, & A. Oh, Eds.). *Advances in Neural Information Processing Systems*, *35*, 26091–26104.
Hafner, D., Lillicrap, T., Ba, J., & Norouzi, M. (2020). *Dream to control: Learning behaviors by latent imagination*. *International Conference on Learning Representations*.
Hafner, D., Lillicrap, T., Fischer, I., Villegas, R., Ha, D., Lee, H., & Davidson, J. (2018). *Learning latent dynamics for planning from pixels*.
Hafner, D., Pasukonis, J., Ba, J., & Lillicrap, T. (2025). *Mastering diverse control tasks through world models*. *Nature*, *640*(8059), 647–653.
Hansen, N., Su, H., & Wang, X. (2024). *TD-MPC2: Scalable, robust world models for continuous control*. *The Twelfth International Conference on Learning Representations*.
Hausknecht, M., & Stone, P. (2015). *Deep recurrent q-learning for partially observable mdps*. *Sequential Decision Making for Intelligent Agents Papers from the AAAI 2015 Fall Symposium*.
Hayes, C. F., Rădulescu, R., Bargiacchi, E., Källström, J., Macfarlane, M., Reymond, M., Verstraeten, T., Zintgraf, L. M., Dazeley, R., Heintz, F., Howley, E., Irissappane, A. A., Mannion, P., Nowé, A., Ramos, G., Restelli, M., Vamplew, P., & Roijers, D. M. (2022). *A practical guide to multi-objective reinforcement learning and planning*. *Autonomous Agents and Multi-Agent Systems*, *36*(1).
Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., & Meger, D. (2018). *Deep reinforcement learning that matters*. *Proceedings of the AAAI Conference on Artificial Intelligence*, *32*(1).
Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., & Silver, D. (2018). *Rainbow: Combining improvements in deep reinforcement learning*. *Proceedings of the AAAI Conference on Artificial Intelligence*, *32*(1).
Hester, T., & Stone, P. (2012). *Texplore: Real-time sample-efficient reinforcement learning for robots*. *Machine Learning*, *90*(3), 385–429.
Hester, T., Vecerik, M., Pietquin, O., Lanctot, M., Schaul, T., Piot, B., Horgan, D., Quan, J., Sendonaris, A., Osband, I., Dulac-Arnold, G., Agapiou, J., Leibo, J., & Gruslys, A. (2018). *Deep q-learning from demonstrations*. *Proceedings of the AAAI Conference on Artificial Intelligence*, *32*(1).
Hochreiter, S., & Schmidhuber, J. (1997). *Long short-term memory*. *Neural Computation*, *9*(8), 1735–1780.
Howard, R. A. (1960). *Dynamic programming and markov processes.* John Wiley.
Hu, E. S., Chang, R., Rybkin, O., & Jayaraman, D. (2023). *Planning goals for exploration*. *The Eleventh International Conference on Learning Representations*.
Hu, J., Wang, Z., Stone, P., & Martín-Martín, R. (2024). *Disentangled unsupervised skill discovery for efficient hierarchical reinforcement learning*. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, & C. Zhang (Eds.), *Advances in neural information processing systems* (pp. 76529–76552, Vol. 37). Curran Associates, Inc.
Ibarz, J., Tan, J., Finn, C., Kalakrishnan, M., Pastor, P., & Levine, S. (2021). *How to train your robot with deep reinforcement learning: Lessons we have learned*. *The International Journal of Robotics Research*, *40*(4-5).
Jiang, M., Rocktäschel, T., & Grefenstette, E. (2023). *General intelligence requires rethinking exploration*. *Royal Society Open Science*, *10*(6).

Kapturowski, S., Ostrovski, G., Quan, J., Munos, R., & Dabney, W. (2019). *Recurrent experience replay in distributed reinforcement learning*. *7th International Conference on Learning Representations*.

Kiran, B. R., Sobh, I., Talpaert, V., Mannion, P., Sallab, A. A. A., Yogamani, S., & Perez, P. (2022). *Deep reinforcement learning for autonomous driving: A survey*. *IEEE Transactions on Intelligent Transportation Systems*, *23*(6), 4909–4926.

Laud, A. D. (2004). *Theory and application of reward shaping in reinforcement learning* [Doctoral dissertation, University of Illinois at Urbana-Champaign].

LeCun, Y., Boser, B., Denker, J., Henderson, D., Howard, R., Hubbard, W., & Jackel, L. (1989). *Handwritten digit recognition with a back-propagation network*. In D. Touretzky (Ed.), *Advances in neural information processing systems* (Vol. 2). Morgan-Kaufmann.

Leike, J., Krueger, D., Everitt, T., Martic, M., Maini, V., & Legg, S. (2018). *Scalable agent alignment via reward modeling: A research direction* [arxiv:1811.07871].

Levine, S., & Abbeel, P. (2014). *Learning neural network policies with guided policy search under unknown dynamics*. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, & K. Weinberger (Eds.), *Advances in neural information processing systems* (Vol. 27). Curran Associates, Inc.

Levine, S., & Koltun, V. (2013). *Guided policy search*. In S. Dasgupta & D. McAllester (Eds.), *Proceedings of the 30th international conference on machine learning* (pp. 1–9, Vol. 28). PMLR.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2015). *Continuous control with deep reinforcement learning* [arxiv:1509.02971].

Lin, L.-J. (1992). *Reinforcement learning for robots using neural networks* [Doctoral dissertation]. Carnegie Mellon University.

Lin, Z., Zhao, T., Yang, G., & Zhang, L. (2018). *Episodic memory deep q-networks*. *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, 2433–2439.

Macenski, S., Foote, T., Gerkey, B., Lalancette, C., & Woodall, W. (2022). *Robot operating system 2: Design, architecture, and uses in the wild*. *Science Robotics*, *7*(66), eabm6074.

Mendonca, R., Rybkin, O., Daniilidis, K., Hafner, D., & Pathak, D. (2021). *Discovering and achieving goals via world models*. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, & J. W. Vaughan (Eds.), *Advances in neural information processing systems* (pp. 24379–24391, Vol. 34). Curran Associates, Inc.

Michie, D., & Chambers, R. A. (1968). *Boxes: An experiment in adaptive control*. *Machine intelligence*, *2*(2), 137–152.

Minsky, M. (1961). *Steps toward artificial intelligence*. *Proceedings of the IRE*, *49*(1), 8–30.

Mitchell, M. (2021). *Why ai is harder than we think*. *Proceedings of the Genetic and Evolutionary Computation Conference*.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). *Playing atari with deep reinforcement learning* [arxiv:1312.5602].

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M. A., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). *Human-level control through deep reinforcement learning*. *Nature 518*, *518*(7540), 529–533.

Moerland, T. M., Broekens, J., Plaat, A., & Jonker, C. M. (2023). *Model-based reinforcement learning: A survey*. *Foundations and Trends® in Machine Learning*, *16*(1), 1–118.

Montague, P. R., Dolan, R. J., Friston, K. J., & Dayan, P. (2012). *Computational psychiatry*. *Trends in Cognitive Sciences*, *16*(1), 72–80.

Narendra, K., & Parthasarathy, K. (1990). *Identification and control of dynamical systems using neural networks*. *IEEE Transactions on Neural Networks*, *1*(1), 4–27.

Ng, A. Y., Harada, D., & Russell, S. J. (1999). *Policy invariance under reward transformations: Theory and application to reward shaping*. *Proceedings of the Sixteenth International Conference on Machine Learning*, 278–287.

Oh, J., Chockalingam, V., Singh, S., & Lee, H. (2016). *Control of memory, active perception, and action in minecraft*. In M. F. Balcan & K. Q. Weinberger (Eds.), *Proceedings of the 33rd international conference on machine learning* (pp. 2790–2799, Vol. 48). PMLR.

Oh, J., Guo, X., Lee, H., Lewis, R. L., & Singh, S. (2015). *Action-conditional video prediction using deep networks in atari games*. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, & R. Garnett (Eds.), *Advances in neural information processing systems* (Vol. 28). Curran Associates, Inc.

Oudeyer, P.-Y., & Kaplan, F. (2007). *What is intrinsic motivation? a typology of computational approaches*. *Frontiers in neurorobotics*, *1*.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P. F., Leike, J., & Lowe, R. (2022). *Training language models to follow instructions with human feedback*. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, & A. Oh (Eds.), *Advances in neural information processing systems* (pp. 27730–27744, Vol. 35). Curran Associates, Inc.

Pateria, S., Subagdja, B., Tan, A.-h., & Quek, C. (2021). *Hierarchical reinforcement learning: A comprehensive survey*. *ACM Computing Surveys*, *54*(5), 1–35.
Pathak, D., Agrawal, P., Efros, A. A., & Darrell, T. (2017). *Curiosity-driven exploration by self-supervised prediction*. *Proceedings of the 34th International Conference on Machine Learning*, *PMLR 70*, 2778–2787.
Peng, X. B., Andrychowicz, M., Zaremba, W., & Abbeel, P. (2018). *Sim-to-real transfer of robotic control with dynamics randomization*. *IEEE International Conference on Robotics and Automation (ICRA)*.
Plappert, M., Houthooft, R., Dhariwal, P., Sidor, S., Chen, R. Y., Chen, X., Asfour, T., Abbeel, P., & Andrychowicz, M. (2018). *Parameter space noise for exploration*. *International Conference on Learning Representations*.
Rădulescu, R., Mannion, P., Roijers, D. M., & Nowé, A. (2019). *Multi-objective multi-agent decision making: A utility-based analysis and survey*. *Autonomous Agents and Multi-Agent Systems*, *34*(1).
Rescorla, R., & Wagner, A. (1972). *A theory of pavlovian conditioning: Variations in the effectiveness of reinforcement and nonreinforcement*.
Roijers, D. M., Vamplew, P., Whiteson, S., & Dazeley, R. (2013). *A survey of multi-objective sequential decision-making*. *Journal of Artificial Intelligence Research*, *48*, 67–113.
Rudin, C. (2019). *Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead*. *Nature Machine Intelligence*, *1*(5), 206–215.
Rummery, G. A., & Niranjan, M. (1994). *On-line q-learning using connectionnist systems* (tech. rep.). Engineering Department, Cambridge University.
Samek, W., Montavon, G., Lapuschkin, S., Anders, C. J., & Muller, K.-R. (2021). *Explaining deep neural networks and beyond: A review of methods and applications*. *Proceedings of the IEEE*, *109*(3), 247–278.
Samsami, M. R., Zholus, A., Rajendran, J., & Chandar, S. (2024). *Mastering memory tasks with world models*. *The Twelfth International Conference on Learning Representations*.
Samuel, A. L. (1959). *Some studies in machine learning using the game of checkers*. *IBM Journal of Research and Development*, *3*(3), 210–229.
Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2016). *Prioritized experience replay*. *4th International Conference on Learning Representations*.
Schmidhuber, J. (1991). *A possibility for implementing curiosity and boredom in model-building neural controllers*. *Proceedings of the international conference on simulation of adaptive behavior: From animals to animats*, 222–227.
Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., Lillicrap, T., & Silver, D. (2019). *Mastering atari, go, chess and shogi by planning with a learned model* [arxiv:1911.08265].
Schulman, J., Levine, S., Abbeel, P., Jordan, M., & Moritz, P. (2015). *Trust region policy optimization*. In F. Bach & D. Blei (Eds.), *Proceedings of the 32nd international conference on machine learning* (pp. 1889–1897, Vol. 37). PMLR.
Schulman, J., Moritz, P., Levine, S., Jordan, M., & Abbeel, P. (2016). *High-dimensional continuous control using generalized advantage estimation* (Y. Bengio & Y. LeCun, Eds.).
Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). *Proximal policy optimization algorithms* [arxiv:1707.06347].
Schultz, W., Dayan, P., & Montague, P. R. (1997). *A neural substrate of prediction and reward*. *Science*, *275*(5306), 1593–1599.
Sekar, R., Rybkin, O., Daniilidis, K., Abbeel, P., Hafner, D., & Pathak, D. (2020). *Planning to explore via self-supervised world models*. *Proceedings of the 37th International Conference on Machine Learning*, *PMLR 119*, 8583–8592.
Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T. P., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016). *Mastering the game of go with deep neural networks and tree search*. *Nature 529*, *529*(7587), 484–489.
Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., & Riedmiller, M. (2014). *Deterministic policy gradient algorithms*. In E. P. Xing & T. Jebara (Eds.), *Proceedings of the 31st international conference on machine learning* (pp. 387–395, Vol. 32). PMLR.
Silver, D., Singh, S., Precup, D., & Sutton, R. S. (2021). *Reward is enough*. *Artificial Intelligence*, *299*, 103535.
Stiennon, N., Ouyang, L., Wu, J., Ziegler, D., Lowe, R., Voss, C., Radford, A., Amodei, D., & Christiano, P. F. (2020). *Learning to summarize with human feedback*. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, & H. Lin (Eds.), *Advances in neural information processing systems* (pp. 3008–3021, Vol. 33). Curran Associates, Inc.
Sünderhauf, N., Brock, O., Scheirer, W., Hadsell, R., Fox, D., Leitner, J., Upcroft, B., Abbeel, P., Burgard, W., Milford, M., & Corke, P. (2018). *The limits and potentials of deep learning for robotics*. *The International Journal of Robotics Research*, *37*(4-5), 405–420.
Sutton, R. S. (1988). *Learning to predict by the methods of temporal differences*. *Machine Learning*, *3*(1), 9–44.

Sutton, R. S. (1991). *Dyna, an integrated architecture for learning, planning, and reacting*. *ACM SIGART Bulletin*, *2*(4), 160–163.
Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction, second edition*. The MIT Press.
Sutton, R. S., Precup, D., & Singh, S. (1999). *Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning*. *Artificial Intelligence*, *112*(1), 181–211.
Sutton, R. S., Szepesvári, C., Geramifard, A., & Bowling, M. (2008). *Dyna-style planning with linear function approximation and prioritized sweeping*. *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*, 528–536.
Sutton, R. S. (1984). *Temporal credit assignment in reinforcement learning* [Doctoral dissertation]. University of Massachusetts Amherst.
Tassa, Y., Doron, Y., Muldal, A., Erez, T., Li, Y., Casas, D. d. L., Budden, D., Abdolmaleki, A., Merel, J., Lefrancq, A., Lillicrap, T., & Riedmiller, M. (2018). *Deepmind control suite* [arxiv:1801.00690].
Tesauro, G. (1994). *Td-gammon, a self-teaching backgammon program, achieves master-level play*. *Neural Computation*, *6*(2), 215–219.
Tesauro, G., & Galperin, G. (1996). *On-line policy improvement using monte-carlo search*. In M. Mozer, M. Jordan, & T. Petsche (Eds.), *Advances in neural information processing systems* (Vol. 9). MIT Press.
Tessler, C., Givony, S., Zahavy, T., Mankowitz, D., & Mannor, S. (2017). *A deep hierarchical approach to lifelong learning in minecraft*. *Proceedings of the AAAI Conference on Artificial Intelligence*, *31*(1).
Thorndike, E. L. (1911). *Animal intelligence; experimental studies*. The Macmillan Company.
Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W., & Abbeel, P. (2017). *Domain randomization for transferring deep neural networks from simulation to the real world*. *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 23–30.
Todorov, E., Erez, T., & Tassa, Y. (2012). *Mujoco: A physics engine for model-based control*. *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 5026–5033.
Tulving, E. (2002). *Episodic memory: From mind to brain*. *Annual Review of Psychology*, *53*(1), 1–25.
Uchendu, I., Xiao, T., Lu, Y., Zhu, B., Yan, M., Simon, J., Bennice, M., Fu, C., Ma, C., Jiao, J., Levine, S., & Hausman, K. (2023). *Jump-start reinforcement learning*. In A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, & J. Scarlett (Eds.), *Proceedings of the 40th international conference on machine learning* (pp. 34556–34583, Vol. 202). PMLR.
van Hasselt, H. (2010). *Double q-learning*. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, & A. Culotta (Eds.), *Advances in neural information processing systems* (Vol. 23). Curran Associates, Inc.
van Hasselt, H., Guez, A., & Silver, D. (2016). *Deep reinforcement learning with double q-learning*. In D. Schuurmans & M. P. Wellman (Eds.), *Proceedings of the thirtieth AAAI conference on artificial intelligence, february 12-17, 2016, phoenix, arizona, USA* (pp. 2094–2100). AAAI Press.
Vecerik, M., Hester, T., Scholz, J., Wang, F., Pietquin, O., Piot, B., Heess, N., Rothörl, T., Lampe, T., & Riedmiller, M. (2017). *Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards* [arxiv:1707.08817].
Walter, W. G. (1950). *An imitation of life*. *Scientific American*, *182*(5), 42–45.
Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., & de Freitas, N. (2016). *Dueling network architectures for deep reinforcement learning*. In M. Balcan & K. Q. Weinberger (Eds.), *Proceedings of the 33rd international conference on machine learning, ICML 2016, new york city, ny, usa, june 19-24, 2016* (pp. 1995–2003, Vol. 48).
Watkins, C. J. C. H. (1989). *Learning from delayed rewards* [Doctoral dissertation, Cambridge University, Cambridge, England].
Wierstra, D., Foerster, A., Peters, J., & Schmidhuber, J. (2007). *Solving deep memory pomdps with recurrent policy gradients*. In J. M. de Sá, L. A. Alexandre, W. Duch, & D. Mandic (Eds.), *Artificial neural networks – icann 2007* (pp. 697–706). Springer Berlin Heidelberg.
Williams, R. J. (1992). *Simple statistical gradient-following algorithms for connectionist reinforcement learning*. *Machine Learning*, *8*(3–4), 229–256.
Wooldridge, M. (2009). *An introduction to multiagent systems*. John Wiley & Sons.
Wooldridge, M., & Jennings, N. R. (1995). *Intelligent agents: Theory and practice*. *The Knowledge Engineering Review*, *10*(2), 115–152.
Xu, J., Tian, Y., Ma, P., Rus, D., Sueda, S., & Matusik, W. (2020). *Prediction-guided multi-objective reinforcement learning for continuous robot control*. In H. D. III & A. Singh (Eds.), *Proceedings of the 37th international conference on machine learning* (pp. 10607–10616, Vol. 119). PMLR.
Yang, R., Sun, X., & Narasimhan, K. (2019). *A generalized algorithm for multi-objective reinforcement learning and policy adaptation*. *Advances in Neural Information Processing Systems*, *32*.
Zhu, C., Dastani, M., & Wang, S. (2024). *A survey of multi-agent deep reinforcement learning with communication*. *Autonomous Agents and Multi-Agent Systems*, *38*(1).

Zhu, G., Lin, Z., Yang, G., & Zhang, C. (2020). *Episodic reinforcement learning with associative memory*. *International Conference on Learning Representations*.