University Tunis El Manar
National Engineering School of Tunis

Information and Communication Technologies Department

End-of-Year Project

Created by :

Mohamed MAHDI

Raed BOUSSAA

Class :

1ATEL 1

Exploring Reinforcement Learning Algorithms

Supervised by :

Mme. Wafa MEFTEH

2023-2024

Table of Contents

List of Figures

List of Tables

Acknowledgments

Firstly, we are very thankful to God for guiding us through this journey and granting us the strength and wisdom to overcome challenges along the way.

We are deeply grateful to ENIT for providing us with the opportunity to pursue our studies and take this project. The support and resources provided by the institution have been invaluable in shaping my academic journey.

We would like to express our gratitude to all the individuals who supported us throughout this end of year project.

We would like to extend our gratitude and acknowledge the entire community of researchers and developers in the field of Reinforcement Learning (RL) whose collective efforts have significantly enriched and advanced the domain.

We would like to thank the professor Wafa Mefteh for accepting us into her team, we also thank Mr. Ali Frihida, the chief for the TIC department of ENIT.

We are grateful to them for their patience and invaluable guidance throughout this project. Their availability and commitment were essential in helping us overcome all the difficulties and achieve our goals.

I would also like to express my appreciation to my colleagues for their collaboration, support, and enriching exchanges we had throughout this project.

Finally, and most importantly, we would like to thank ourselves for our commitment, hard work, and the collaborative spirit we shared throughout this project . Despite the challenges we encountered, we persevered with determination and resilience. It is through our collective effort and dedication that we have successfully brought this project . Let us take a moment to celebrate our accomplishments and the invaluable contribution each of us has made to its success.

General Introduction

Reinforcement Learning (RL) represents an advanced approach within artificial intelligence, it introduces a learning method where computational agents independently explore the environment, make decisions, and modify their strategies to maximize rewards. RL is learning how to map situations to actions so that we maximize our reward signal. The agent (the learner) is not taught what actions to pick, instead he must discover what are the actions that result in the highest reward possible by trying them. In complex case studies, actions may influence the instant rewards as well as the next situation and, by induction, all subsequent rewards. *Trial and error search* and *delayed reward* are the two most important characteristic features of RL. *Agents* are computational entities that have the capacity to see their environment, take actions, and adjust based on input within the shape of rewards or punishments. These agents aim to determine an ideal set of rules with the objective of maximizing future rewards. The RL process presents in a nonstop loop of exploration and exploitation. During exploration, agents try different actions and moves to understand their impacts, and during exploitation, they use the knowledge they gained earlier to make better decisions and increase anticipated expected rewards. *Reinforcement Learning* has made significant advances over the years, leading to applications in a variety of industries, including finance, healthcare, and energy. RL algorithms, particularly deep neural networks, have revolutionized the ability of agents to generalize knowledge and solve complex problems.

Past the agent and the environment, we are able to identify 7 first elements of a RL model : An agent, an environment, a state, an action, a policy, a reward signal and a value function.
An agent is an autonomous entity that interacts with the environment to achieve a well defined goal. It learns by trial and error, receiving feedback in the form of rewards based on its actions. The agent's objective is to maximize cumulative rewards over time by selecting the highest rewarded actions. Through iterative exploration and exploitation, the agent refines its decision-making policy, aiming to achieve optimal performance.
The environment is the outside framework with which the agent interacts. It may be a physical environment, a recreated environment in a laptop program, or a digital environment.

A state represents the present situation or configuration of the environment at a specific timestep. It encapsulates all of the applicable statistics essential for the agent to make decisions.
An action is a choice or preference made through the agent at a specific state. Actions have an effect on the following states and the rewards obtained through the agent.
The policy is the approach or rule that the agent follows to pick movements at every state. It maps states to actions or probabilities of choosing movements. Policies may be deterministic which means choosing the identical movement for a given state, or stochastic choosing movements probabilistically.
The reward signal in reinforcement learning serves as a crucial component, directing the agent at his choice making process. This numerical feedback provided by the environment after each action, indicates how desirable the agent's choices are. By associating actions with rewards, the agent learns to improve its behavior over time, aiming to maximize the total rewards.
The value function estimates the anticipated cumulative reward an agent can get from a given state beneath a certain approach. It shows the long-term desirability of being in a specific state. Value functions offer assistance to the agent to estimate the quality of different states and make choices appropriately.

Reinforcement Learning (RL) pursues a few critical objectives and tackles numerous issues in different areas. One of the most objectives of RL is to enable agents to learn ideal decision-making procedures through interaction with the environment. This incorporates tasks such as robot control, and resource optimization. RL looks to solve complex issues related to vulnerability, dynamic situations, and the require for adjustment. In fields like healthcare and logistics, RL is used to improve resources allocation and enhance decision-making processes. Moreover, RL encourages the advancement of intelligent agents that can learn from experience, thereby decreasing dependence on explicit programming. Therefore, the most objective of RL is to form operators that can successfully explore and succeed in real-world scenarios by continually learning, adjusting, and optimizing their behavior based on input and rewards from the environment.

CHAPTER 1

Tabular Techniques in RL

Introduction

In this chapter, we're diving into different ways to solve problems in reinforcement learning. We'll start by exploring something called Multi-armed Bandit problems. These are like choosing between different slot machines, each with its own rewards. We'll learn methods to make smart choices and get the most rewards.

After that, we'll talk about Finite Markov Decision Processes (MDPs). These are about making decisions in situations where the outcome is uncertain. We'll learn how to figure out the best strategies for these situations using techniques like Policy Evaluation, Policy Improvement, and others.

Then, we'll look at Monte Carlo Methods and Temporal Difference Learning. These are ways to learn from experience and improve decision-making over time. We'll explore different approaches like SARSA and Q-Learning.

By the end of this chapter, we'll have a good understanding of these methods, which will help us solve a wide range of problems in reinforcement learning.

1.1 Multi-armed Bandit

A *bandit* is a thief that steals money from people. A *one-armed bandit* represents a slot machine (the one we find in casinos) where you pull the lever, and if it's your lucky day you win a sum of money. The reward in this case is immediate. We call it a bandit because it is well known that casinos set tricky configurations in the slot machines, so that gamblers lose more than they win.

The *multi-armed bandit* is a slot machine featuring multiple levers that a gambler operates independently. Each lever provides a distinct reward, and the probabilities of rewards associated with each lever are independently distributed and typically unknown to the gambler. The picture, 1.1, illustrates the multi-armed bandit in action.

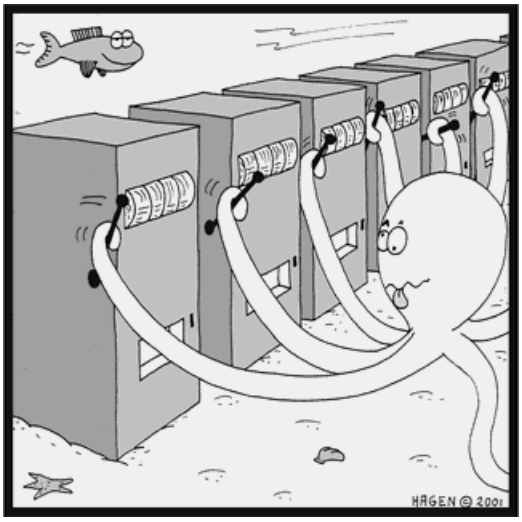

FIGURE 1.1 – A gambler playing a multi-armed bandit. Image sourced from the website Towards Data Science article 'Solving the Multi-Armed Bandit Problem'[15].

1.1.1 A k-armed Bandit Problem

We will be repeatedly selecting actions. Our goal then is to maximize our winnings by focusing on the most rewarding levers.

In our k-armed bandit problem, each of the k actions provides an expected or mean reward when selected. This expected reward is referred to as the *value* of that action. We represent the action selected at time step t as A_t, and its corresponding reward as R_t. The value of any arbitrary action a, denoted as $q^*(a)$, is the expected reward when action a is selected :

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

If we knew the value of each action, solving the k-armed bandit problem becomes trivial : we simply select the action with the highest value as the optimal action. However, generally it is not the case, we will then rely on statistics and estimations. The estimated value of action a at time step t, denoted as $Q_t(a)$, has to closely approximate $q^*(a)$, using an effective solution.

At any time step t, there is generally at least one action with the highest estimated value, called the *greedy* action. Choosing greedy actions constitutes *exploitation* of current knowledge learned. Alternatively, opting for non-greedy actions is called *exploration*. In the beginning of our training exploring non-greedy actions may be advantageous to discover more awarding actions. Although we might not get immediate rewards when exploring, we will learn better actions that will pay off in the long run. The decision to explore or exploit depends on the precision of estimates, uncertainties, and the remaining number of steps.

1.1.2 Action-value Methods

One method to estimate the values of actions and use them to select the action is called *action-value methods*. We recall that the exact action value is the average reward we get when choosing this action. We will then estimate the value of the action by averaging the rewards received from the training :

$$Q_t(a) = \frac{\text{sum of the rewards when } a \text{ was taken prior to } t}{\text{total number of times } a \text{ was taken prior to } t} = \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{I}(A_i = a)}{\sum_{i=1}^{t-1} \mathbb{I}(A_i = a)}$$

In our case, $\mathbb{I}$ evaluates to 1 if the predicate $(A_i = a)$ is true and 0 otherwise. As the denominator approaches infinity, $Q_t(a)$ tends toward $q^*(a)$. This method is known as the *sample-average method* for estimating action values since each estimate represents the average of a relevant reward sample.

We define our action selection rule as to select one of the greedy actions (one of the actions with the highest estimated value). If there is multiple greedy actions, then we select one randomly. We denote this greedy action selection algorithm as :

$$A_t = \text{argmax}_a Q_t(a),$$

where argmax_a presents the action a for which the action value $Q_t(a)$ is maximum and where ties are broken randomly. Greedy action selection always *exploits* current learned actions to maximize immediate reward. It does consider seemingly inferior actions to determine if they might actually be better. An alternative approach involves behaving greedily most of the time, but occasionally, with a small probability ϵ, we select randomly from among all actions with equal probability, regardless of the action-value estimates. We refer to methods employing this near-greedy action selection rule as *ϵ-greedy methods*.

Following this method, each action will be sampled a quasi infinite number of times as the number of steps increases, thus we ensure that all $Q_t(a)$ converge to $q_*(a)$.

1.2 Finite Markov Decision Process

Markov Decision Process (MDP) is used for the decision-making process in a stochastic environment. The goal of MDP is to give the mapping of optimal actions for each state of an environment. The *Markov principle* states that the future state of a system depends only on its present state and is not influenced by past states, provided that the present state is completely observed. In other words, the Markov principle states that future events are conditionally independent of past events, given the present state of the system.
A Markov state carries all important statistics from the past to expect the future accurately. For example, in games like Tic-tac-toe, the present board configuration serves as a Markov state as it contains all statistics required to predict future developments. We don't want to concern ourselves with the complete collection of past movements ; the present state is enough because

it encapsulates all applicable statistics.

The agent interacts with the environment in discrete steps. At each step, we have a state, based on which the agent takes the most optimal action from the set of all possible actions. In the next step, the agent receives a new state and a new reward (see Figure 1.2).

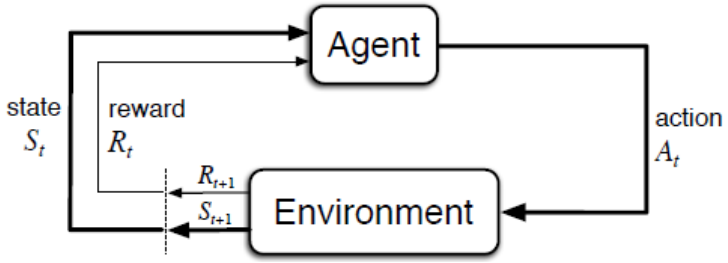

FIGURE 1.2 – The agent–environment interaction in a Markov decision process. Image sourced from the book 'Reinforcement Learning : An Introduction' [14].

In cases where the agent's interaction with the environment naturally breaks into steps, the goal is to maximize the expected return. Each step ends in a terminal state, and the game is reset to a starting state. For continuing tasks, where the interaction continues indefinitely without a final time step, the concept of the return becomes problematic. For instance, if the agent receives a reward of +1 at each time step, the return would be infinite since there is no final time step. To address this issue, we introduce discounting, denoted by the parameter γ, where $0 \leq \gamma \leq 1$. The agent's objective is to select actions that maximize the expected discounted reward, where the reward received k time steps later, is valued as the original reward multiplied by γ^{k-1}.

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

Finding the most optimal policy is necessary to solve complex RL problems. For finite Markov decision processes, an optimal policy is defined as follows : a policy π is better than a policy π' when the return value of π is greater or equal to the return value of π'. Put differently,

$$\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s)$$

where $v_\pi(s)$ is the state value within the policy π. We define the most optimal policy as : $v_*(s) = \max_\pi v_\pi(s)$.

1.3 Dynamic Programming

Dynamic programming (DP) is an approach to solve optimization problems by breaking them down into smaller problems.

In the context of RL, *dynamic programming* is based on the Markov Decision Process structure. It involves two fundamental concepts : the *value function* and the *policy*. It also involves two essential steps : *policy evaluation* and *policy improvement*. Policy evaluation estimates the

value function for a given policy, while policy improvement involves refining the policy based on the current value function.

In order to build and organize good policies, the use of the value function is essential. The optimal value function v_* and the optimal action-value function q_* verify the Bellman optimality equations :

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')], \text{ and} \\ q_*(s, a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')] \end{aligned}$$

1.3.1 Policy Evaluation

Policy evaluation in DP refers to computing the state-value function v_π of the policy π. We also refer to it as the *prediction problem.*

For all $s \in S$,

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t | S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left(r + \gamma v_\pi(s')\right), \end{aligned}$$

where $\pi(a|s)$ is the probability of taking the action a in the state s under the policy π.

Let's examine a series of approximate value functions $v_0, v_1, v_2, \ldots$, each one of them maps S^+ to $\mathbb{R}$. The initial approximation, v_0, is randomly selected (with the condition that the terminal state, if it exists, must be assigned 0). Subsequent approximations are derived using the Bellman equation for v_π as the update rule :

$$\begin{aligned} v_{k+1}(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left(r + \gamma v_k(s')\right), \end{aligned}$$

for all $s \in S$.

The sequence of value functions $\{v_k\}$ converges to v_π as $k \to \infty$. This is known as the *iterative policy evaluation algorithm.*

1.3.2 Policy Improvement

Given the value function v_π for a specific policy π, we can decide based on the policy evaluation whether altering the policy to select a different action $a \neq \pi(s)$ for a given state s would be beneficial. This is done by simply comparing the expected value of sticking to the current policy, $v_\pi(s)$, with the potential value of adopting the new policy.
One way to determine whether it's more awarding to change the policy or not is by selecting action a when in state s and then continue following the current policy, π. This approach evaluates the potential value gained from such behavior :

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \,|\, S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r \,|\, s, a)\,(r + \gamma v_\pi(s'))\,. \end{aligned}$$

The crucial concept is to determine whether $q_\pi(s,a) > v_\pi(s)$ or $q_\pi(s,a) < v_\pi(s)$. If $q_\pi(s,a) > v_\pi(s)$, we can assume that selecting action a every time we encounter the state s would lead to a more optimal policy on the whole.

This theorem, known as the *policy improvement theorem*, states that for any deterministic policies pair (π, π'), if for all states $s \in S$, the action-value function under policy π' is greater than or equal to the value function under policy π :

$$q_\pi(s, \pi'(s)) \geq v_\pi(s),$$

Thus, π' must outperform or at least be as good as π. In other words, it should yield an equivalent or higher expected reward from all states $s \in S$:

$$v_{\pi'}(s) \geq v_\pi(s).$$

1.3.3 Policy Iteration

After enhancing a policy π using v_π to produce a better policy π', we can then calculate $v_{\pi'}$ and enhance it further to derive an even better policy π''. This process generates a sequence of progressively improving policies and value functions :

$$\pi \xrightarrow{\text{E}} v^\pi \xrightarrow{\text{I}} \pi' \xrightarrow{\text{E}} v^{\pi'} \xrightarrow{\text{I}} \pi'' \xrightarrow{\text{E}} \ldots \xrightarrow{\text{I}} \pi^* \xrightarrow{\text{E}} v^*,$$

where $\xrightarrow{\text{E}}$ signifies policy evaluation and $\xrightarrow{\text{I}}$ represents policy improvement. Every policy represents a clear enhancement over the preceding one, except when the previous policy is already optimal. Since a finite MDP has a finite number of policies, this iterative process must converge to an optimal policy and optimal value function within a finite number of iterations. This method of identifying an optimal policy is referred to as *policy iteration*.

1.3.4 Value Iteration

Policy iteration presents a drawback in that each iteration necessitates policy evaluation, which might be computationally costly in terms of time and memory. If policy evaluation is conducted iteratively, exact convergence to v_π is achieved solely in the asymptotic limit.

However, there are various ways to truncate the policy evaluation step in policy iteration without compromising its convergence guarantees. A notable instance is when policy evaluation is terminated after a single update of each state. This approach is known as *value iteration*. It can be expressed as a straightforward update operation that integrates both policy improvement and truncated policy evaluation steps :

$$v_{k+1}(s) = \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a] = \max_a \sum_{s',r} p(s', r|s, a) \left(r + \gamma v_k(s')\right)$$

1.4 Monte Carlo Methods

Monte Carlo methods offer a unique approach to estimate the value function and discover the optimal policy, it requires the experience from previous simulated interactions with the environment. MC methods rely on the approximation and the ability to estimate complex systems, its main idea consists of averaging the returns of multiple episodes.
When estimating the value of a state s under a policy π we may consider a set of episodes that pass by the state s, if that state is visited more than once in the same episode (experience) we can either consider all visiting when calculating the average or we can only take in consideration only the first visit for each state.
Algorithm 1 below is derived from the book of Sutton and Barto [14].

A model of the enviromment is crucial in order to compute the state values, so when the model is unavailable, it becomes neccesary for us to estimate action values rather than state values. For this reason, *Monte Carlo methods* for action values focus on visits to state-action pairs rather than states alone.

Monte Carlo methods can also be used to determine an approximation of the optimal policy through a process named *generalized policy iteration* (GPI), it involves updating the approximate policy and the the approximate value function. This process helps these both estimations to reach optimality over time.

Algorithm 1 First-visit MC prediction for estimating V_π

Require: A policy π to be evaluated
  Initialize :
  for each state $s \in S$ **do**
    $V(s) \leftarrow$ arbitrary value
    Returns$(s) \leftarrow$ empty list
  end for
  loop
    Generate an episode following $\pi : S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    for each step of episode, $t = T-1, T-2, \ldots, 0$: **do**
      $G \leftarrow \gamma G + R_{t+1}$
      if S_t does not appear in $S_0, S_1, \ldots, S_{t-1}$ **then**
        Append G to Returns(S_t)
        $V(S_t) \leftarrow$ average(Returns(S_t))
      end if
    end for
  end loop

The policy improvement is executed by ensuring that the policy becomes greedy with regard to the current value function. As a result, the policy improvement theorem is applicable to π_k in the following manner :

$$\begin{aligned} q_{\pi_k}(s, \pi_{k+1}(s)) &= q_{\pi_k}(s, \arg\max_a q_{\pi_k}(s,a)) \\ &= \max_a q_{\pi_k}(s,a) \\ &\geq q_{\pi_k}(s, \pi_k(s)) \\ &\geq v_{\pi_k}(s). \end{aligned}$$

In order to avoid unrealstic assumptions in monte carlo methods, we use the *on-policy method*. On-policy method generally assigns positive probability to all actions in all states. One of the most known approaches with on-policy methods is ϵ-greedy policy, which chooses the action with the highest estimated action, but in some cases it takes random actions based on a probability ϵ, which makes a balance between exploring and exploiting.
The policy improvement theorem guarantees that any ϵ-greedy policy with respect to q_π represents an enhancement over any ϵ-soft policy π. Denote π' as the ϵ-greedy policy. The policy improvement theorem conditions are satisfied due to the fact that for every state $s \in S$,

$$\begin{aligned} q_\pi(s, \pi'(s)) &= \sum_a \pi'(a|s) q_\pi(s,a) \\ &= \frac{\epsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + (1-\epsilon) \max_a q_\pi(s,a) \\ &\geq \frac{\epsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + (1-\epsilon) \sum_a \pi(a|s) \frac{-\epsilon}{1-\epsilon} q_\pi(s,a) \end{aligned}$$

(The summation represents a weighted average comprising non-negative weights that sum up to 1. Consequently, it cannot exceed the largest value being averaged.)

$$= \frac{\epsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) - \frac{\epsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + \sum_a \pi(a|s) q_\pi(s,a)$$
$$= v_\pi(s).$$

Thus, following the policy improvement theorem, $\pi' \geq \pi$ ($v_{\pi'}(s) \geq v_\pi(s)$ (for all $s \in S$). This proves that equality can hold only when π' and π are both optimal among all the ϵ-soft policies. Meaning that they are better than or equal to all other policies.

1.5 Temporal Difference Learning

Temporal-Difference learning is an important technique in *reinforcement learning*, mixing thoughts from each Monte Carlo methods and Dynamic Programming. TD can learn from experience without requiring a model of the environment. Like DP, TD methods update estimates based on the learned estimates, without waiting for a final outcome. TD methods do not require a model of the environment, the reward or its probability of the next state which makes it more practical in real life problems. Also TD methods can be used in an online, fully incremental learning which means learning step by step unlike Monte Carlo methods which requires the returning of a complete episode which can be critical for applications with long episodes.
The TD method is also called TD(0), or one-step TD, because it updates its optimal policy on each iteration. The algorithm below specifies TD(0) completely in procedural form.

Algorithm 2 Tabular TD(0) for estimating v_π

Require: The policy π to be evaluated
  Algorithm parameter : Step size $\alpha \in (0, 1]$
  Initialize $V(s)$ for all $s \in S^+$, arbitrarily except that $V(\text{terminal}) = 0$
  loop
    For each episode :
    Initialize S
    loop
      For each step of episode :
      A action given by π for S
      Take action A, observe R, S_0
      $V(S) \leftarrow V(S) + \alpha\left(R + V(S_0) - V(S)\right)$
      $S \leftarrow S_0$
      until S is terminal
    end loop
  end loop

It has been shown that TD(0) method always converges to the correct value function for any policy, thus convergence is guaranteed under certain parametres

1.5.1 On-Policy Learning with SARSA

SARSA, an abbreviation for *State-Action-Reward-State-Action*, is another fundamental method in reinforcement learning that operates *on-policy*, meaning that it updates its Q-value function based on the previous taken action.
One notable characteristic of SARSA is its on-policy nature, SARSA updates its Q-values using the same policy that is being improved upon. This means that the action chosen in the next state directly influences the update to the Q-value of the current state.

The equation for updating the Q-value in SARSA is given by :

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[R + \gamma Q(s',a') - Q(s,a)\right]$$

Here, α represents the learning rate, which typically ranges between 0 and 1, and R denotes the reward received. The discount factor γ determines the extent to which future rewards are considered.

SARSA's on-policy approach ensures that it learns from the current policy. This characteristic makes SARSA particularly suitable for scenarios where stability and safety are crucial, such as in robotics or autonomous systems.

1.5.2 Off-policy Learning with Q-Learning

Q-learning wich stands for *quality learning* is a prominent method in reinforcement learning that operates without relying on a predefined model of the environment. Through iterative exploration and learning, it enables agents to make optimal decisions in various scenarios. The Q-learning process involves agents interacting with an environment, making decisions based on current states, receiving rewards or penalties for actions taken, and updating a Q-table to guide future actions.

One significant advantage of Q-learning is its model-free nature, which eliminates the need for prior knowledge about the environment. This flexibility makes it applicable to scenarios where environment dynamics are complex or unknown.

Q-learning stands out from earlier algorithms by adopting an *off-policy method*. Unlike earlier approaches that didn't distinguish between behavior and learning, Q-learning separates the acting policy from the learning policy. This means it updates the current Q-value function independently of the action taken. Consequently, even if the action chosen in the next state isn't optimal, this information doesn't affect the updating of the Q-function for the current state. Equation for the Q-value is as follows :

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[R + \gamma \max Q(s',a') - Q(s,a)\right]$$

where α is the learning rate and has a value between 0 and 1. R is a reward and is the reduction rate of the reward as time passes.

This flexibility and adaptability make Q-learning suitable for a wide range of problems and environments, including offline training on pre-collected datasets.

Conclusion

In this chapter, we delve into the methodologies employed in reinforcement learning problem-solving, particularly focusing on tabular methods. Beginning with the rudimentary Multi-armed Bandit problem, we elucidate its fundamental version and the strategies employed to tackle it. Progressing further, we encounter more intricate scenarios known as Finite Markov Decision Processes, pivotal in the realm of reinforcement learning.

Within this framework, our discourse gravitates towards Dynamic Programming techniques, akin to systematic blueprints aimed at determining optimal solutions. We expound upon four primary methodologies : Policy Evaluation, Policy Improvement, Policy Iteration, and Value Iteration, instrumental in facilitating informed decision-making within these processes. Subsequently, we embark on an exploration of Monte Carlo Methods, offering avenues to resolve predicaments through simulating numerous plausible scenarios, especially beneficial in navigating uncertain terrains.

This chapter also delves into Temporal Difference Learning, a methodology predicated on iterative learning through hypothesis and information assimilation. We elucidate two prominent approaches within this paradigm : SARSA and Q-Learning, devised to glean insights from experiences and refine decision-making prowess over time. In essence, Chapter 1 serves as a comprehensive primer on the diverse methodologies harnessed to tackle challenges in reinforcement learning, traversing a spectrum from elementary scenarios to more intricate predicaments.

CHAPTER 2

Approximation Techniques in RL

Introduction

In this chapter, we delve into approximate solution methods in reinforcement learning, which offer practical approaches for handling large-scale problems. We start by exploring On-policy Prediction with Approximation, where we seek to estimate the value function using approximation techniques. This involves methods like Value Function Approximation, where we aim to represent the value function in a compact form, and Linear Methods, which provide efficient ways to approximate complex functions.

We then delve into On-policy Control with Approximation, focusing on techniques for finding optimal policies while approximating the value function. Through Episodic Semi-gradient Control, we navigate the challenge of balancing exploration and exploitation to converge towards optimal policies. Additionally, we explore the concept of Average Reward, introducing a new problem setting for continuing tasks that offers fresh insights into reinforcement learning.

Transitioning to Off-policy Methods with Approximation, we explore semi-gradient methods that enable learning from experiences generated by different policies. We also discuss the concept of Learnability, shedding light on the theoretical aspects of approximation in reinforcement learning and its implications for practical applications.

Through these discussions, we aim to equip readers with a deeper understanding of approximate solution methods and their applications in addressing complex reinforcement learning problems.

2.1 On-policy Prediction with Approximation

In many scenarios employing Reinforcement Learning, the state space is expansive and composed of combinations, rendering it exceedingly vast. Under these circumstances, achieving an optimal policy or optimal value function becomes impractical. Instead, our objective is to derive a satisfactory approximation with restricted computational capabilities.

In this chapter, a novel method is introduced wherein the approximate value function is depicted through a parameterized functional structure, characterized by a weight vector $\mathbf{w} \in \mathbb{R}^d$. We will denote the approximate value of state $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$, given weight vector $\mathbf{w}$.
By extending reinforcement learning to function approximation, *reinforcement learning* becomes applicable to partially observable problems, where the full state is not observable to the agent.

2.1.1 Value Function Approximation

In this part we will update our initial estimated value function. This function shifts its value toward a *backed-up value*, or update target for that state.

Let consider our individual update $s \mapsto u$. Where s is the state and u is update target that's s estimated value is shifted toward. In the *Dynamic Programming* policy-evaluation update $s \mapsto E_\pi[R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}_t)|S_t = s]$, the state s is updated, however in the other cases the actual experience's state S_t is the one updated. In essence, updating s to u implies that the estimated value for a state s should resemble the update target u to a greater extent.

Machine learning methods that aim to replicate input-output examples are known as *Supervised Learning* methods. When the outputs are numerical, such as u, this process is also referred to as *function approximation*. Function approximation methods rely on examples that demonstrate the desired input-output policy that they are trying to approximate.

In *Reinforcement Learning*, it is vital for learning to take place online, allowing the agent to interact dynamically with its environment.

2.1.2 The Prediction Objective ($\overline{VE}$)

Despite employing a genuine approximation approach, updates in one state influences multiple others, making it infeasible to precisely determine the values for all states. Consequently, it becomes necessary to specify the states of primary concern.

To mathematically represent this concept, let's introduce a state distribution $\mu(s) \geq 0$, where $\sum_s \mu(s) = 1$, indicating the degree of importance assigned to the error in each state s. We define the error as :

$$e(s, \mathbf{w}) = [v_\pi(s) - \hat{v}(s, \mathbf{w})]^2$$

Weighting this over the state space by μ, we obtain a natural objective function, the *mean squared value error*, denoted ($\overline{VE}$). Such that :

$$\overline{VE} = \sum_{s \in S} e(s, \mathbf{w})$$

Under On-policy training this is called the *the on-policy distribution*.

Utility of The Mean Squared Value Error

An optimal objective concerning $\overline{VE}$ would be identifying a global optimum, represented by a weight vector w^*, where $\overline{VE}(\mathbf{w}^*) \leq \overline{VE}(\mathbf{w})$ for all feasible values of $\mathbf{w}$. Achieving this

objective is occasionally viable for straightforward approximations like linear models, but seldom feasible for intricate functions such as artificial neural networks or decision trees. Under such circumstances, complex function approximators might converge towards a *local optimum*, represented by a weight vector $\mathbf{w}^*$, where $\overline{VE}(\mathbf{w}^*) \leq \overline{VE}(\mathbf{w})$ for all $\mathbf{w}$ within a certain interval that includes $\mathbf{w}^*$.

This typically represents the optimal approach for nonlinear function approximators, and frequently proves to be adequate. However, in numerous scenarios relevant to reinforcement learning, there is no guarantee of reaching an optimum solution, or even being close to one.

2.1.3 Linear Methods

In this scenario, the approximate function $\hat{f}(., \mathbf{w})$ takes the form of a linear function involving the weight vector $\mathbf{w}$. For each state s, there exists a real-valued vector $x(s) = \begin{bmatrix} x_1(s) \\ x_2(s) \\ \vdots \\ x_n(s) \end{bmatrix}$, containing the same number of components as $\mathbf{w}$.

Linear techniques estimate the state-value function by computing the inner product between $\mathbf{w}$ and $x(s)$.

$$\hat{v}(s, \mathbf{w}) = \mathbf{w}^T x(s) = \sum_{i=1}^{d} \mathbf{w}_i x_i(s)$$

Here, $x(s)$ denotes the feature vector representing state s.

2.1.4 Stochastic Gradient-descent (SGD) for Linear Methods

Stochastic gradient-descent (SGD) try to minimize the $\overline{VE}$ by adjusting the weight vector

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \frac{1}{2}\alpha\nabla[v_\pi(s_t) - \hat{v}(s_t, \mathbf{w}_t)]^2$$
$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha[v_\pi(s_t) - \hat{v}(s_t, \mathbf{w}_t)]\nabla\hat{v}(s_t, \mathbf{w}_t)$$

In linear case $\nabla\hat{v}(s, \mathbf{w}) = x(s)$, let's also assume that the target output, that we denote $U_t \in \mathbb{R}$ of the t^th training, $S_t \mapsto U_t$, is not the exact value, $v_\pi(s_t)$, but a random approximation to it. In our case, we cannot use the exact value of $v_\pi(s_t)$ but an approximation of it. As a result,

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha[U_t - \hat{v}(s_t, \mathbf{w}_t)]x(s_t)$$

If U_t is an *unbiased* estimate, meaning $\mathbb{E}[U_t|s_t = s] = v_\pi(s_t)$, for each training t, then $\mathbf{w}_t$ is guarenteed to converge to a local optimum.
Here the weight vector is a point near the local optimum. The update at each training time t is

$$\begin{aligned}\mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha\big(R_{t+1} + \gamma\mathbf{w}^T x(s_{t+1}) - \mathbf{w}^T x(s_t)\big)x(s_t) \\ &= \mathbf{w}_t + \alpha\Big(R_{t+1}x(s_t) - x(s_t)\big(x(s_t) - \gamma x(s_{t+1})\big)^T\mathbf{w}_t\Big)\end{aligned}$$

For any given $\mathbf{w}_t$, the next weight vector can be noted as :

$$\mathbb{E}\left[\mathbf{w}_{t+1}|s\right] = \mathbf{w}_t + \alpha(b - A\mathbf{w}_t)$$

We prove that $b = \mathbb{E}\left[R_{t+1}x(s_t)\right]$ and

$A = \mathbb{E}\left[x(s_t)\big(x(s_t) - \gamma x(s_{t+1})\big)^T\right] \in \mathbb{R}^d \times \mathbb{R}^d$.

If the system achieves convergence, it will asymptotically approach and stabilize at the weight vector. $\mathbf{w}_{TD}$ that implies :

$$\mathbb{E}\left[\mathbf{w}_{t+1} \mid s\right] = \mathbf{w}_t \Leftrightarrow b - A\mathbf{w}_{TD} = 0 \Leftrightarrow \mathbf{w}_{TD} = A^{-1}b$$

Here $\mathbf{w}_{TD}$ is called *temporal difference fixed point*, and at the TD fixed point, it is proved that :

$$\overline{VE}(\mathbf{w}_{TD}) \leq \frac{1}{1-\gamma} \min_w \overline{VE}$$

2.2 On-policy Control with Approximation

In this section, we delve into the analysis of control methods, this time employing parametric approximation for the action-value function $\hat{q}(s, a, \mathbf{w}) \approx q_*(s, a)$, such that $\mathbf{w} \in \mathbb{R}^d$ represents our finite-dimensional weight vector. We expand upon the concepts of function approximation previously applied to state values to investigate action values.

2.2.1 Episodic Semi-gradient Control

In this case it is the approximate-value function $\hat{q} \approx q_\pi$, that is represented as a parameterized functional for with weight $\mathbf{w}$.

Example : Mountain Car Task

Let's examine the challenge of navigating a low-powered vehicle along a steep mountain road, as illustrated in the picture 2.1 below.

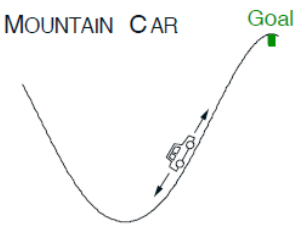

FIGURE 2.1 – The Mountain Car task. Image sourced from the book 'Reinforcement Learning : An Introduction' [14].

The difficulty in our example is that the pulling gravity is much stronger than the car's engine, so even at full acceleration the car cannot accelerate up to its final Goal.
The only solution here is to go up to the left before moving towards the goal. Then, by applying full acceleration the car can develop enough inertia to reach the goal, even though gravity is slowing it down the whole time. This is an example of a *continuous control task*, meaning that things must get worse (get far from the destination) before getting better (get closer the the goal).
The reward is (-1) on all training steps except when the car moves toward its final goal, which ends the episode. There are three possible actions :

- Full acceleration forward (+1)
- Full acceleration backward (-1)
- Zero acceleration (0)

Let's model the car's movement according to simplified classical mechanics. Its positon x_t and its velocity, $\dot{x}_t$ follow these rules :

$$x_{t+1} = x_t + \dot{x_{t+1}}$$

$$\dot{x}_{t+1} = \dot{x}_t + \alpha A_t + \beta \cos(\omega x_t)$$

The feature vector $x(s,a)$ is combined linearly with the parameter vector to approximate the action-value function $\hat{q}(s,a,w) = \mathbf{w}^T x(s,a) = \sum_{i=1}^{d} \mathbf{w}_i x_i(s,a)$ for each state s and action a.

2.2.2 Average Reward : A new problem Setting for Continuing Tasks

We define *continuing problems*, problem for which the agent-environement interation goes on forever without beggining state or ending state. The context of the *mean reward environment* is relevant to ongoing challenges. In this instance, our agent assigns equal importance to both immediate and mean rewards. Within the mean reward framework, the efficacy of a policy π is measured by its mean reward rate.

$$r(\pi) = \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) r$$

Here expectations are conditioned on the initial state, S_0. Whereas, the subsequent actions $A_0, A_1, ...A_{t-1}$ is taken according to our policy π. Partically, we order policies according to their average reward per time step $r(\pi)$.
Within the mean reward environment, returns are characterized by the disparities between rewards and the average reward : $G_t = R_{t+1} - r(\pi) + R_{t+2} - r(\pi) + R_{t+3} - r(\pi)$.... This concept is referred to as *differential return*, with the associated value functions termed *differential value functions*.

2.3 Off-policy Methods with Approximation

In this section, we will explore the convergence problems and will take a closer look at the theory for linear function approximation as well. Additionally, we will introduce the notion of learnability and discuss new algorithms with stronger convergence guarantees.

2.3.1 Semi-gradient Methods

To transform *off-policy* algorithms into semi-gradient form, we merely substitute the update of an array (v or q) with an update to the weight vector $\mathbf{w}$, utilizing the approximate value function ($\hat{v}$ or $\hat{q}$) and its gradient. A majority of these algorithms employ the *per-step importance sampling ratio* :

$$\rho_t = \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$$

Where, $b(a|s)$ denotes the behavior policy, it is used to select actions while learning about the target policy π.
For example, in the *one-step state-value algorithm* we add the ρ_t :

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\rho_t\delta_t\nabla\hat{v}(S_t, \mathbf{w}_t)$$

2.3.2 Learnability

In the realm of reinforcement learning, the concept of "learnability" pertains to the ability to be learned with a certain level of experience. However, it is discovered that numerous concepts cannot be learned in reinforcement learning, even with an infinite amount of experimental data. While these quantities are well-defined and computable with knowledge of the internal environment structure, they remain uncomputable or unestimable from the observable sequence of features, actions, or reward sets. As a result, we classify them as being *non-learnable*.

Example of Non-learnable Concepts : The Bellman Error

Consider the two *Markov reward process* 2.2 diagrammed below :

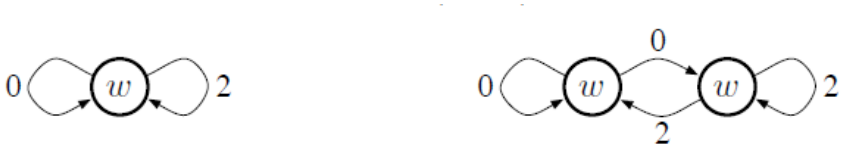

FIGURE 2.2 – Markov reward process diagram. Image sourced from the book 'Reinforcement Learning : An Introduction' [14].

Consider a scenario where a state has two outgoing transitions, each occurring with equal probability. The numerical values denote the rewards associated with these transitions. In one path, the Markov Reward Process (MRP) remains in the same state and yields either 0s or 2s randomly at each step, each with a probability of $\frac{1}{2}$. Conversely, in the other path, the MRP either remains in its current state or transitions to the other state with equal probability at each step. Although the rewards are deterministically 0 or 2 in the MRP, the observable data consists of a sequence of 0s and 2s generated randomly, just like that produced by the first path. Thus, despite having access to an infinite amount of data, discerning between the two processes is infeasible.

Conclusion

In wrapping up this chapter, let's reflect on what we've learned about approximate solution methods in reinforcement learning. These methods offer us practical ways to deal with big and complex problems, making it easier to find solutions. We started by learning how to estimate the value function using approximation techniques. This helps us handle situations where the number of possible states is too large to deal with directly. Techniques like Linear Methods simplify the representation of the value function, making it more manageable. Moving on to finding optimal policies, we explored methods like Episodic Semi-gradient Control. These techniques help us balance between exploring new options and exploiting what we already know to find the best policies efficiently. We also discussed the concept of Average Reward, which gives us a new perspective on tasks that continue indefinitely.

We then looked into Off-policy Methods, which allow us to learn from different policies. This flexibility is valuable in real-world scenarios where we might have data from various sources. Understanding the theory behind Learnability helped us grasp the potential and limitations of approximation in reinforcement learning. Overall, these approximate solution methods are powerful tools for tackling real-world challenges in reinforcement learning. By mastering these techniques, we can handle complex problems more effectively and drive progress in artificial intelligence.

As we move forward, let's keep exploring and experimenting with these methods, pushing the boundaries of what's possible in reinforcement learning. With dedication and creativity, we can continue to advance the field and make meaningful contributions to AI research and applications.

CHAPTER 3

Reinforcement Learning for Generative AI

Introduction

In this chapter, we'll explore how reinforcement learning (RL) techniques can be used in generative artificial intelligence (AI). We'll discuss different methods and applications where RL helps in training models to generate various outputs, optimize objectives, and improve non-quantifiable features.

We'll start by looking at RL for simple generation tasks like sequences and codes. Techniques like SeqGAN and Proximal Policy Optimization are used here. Then, we'll move on to RL for maximizing specific objectives, such as improving translation quality or generating narrative plots with specific characteristics.

Finally, we'll discuss how RL can enhance subjective features in generated outputs. One method we'll explore is Reinforcement Learning from Human Preferences, where RL learns from human feedback to improve outputs.

A wide variety of life applications rely on generation, including machine translation, text summarization, image captioning, and a lot more. From a *machine learning* (ML) perspective, data generation is the problem of predicting a syntactically and semantically correct data given some context. For example, given an image, generate an appropriate caption or given a sentence in English language, translate it to Arabic. *Generative AI* (GenAI) is the class of *machine learning* (ML) algorithms that can learn from data like text, images, and audio in order to generate new content. One major recent development of GenAI is the introduction of OpenAI's GPT-3 and GPT-4 models, which can generate human-like language output. Altough all these achievements, there are many open doubts for how to make GenAI more user friendly. For these reasons, it is crucial to use *reinforcement learning* to mitigate biases and align GenAI models to human values.

3.1 RL for Mere Generation

Mere generation refers to training a GenAI model to approximate outputs from a given domain of interest as accurate as possible. As matter of fact, RL can be used as a solution to the generative modeling problem like text generation. The generative model plays the role of the *agent* and the generated output represents the *state*. *Actions* model how the state can be modified and finally, the *reward* is an indicator of the quality of the output generated. Three essential design components can be identified :

— The instantiation of the agent.
— The specification of the system dynamics.
— The selection of the reward system.

3.1.1 Sequence Generative Adversarial Nets (SeqGAN)

Generative Adversarial Networks (GANs) are a framework for training generative models in an *adversarial* setup, with a *generator* generating data that is trying to fool a *discriminator* that is trained to discriminate between real and generated data.

Given a dataset comprising structured sequences from the real world, our objective is to train a generative model G_θ, parameterized by θ, to generate a sequence $Y_{1:T} = (y_1, \ldots, y_t, \ldots, y_T)$, where $y_t \in Y$ and Y represents the set of all possible words or symbols appearing in the generated sequences. This problem is referred to as the *sequence generation problem*, which we approach through the lens of reinforcement learning.

Interpreting this problem within the context of reinforcement learning, we consider timestep t, where the state s is represented by the current sequence of generated information $(y_1, \ldots, y_{t-1})$, and the action a corresponds to selecting the next information y_t. We also train a *φ-parameterized discriminative model* D_φ to provide guidance to improve generator G_θ. Here, $D_\varphi(Y_{1:T})$ is a probability indicating how a sequence $Y_{1:T}$ is coming from real sequence data or not.

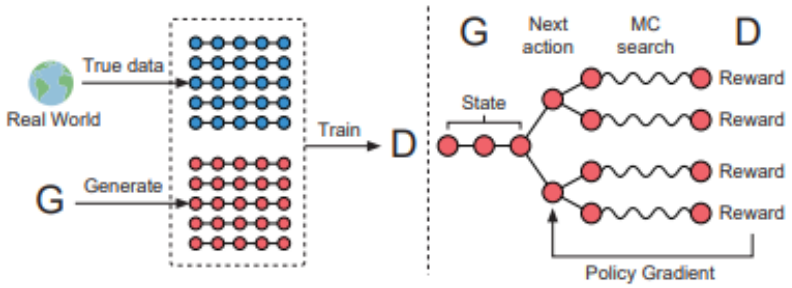

FIGURE 3.1 – The illustration of SeqGAN. Image retrieved from SeqGAN : Sequence Generative Adversarial Nets with Policy Gradient[1]

In SeqGAN, there are two main parts : *the Discriminator (D)* and *the Generator (G)* (See figure 3.1). The Discriminator learns from real and generated data, then figures out which is (D) and which is (G). The Generator gets feedback from the Discriminator to improve its output, and to make it more like the real data. In short, *SeqGAN* works by training both parts

together, and the *the Discriminator (D)* guides the *the Generator (G)* to create more realistic data.

3.1.2 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is an on-policy algorithm that aims at making the biggest possible improvement step on a policy using the current data, without stepping so far that it might cause performance collapse. PPO is a first-order technique family that is easy to use and appears to work well experimentally. PPO investigates by sampling activities in accordance with its most recent stochastic policy. Initial circumstances and the training process determine how random the action selection process is. Over training, the policy gets less and less random as the update rule pushes it to take advantage of rewards that it has previously discovered . This may lead the policy to get stuck in a local optima.
A proximal policy optimization (PPO) algorithm 3 is shown below.

Algorithm 3 Proximal Policy Optimization

1: **for** iteration $= 1, 2, \ldots$ **do**
2: **for** actor $= 1, 2, \ldots, N$ **do**
3: Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
4: Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
5: **end for**
6: Optimize surrogate L with respect to θ, with K epochs and minibatch size $M \leq NT$
7: $\theta_{\text{old}} \leftarrow \theta$
8: **end for**

3.1.3 CodeRL Algorithm For Code Generation

The term "program synthesis" or "*code generation*" refers to the process of creating a program that solves a particular issue. the problem description for the program synthesis job is given as an input sequence D and the program $\hat{W} = (\hat{w}_1, \ldots, \hat{w}_T)$, $\hat{w}_t \in V$ which is the program's possible output sequence. We suggest treating the program synthesis as an RL problem and using an actor-critic RL technique to enhance a previously trained Language Model's performance .
An error predictor is the definition of the critic model. Both the programs and the problem requirements are sent into the model as input sequences. The model is trained to predict one of the following results for each program : {CompileError, RuntimeError, FailedTest, PassedTest}. Baseline programs are taken into consideration, and relative returns are included into the loss function to optimise the neural network that is an element of our reinforcement learning architecture in order to improve and stabilise the training process. The functional soundness of the created program is used to determine the return r that is given to the model. The predicted return should be minimised as the aim of RL fine-tuning :

$$L_{\text{RL}}(\theta) = -\mathbb{E}_{W^s \sim p_\theta}[r(W^s)]$$

where $W_s = (w_1^s, \ldots, w_T^s)$ is a generative sample.

3.2 RL for Objective Maximization

Reinforcement learning can be formalized as an objective maximization problem. RL for quantity maximization has been mainly adopted in text generation, especially for dialogue and translation. RL allows mitigating the tendency of models trained using supervised learning to perform poorly when they are deployed for usage.

3.2.1 Google's Neural Machine Translation System

Neural Machine Translation NMT)is an automated translation technique that utilises end-to-end learning and has the capability of overcoming the weaknesses of traditional phrase-based translation systems. But a number of writers have claimed that NMT algorithms are not strong enough, especially when input sentences contain uncommon terms. We will thus use a human side-by-side judgement on a collection of discrete short statements. With this, translation mistakes will be reduced by an average of sixty percent as compared to Google's phrase-based production approach.

with a dataset of parallel text in which contains N input-output sequence pairs, denoted $D \equiv \{(X^{(i)}, Y^{*(i)})\}_{i=1}^{N}$. Our goal is to estimate the parameters θ of a statistical model in a way that it maximizes the probability of observing correct outputs given the corresponding inputs in the dataset.

$$\mathcal{L}_{\mathrm{ML}}(\theta) = \sum_{i=1}^{N} \log P_\theta(Y^{*(i)}|X^{(i)}).$$

We consider model refinement using the expected reward objective , which can be expressed as

$$\mathcal{O}_{\mathrm{RL}}(\theta) = \sum_{i=1}^{N} \sum_{Y \in \mathcal{Y}} P_\theta(Y|X^{(i)}) \cdot r(Y, Y^{*(i)}).$$

Here, $r(Y, Y^{*(i)})$ denotes the score per sentence, and we are computing an expectation over all of the output sentences $\mathcal{Y}$ (See picture 3.2).
In the preceding equation we deduce the mean reward from $r(Y, Y^{*(i)})$. The sample mean of m sequences selected randomly from distribution $P_\theta(Y|X^{(i)})$ is used to estimate the mean. We optimise a linear mixture of ML and RL objectives as follows to further stabilise training :

$$\mathcal{O}_{\mathrm{Mixed}}(\theta) = \alpha \cdot \mathcal{O}_{\mathrm{ML}}(\theta) + \mathcal{O}_{\mathrm{RL}}(\theta)$$

We demonstrate that the model approaches the accuracy obtained by typical bilingual human translators using a humanely rated comparison as a metric

3.2.2 Refining Story Generation via Reward Shaping in Neural Networks

Approaches that are based on language modeling to *generate a story plot* attempt to construct the plot by picking words from a language model (LM) to guess what word comes

next in order to add it to the story.LM techniques cant get help from the user to reach a specific goal, which leads to stories that don't have coherence. So, we must implement a technique called *Reward Shaping* to help our model learn better. This technique looks at a bunch of stories and gives the model rewards along the way. These rewards help the model get closer to our final goal.
We make up stories by figuring out what to do and how to change things to get what we want. The aim of this work is to make sure that a certain verb (like waddle, doodle, munch) is the last thing that happens in the story. We use a method called reinforcement learning to figure out what to do in a story and use a way of learning called policy gradients to make a model of how to act. We begin by teaching a language model on a bunch of stories.

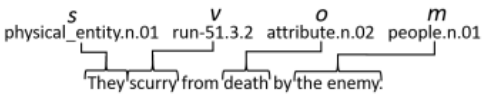

FIGURE 3.2 – A sentence and its representation of the action. Image captured from Controllable Neural Story Plot Generation via Reward Shaping [5]

A Language Model $P(x_n|x_{n-1}.x_{n-k};\theta)$gives a range of possible states. x_n that are probable to come next given a history of states $x_{n-1}.x_{n-k}$ and the parameters of a model θ.

With the help of repeatedly sampling from this language version, we use it to generate a plot.However, there is no guarantee that the plot will arrive at a desired goal. To fix this issue, we will apply *Policy gradient learning*. It's true that this method needs a lot of reward signals in order to give a feedback each step but in return it assures that are rewards are not very sparse.

Policy Gradient Descent

We choose our policy model θ in a way that $P(e_{t+1}|e_t;\theta)$ is the distribution over events according to the story's corpus and also that increase the probability of reaching a futur goal event .an action selects the most likely subsequent event. The event one from the probability distribution of the language model. The calculation of the reward is done. Determine the distance between the event et+1 and our goal event. The last gradient utilized. Adjusting the networks parameters and changing the language models distribution is necessary. The calculation is as follows :

$$\nabla_\theta J(\theta) = R(v(e_{t+1}))\nabla_\theta \log P(e_{t+1}|e_t;\theta)$$

where e_{t+1} and $R(v(e_{t+1}))$ are, respectively, the event at the timestep $t+1$ and the reward for the verb in that event. The policy gradient technique provides an advantage to highly rewarding events by enabling a more significant movement to the probability which predicts these events , over events that have a lower reward.

Reward Shaping

To make sure the *controllability* in the plot generation, we chiose to design the reward system in a way that we give a reward the network every time it generates an event which increases its probability to achieve its goal.

the reward function is created by pre-processing the training stories and determining two important factors :

— the distance of each verb to target or goal verb

— the verbs frequency in the existing stories

Distance

The distance measures how close is the verb v of an event to the wanted verb g, It is employed to give the model reward for events it generates that have verbs that are more close to the goal verb.for a verb v the formula that calculates this metric is :

$$r_1(v) = \log\left(\sum_{s \in S_{v,g}} l_s - d_s(v,g)\right)$$

where $S_{v,g}$ is the portion of the corpus's stories that have v before the target verb g, l_s is the story length s, and $d_s(v,g)$ is the events number between the event that v and the event that contains g . A greater reward is obtained by shortening the story when the events involving v and g are closer together.

Story-Verb Frequency

Rewarding story-verb frequency depends on which verb is most likely to appear in a story before the goal verb. This segment calculates the frequency with which a certain verb v occurs before the goal verb g across all tales in the corpus. By doing this, the model is deterred from producing events that begin with verbs that don't often appear in stories before the target verb. The story-verb frequency measure is computed using the following formula :

$$r_2(v) = \log \frac{k_{v,g}}{N_v}$$

where N_v represents the occurence of the verb v in the corpus, and $k_{v,g}$ is the count of v that appears prior the target verb g in any story.

Final Reward

The sum of the distance and frequency metrics determines the ultimate reward for a verb.Every verb in the corpus has its rewards normalised. The final reward is :

$$R(v) = \alpha \times r_1(v) \times r_2(v)$$

where the normalisation constant is denoted by α. Together, these indicators provide verbs that

— Advantage verbs in which are present close to the target.

— Advantage verbs which appear prior the target in a story in such a frequency that makes it significant.

This reward shaping approach develops a strategy that creates stories that are probabilistically similar with the training corpus, whereas language model–based narrative and plot creation systems result in stories that are aimless.

3.3 Enhancing Non-Quantifiable Features

Using a specific evaluation to check the efficiency of our model during testing, assures that the model performs similarly to how it was trained. However, evaluation don't always match up with what humans think. Even if a model gets good scores based on these measurements, it might still make mistakes that human users find obvious or produce results that don't make sense in real-life situations.

Furthermore, there could be certain characteristics that do not have a corresponding metric because they are subjective, hard to define, or not measurable. Typically, users only have an implicit understanding of the task objective, making it nearly impossible to design a suitable reward function. this issue is often called the agent alignment problem. A promising path is to use rewards as a way to guide the model.
Our method is to get the reward function by getting human feedback and then we try to optimize the function that we got. The reward functions should :

1. Allows us to take on tasks where we can merely identify the required behavior rather than necessarily demonstrating it,
2. enables begginer users to teach agents
3. Scales to complex problems
4. Economical with users feedback

The introduced algorithm makes a reward function to the human's preferences, while simultaneously developing a policy to maximise the reward function's present prediction. Rather than providing an exact score, we ask the human to compare brief video clips of the agent's actions (See figure 3.3). We discovered that while comparisons are equally helpful for understanding human preferences, they are more straightforward for humans to offer in some fields.

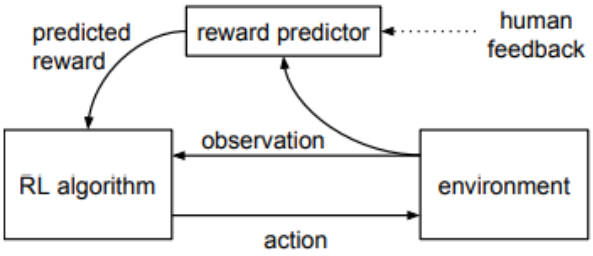

FIGURE 3.3 – The reward predictor learns separately from comparing different parts of the environment, and then the agent tries to get the highest reward predicted by this predictor. Image captured from : Deep Reinforcement Learning from Human Preferences[4]

Think of an agent that interacts with an environment in a series of stages. at each time t, The agent communicates an action to the environment at $t \in \mathcal{A}$ after receiving an observation from the environment at $t \in \mathcal{O}$. The agent's objective in classical reinforcement learning is to maximize the discounted sum of rewards, while the environment also provides a reward, $r_t \in \mathbb{R}$. We assume the existence of a human user who may express preferences between a series of observations and actions, $\sigma \in (\mathcal{O} \times \mathcal{A})^k$, rather than the environment feeding back a reward signal. To show that the human favored sequence σ_1 over sequence σ_2, write $\sigma_1 \succ \sigma_2$.

Informally, the agent's objective is to generate sequences that the human prefers.while making as few queries as possible to the human.
We will evaluate our algorithms' behavior in two ways :

Quantitative

We say that preferences $\succ$ are generated by a reward function $r : O \times A \to \mathbb{R}$ if

$$r(o_1^0, a_1^0, \ldots, o_1^{k-1}, a_1^{k-1}) \succ r(o_2^0, a_2^0, \ldots, o_2^{k-1}, a_2^{k-1})$$

whenever

$$r(o_1^0, a_1^0) + \ldots + r(o_1^{k-1}, a_1^{k-1}) > r(o_2^0, a_2^0) + \ldots + r(o_2^{k-1}, a_2^{k-1}).$$

Our AI should get a large total reward if the human's preferences are determined by a reward function r. Thus, we may assess if we are aware of the reward function r. the agent quantitatively. Ideally, the agent will achieve a reward nearly as high as if it had been using reinforcement learning (RL) to optimize its behavior.

Qualitative

this approach would be practically useful when we dont have a way to measure behavior quantitatively. this sentence : In these cases, all we can do is qualitatively evaluate how well the agent satisfies to the humans preferences. We begin by defining a goal expressed in natural language, and then ask a human to evaluate the agents behavior based on how well it achieves that goal

At every step, we will keep a policy $\pi : O \to A$ and a reward function approximation $\hat{r} : O \times A \to \mathbb{R}$, each limited by deep neural networks.
Three procedures are used to update these networks :

1. A bunch of sequences $\{\tau_1, \ldots, \tau_i\}$ are produced by the policy π that interacts with the environment. A classical reinforcement learning algorithm updates the parameters of π to maximise the sum of the anticipated rewards $r_t = \hat{r}(o_t, a_t)$.
2. we choose pairs of segments σ_1, σ_2 and forward them to a human for comparison.
3. the parameters $\hat{r}$ get optimised to fit the comparisons that have been gathered thus far from humans.

Trajectories flow from the process number (1) to the process number (2), human comparisons from process (2) to process (3), and parameters for $\hat{r}$ from process (3) to process (1) in an asynchronous manner.

Once rewards get computed with $\hat{r}$, we are left with a classical reinforcement learning problem. Any RL algorithm that suits the domain can be used to solve this problem. We like techniques like *Policy Gradient Methods* or *Temporal Difference (TD) Learning* since they are flexible to changes in the reward because of a subtlety : the reward function $\hat{r}$ might not be non-stationary.

Conclusion

In conclusion, this chapter has shed light on the diverse applications of reinforcement learning (RL) in generative artificial intelligence (AI). We've explored various methods and use cases where RL techniques play a crucial role in training models to generate outputs, optimize objectives, and enhance non-quantifiable features. Starting with RL for Mere Generation, we discussed techniques like SeqGAN and Proximal Policy Optimization, which enable the generation of high-quality sequences and codes. The CodeRL algorithm showcased the specialized application of RL in generating code snippets, demonstrating its effectiveness in various domains. Moving on to RL for Objective Maximization, we explored how RL is used to optimize specific objectives, such as translation quality in Google's Neural Machine Translation System. We also discussed narrative plot generation with specific characteristics, highlighting the potential of RL in creative content generation. Finally, we examined Enhancing Non-Quantifiable Features, where RL techniques are employed to improve subjective aspects of generated outputs. Reinforcement Learning from Human Preferences emerged as a promising approach to incorporate human feedback in the training process, leading to outputs that better align with human preferences.

Overall, the applications discussed in this chapter illustrate the versatility and effectiveness of RL in generative AI. By leveraging RL techniques, researchers and practitioners can tackle a wide range of tasks, from generating diverse outputs to optimizing subjective features. As the field continues to evolve, we anticipate further innovations and advancements in the intersection of RL and generative AI, driving progress in artificial intelligence and enriching human-computer interaction.

CHAPTER 4

Reinforcement Learning Implementation

Introduction

In this chapter, we will be applying *reinforcement learning algorithms* to various scenarios using *OpenAI's Gym environments*. Gymnasium, a version of OpenAI's Gym library, provides a collection of Python implementations featuring well-defined observation spaces, action spaces, and predefined reward functions for different models. Using this library, we will tackle a range of reinforcement learning problems, including both *model-based* and *model-free* scenarios. Additionally, we will conduct benchmark tests and performance comparisons among different algorithms to assess their effectiveness in solving these problems.

4.1 Model-based Reinforcement Learning

The game is set in a gymnasium environment, and our goal is to teach our agent (the computer) to navigate through this environment. We'll break down the code step by step, showing how the computer learns to make decisions while playing the game.

Specifically, we'll dissect the process of defining a policy, loading it from an external JSON file, and applying it within the environment to observe the agent's actions. Through this examination, we aim to illuminate the mechanics behind reinforcement learning in the context of a simple yet illustrative gaming scenario.

4.1.1 Cliff Walking Environment

Cliff walking includes going through a grid-world from start to target while avoiding falling off a cliff. The game starts with the player at the initial location [3, 0] of the 4x12 grid world (See figure 4.1) and the goal is located at [3, 11]. If the agent (the player in our case) reaches the goal the episode ends. The cliff runs along [3, 1..10]. If the player collides with the cliff it returns to the start location. The player makes actions taking him to the desired state until they

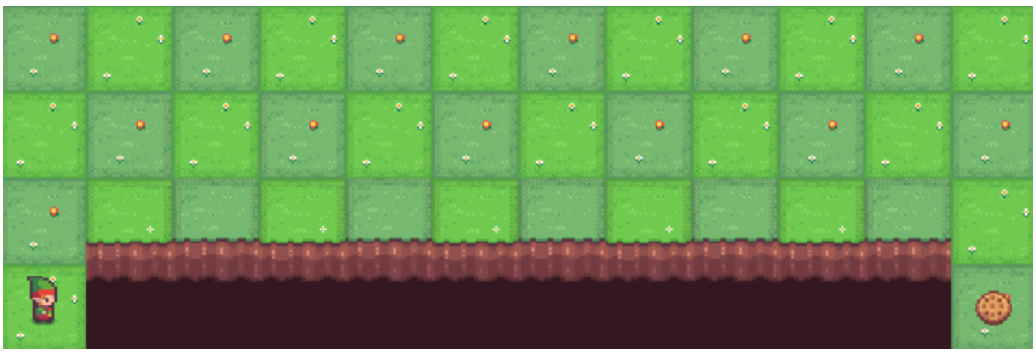

FIGURE 4.1 – Cliff walking game. Image retrieved from OpenAI's gymnasium documentation[12]

reach the goal. The certainty in the agent's mouvement makes our environment a *model-based environment*.

Action Space

The action shape is $(1,)$ in the range $\{0, 3\}$ indicating which direction to move the player.
— **0** : Move up
— **1** : Move right
— **2** : Move down
— **3** : Move left

Observation Space

There are $3 \times 12 + 1$ possible states. The player can't be in a cliff, also he can't be at the goal . the only remainig positions are the first 3 upper rows as well as the bottom left cell.
The observation is a number, where the row and col start at 0, that represents the player's current location as current_row $\times$ nrows + current_col.
For example we can calculate the positions beginning as follows : $3 \times 12 + 0 = 36$.

starting state

The player is in state [36] at location $[3, 0]$ at the beginning of the episode.

reward

Unless the player went into the cliff in which case the reward is -100, each time step results in a -1 reward.

Episode End

The episode ends when the player enters state [47] (location $[3, 11]$).

4.1.2 Policy Improvement Algorithm

This method improves our initial policy by selecting actions that yield the highest rewards in each situation. It evaluates the rewards associated with different actions in each state, determines the optimal action for each state, and updates the policy accordingly. The resulting improved policy is then stored in a file for future use.

After repeating the policy improvement process for several iterations, we observed successful convergence to the optimal policy. Each iteration refined the policy, gradually enhancing its performance until reaching a constant policy, which is, in fact, the optimal policy. For detailed numerical values and further insights into the algorithmic implementation, readers are encouraged to refer to the code implementation available in our GitHub repository (see Appendix A A.1.1).

4.1.3 Value Iteration

As we saw in the previous section, *policy improvement* iteratively improves the policy based on state values, which can be time and memory consuming.

Value iteration represents an advancement over the policy improvement method by directly focusing on optimizing the value function. Unlike policy improvement, this approach compresses the process, bypassing the need for separate policy evaluation and improvement steps, resulting in faster convergence and often more efficient outcomes.

Pseudo code for the training loop will be provided in **Appendix A (A.1.2)**.

4.2 Model-free Reinforcement Learning

The purpose of frozen lake is to walk through the frozen lake from start to finish without slipping into any holes. The frozen lake is slick, thus the player may not always go inyo the direction they wanted. Our agents' movements are uncertain, which renders our *Model-free*.

4.2.1 Frozen Lake Environment

The goal of the game is located at the far end of the globe, [3,3] for the 4x4 setting (See figure 4.2), while the player starts the game at point [0,0] of the frozen lake grid-world.
When using a pre-defined map, the holes in the ice are spread in pre-defined locations ; when a random map is constructed, the holes are located randomly.
The player continues to progress until they either fall into a hole or reach the goal.
The player can move perpendicularally to the desired direction due to the slick nature of the water.
In universes created at random, there is always a means to get there.

FIGURE 4.2 – Frozen Lake game. Image retrieved from OpenAI's gymnasium documentation[12]

Space

The player's direction of movement is indicated by the action shape (1,) in the range {0, 3}.

- 0 : Motion left
- 1 : Motion downward
- 2 : Motion right
- 3 : Motion upward

In world created at random, there is always a way to the goal

Observation space

The observation is a number `current_row * nrows + current_col`, where nrows and col start at 0. This value represents the player's current location.
For instance, $3 * 4 + 3 = 15$ is the calculation for the goal location in the 4x4 map. The size of the map determines how many observations are feasible.

Initial Condition

The player is in state [0] at location [0, 0] at the beginning of the episode.

Rewards

Reward schedule :

- Reach goal : +1
- Reach hole : 0
- Reach frozen : 0

Episode End

The episode ends if the following happens :

Termination :

- The player moves into a hole.
- The player reaches the goal at `max(nrow) * max(ncol) - 1` (location [max(nrow)-1, max(ncol)-1]).

Truncation (when using the time_limit wrapper) :

The episode lasts 100 minutes in a 4x4 setting and 200 minutes in an 8x8-v1 environment.

4.2.2 Training Loop

In the implementation of various *model-free reinforcement learning methods*, a common training loop structure is employed. This loop iterates over a specified number of episodes, during each of which the agent interacts with the environment by selecting actions, observing outcomes, and updating its Q-values accordingly. However, every specific algorithm updates the Q-values differently. In the next sections, we will compare the different `update_q_table` functions. Pseudo code for the training loop will be provided in **Appendix A (A.2)**.

We will be using the following hyper-parameters throughout our experiments :

— $\alpha = 0.1$: Learning rate.
— $\gamma = 1$: Discount factor.
— num_episodes = 1000000 : Number of episodes for training.

4.2.3 Implementing SARSA algorithms

Basic SARSA

We recall from 1.5.1 that the *SARSA* (State-Action-Reward-State-Action) algorithm is an *on-policy temporal difference reinforcement learning* method. In SARSA, the agent updates its Q-values based on the Bellman equation for Q-function updates. The Q-value of the current state-action pair is updated by considering the immediate reward received and the estimated Q-value of the next state-action pair, all weighted by the learning rate (α) and discount factor (γ). This process is iteratively applied within the training loop to learn the optimal policy.

For SARSA, the update rule for the Q-table is given by :

$$Q(s,a) = \alpha\left(r + \gamma Q(s',a')\right) + (1-\alpha)Q(s,a)$$

where :

— s is the current state,
— a is the action taken,
— r is the received reward,
— s' is the next state,
— a' is the next action,
— α is the learning rate, and
— γ is the discount factor.

The pseudo code for this update rule can be found in **Appendix A (A.3.1)**.

Expected SARSA

Expected SARSA is one of the algorithms that we didn't cover in the previous chapter, as it is considered an improvement of the *SARSA algorithm.*

In this approach, we compute the expected values of Q-values for all possible next actions weighted by their probabilities. In our particular case, all actions are equiprobable as we are picking actions randomly. By considering this approach rather than choosing a single sampled action in each timestep, *Expected SARSA* provides a more accurate estimation of the Q-value for the current state-action pair. It achieves this by updating the Q-value using the Bellman equation :

$$Q(s,a) = (1-\alpha) \cdot Q(s,a) + \alpha \cdot (r + \gamma \cdot \mathbb{E}[Q(s',a')])$$

The pseudo code for the update method in Expected SARSA can be found in **Appendix A (A.3.2)**.

4.2.4 Implementing Q-Learning Algorithms

Basic Q-learning

We have seen in 1.5.2 that the *Q-learning algorithm* is an *off-policy temporal difference reinforcement learning technique* used for estimating the Q-value function. Unlike SARSA, which is an *on-policy method*, Q-learning updates the Q-table by considering the maximum Q-value of the next state regardless of the action taken. This approach allows Q-learning to learn the optimal policy while following a different policy for exploration.

However, this off-policy approach can lead to *sub-optimal solutions* in environments where exploration is necessary. Since Q-learning always assumes the greedy action selection strategy for future actions, it may not explore the environment sufficiently, leading to a bias towards exploiting known actions rather than exploring new ones.

The Q-learning update rule is based on the Bellman equation :

$$Q(s,a) = (1-\alpha) \cdot Q(s,a) + \alpha \cdot \left(r + \gamma \cdot \max_{a'} Q(s',a')\right)$$

The pseudo code for the Q-learning update method can be found in **Appendix A (A.4.1)**.

Q-Learning with Epsilon-Greedy Strategy

To improve our *Q-Learning algorithm* used previously, we will apply the *epsilon-greedy* exploration-exploitation balance strategy. On each episode, we will balance exploration and exploitation by alternating between picking greedy actions that are learned so far and random actions. By increasing epsilon, we make sure that our agent exponentially shifts from exploration to exploitation as it gains more experience. This approach avoids converging to a *sub-optimal* solution, preventing the agent from discovering more optimal policies.

We will set the following constants to test the performance of this improved approach :

— Initial epsilon (ϵ) : 0.9
— Epsilon decay rate : 0.999
— Minimum epsilon : 0.01

The Q-learning with epsilon-greedy strategy updates the Q-values using the Bellman equation just like the basic Q-learning algorithm :

$$Q(s,a) = (1-\alpha) \cdot Q(s,a) + \alpha \cdot \left(r + \gamma \cdot \max_{a'} Q(s',a')\right)$$

The pseudo code for Q-learning with the epsilon-greedy strategy can be found in **Appendix A (A.4.2)**.

Double Q-learning Algorithm

To address the issue of overestimation of Q-values, we will use *Double Q-learning*. In fact, instead of using a single Q-function to estimate the Q-values, we will use two separate Q-functions. During the training, one Q-function is used to select the best action, while the other one is used to evaluate the Q-value of that action. This way, we mitigate the overestimation bias present in the basic method, resulting in convergence towards a more accurate policy.

The update method for Double Q-learning involves updating one Q-function using the Bellman equation while using the other Q-function to determine the best action. The Bellman equation for Double Q-learning is :

$$Q_i(s,a) = (1-\alpha) \cdot Q_i(s,a) + \alpha \cdot (r + \gamma \cdot Q_{1-i}(s', \text{argmax}_{a'} Q_i(s',a')))$$

This equation represents how the Q-value of the i-th Q-function is updated using the Bellman equation in Double Q-learning.
The pseudo code for Double Q-learning can be found in **Appendix A (A.4.3)**.

4.2.5 Implementing Monte Carlo Algorithms

Monte Carlo methods, include : *first-visit* and *every-visit* methods. The main difference between these methods is in the way they handle repeated visits to the same state-action pair within an episode.

First-visit Monte Carlo Method

To address the issue of estimating state values without using a model, we will use the *First-Visit Monte Carlo* algorithm. In First-Visit Monte Carlo, we estimate the value of each state by averaging the returns observed after the first visit to that state during an episode. This method averages over multiple episodes to obtain a more accurate estimation of state values, making it suitable for environments where we can simulate episodes without a model.

The pseudo code for First-Visit Monte Carlo can be found in **Appendix A (A.5.1)**.

Every-visit Monte Carlo Method

To further explore methods for estimating state values in reinforcement learning without relying on a model, we introduce the *Every-Visit Monte Carlo* algorithm. Unlike the First-Visit

Monte Carlo algorithm, which considers only the first visit to each state within an episode, Every-Visit Monte Carlo computes the value of each state by averaging the returns obtained from all visits to that state across all episodes. This approach provides a comprehensive estimation of state values and can be particularly useful in environments with non-deterministic dynamics. T

The pseudo code detailing the implementation of Every-Visit Monte Carlo can be found in **Appendix A (A.5.2)**.

4.2.6 Implementing Multi-armed Bandits Algorithm

To explore the *exploration-exploitation* trade-off in the context of *multi-armed bandit* problems, we employ the epsilon-greedy algorithm. This algorithm balances between exploring new bandits and exploiting the known bandits' rewards. At each step, it randomly selects a bandit with a probability determined by epsilon and otherwise selects the bandit with the highest estimated value. By adjusting the value of epsilon, we control the degree of exploration versus exploitation.

To model the multi-armed bandits in our simulation, we represent them as arms in a computer model, where each arm corresponds to a bandit. Our goal is to estimate the reward probabilities associated with each bandit. The k-armed bandit is modeled by an array with length k containing the probability of winning of each bandit.

The hyperparameters we selected for our simulation are as follows :
— Number of bandits : 5
— True probabilities of each bandit : Randomly generated
— Initial epsilon : 0.99
— Minimum epsilon : 0.01
— Epsilon decay rate : 0.999

These hyperparameters control the behavior of the epsilon-greedy algorithm, influencing the balance between exploration and exploitation. For instance, a higher initial epsilon value encourages more exploration, while a lower minimum epsilon value ensures that the algorithm gradually focuses more on exploitation as it learns. The epsilon decay rate determines the rate at which epsilon decreases over time, allowing the algorithm to transition from exploration to exploitation smoothly.

The pseudo code detailing the implementation of the epsilon-greedy algorithm for multi-armed bandits can be found in **Appendix A (A.6)**.

4.3 Bechmarking and Performance Evaluation

In this section, we will conduct benchmarking across all our algorithms. Additionally, we will evaluate the performance of model-free approaches. By this study, we aim to verify the effectiveness and efficiency of our theoretical studies in chapter 1. By benchmarking our algorithms, we seek to gain insights into their strengths, weaknesses, and suitability for different applications, ultimately advancing our understanding and application of Reinforcement Learning techniques.

4.3.1 Comparative Benchmarking of Policy Improvement VS. Value Iteration

TABLE 4.1 – Average Convergence Time for Policy Improvement and Value Iteration

Method	**Average Time (seconds)**
Policy Improvement	0.037
Value Iteration	0.0035

Our benchmarking results demonstrate that *Value Iteration* outperforms *Policy Improvement* in terms of convergence time on an Intel Core i7 11^{th} generation processor. With an average time of 0.037 s, Policy Improvement exhibits a significantly longer convergence time compared to Value Iteration, which achieves convergence in just 0.0035 s on average. These averages were calculated over 10 repetitions, providing a reliable assessment of the methods' performance. The superior efficiency of *Value Iteration* makes it a more favorable choice for scenarios where rapid convergence is crucial.

4.3.2 Benchmark Study of Model-free Algorithms

We conducted a benchmark study to evaluate the runtime performance of model-free reinforcement learning algorithms on the same Intel Core i7 11^{th} generation processor. Each algorithm was run 10 times, and the average time taken for execution was recorded. The results are summarized in Table 4.2.

Algorithm	**Average Time (seconds)**
Multi-Armed Bandits	0.23
First Visit Monte Carlo	88.24
Every Visit Monte Carlo	88.89
SARSA	86.75
Q-Learning	92.40
Double Q-Learning	116.12
Expected SARSA	121.35
Q-Learning with Epsilon Greedy	2088.28

TABLE 4.2 – Average runtime of reinforcement learning algorithms

The results reveal that SARSA and Q-Learning algorithms exhibit similar runtime performance, with SARSA being slightly faster on average. Double Q-Learning and Expected SARSA

algorithms take longer to execute compared to SARSA and Q-Learning. This is expected as these algorithms involve additional computations to maintain and update two Q-functions or compute expected values.

Monte Carlo methods, both first visit and every visit, show comparable runtime performance, indicating that the difference in computational complexity between these methods is minimal. However, the Q-Learning algorithm with Epsilon Greedy strategy demonstrates significantly higher runtime. This is due to the exploration-exploitation trade-off inherent in the epsilon-greedy strategy, which requires additional computations to balance exploration and exploitation during learning.

These runtime results provide insights into the computational efficiency of different reinforcement learning algorithms, which can be crucial for real-world applications where computational resources are limited.

4.3.3 Performance Evaluation of Model-free Algorithms

Evaluating the performance of the Frozen Lake Agent

We conducted an evaluation by playing 10,000 games and calculating the average winning time for each training algorithm. Our agents were initially trained on 1,000,000 episodes to learn optimal policies. The results are summarized in Table 4.3.

Algorithm	**Average Winning Time (%)**
Random Policy	1.34
SARSA	71.043
Q-Learning	0.0
Double Q-Learning	0.0
Expected SARSA	46.64
First Visit Monte Carlo	73.72
Every Visit Monte Carlo	73.12
Q-Learning with Epsilon Greedy	0.0
Q-Learning (trained on 2000 episodes)	73.78
Double Q-Learning (trained on 1000 episodes)	74.28
Q-Learning with Epsilon Greedy (trained on 2000 episodes)	72.78

TABLE 4.3 – Performance of training algorithms based on average winning time

The evaluation indicates that SARSA, First Visit Monte Carlo, and Every Visit Monte Carlo algorithms achieved high average winning times, demonstrating their effectiveness in learning optimal policies. SARSA showed a respectable performance, maintaining a high success rate of 71.043%. First Visit Monte Carlo and Every Visit Monte Carlo exhibited even better performance, with average winning times exceeding 73%. Interestingly, both Monte Carlo methods showed a tendency to stay in the first positions, maximizing their immediate reward but failing to explore further and potentially obtain a higher long-term reward.

On the other hand, Q-Learning, Double Q-Learning, and Q-Learning with Epsilon Greedy exhibited suboptimal performance, with average winning times close to zero. This indicates a convergence to suboptimal solutions and that our model is over-trained. The visualisation of the policies learned by the agent, shows that it tends to avoid exploring new actions and prefer staying in their initial states maximizing their long-term reward by avoiding to lose. However, when Q-Learning was trained on 2000 episodes and Double Q-Learning on 1000 episodes, their performance improved significantly, demonstrating the impact of training duration on Q-learning algorithm effectiveness.

Analysis of Average Reward Evolution of Q-learning Algorithms

In this experiment, we compare the performance of three reinforcement learning algorithms : *Q-learning epsilon-greedy*, *Q-learning*, and *Double Q-learning*. Each algorithm is trained for one million episodes on Frozen Lake environment. The goal is to analyze how these algorithms learn and adapt over an extensive training period, as reflected in their average reward per episode. By examining their learning dynamics and performance differences, we aim to gain insights into the effectiveness of different exploration strategies and their impact on overall learning outcomes.

Figure 4.3 illustrates the evolution of average reward per episode for the three algorithms.

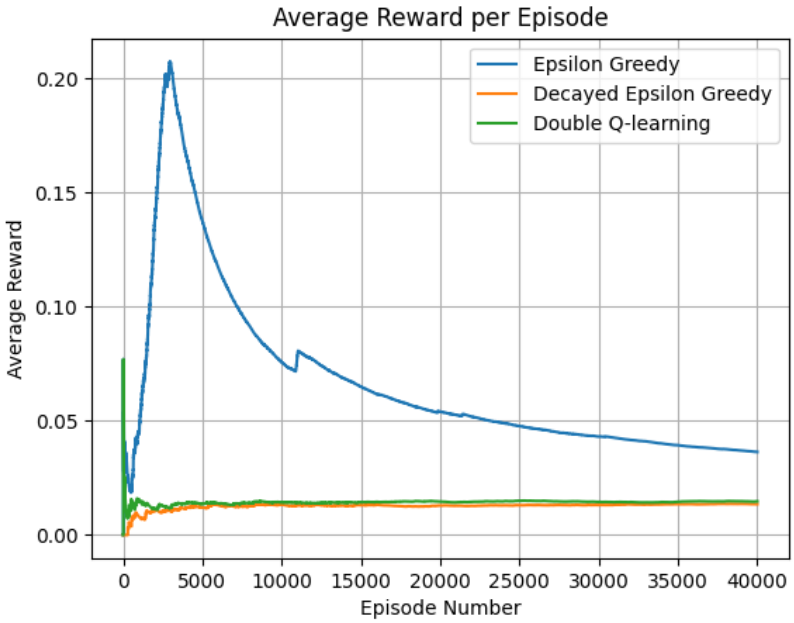

FIGURE 4.3 – Evolution of average reward per episode

Initially, all algorithms undergo exploration of the environment, resulting in a gradual increase in average rewards. Notably, the Q-learning epsilon-greedy algorithm demonstrates the most substantial improvement, with its average reward steadily increasing throughout the episodes. For instance, at episode **1000**, the average reward stands at approximately **0.003**. The culmination of its performance is evident at episode **2738**, where the average reward reaches its peak, with a value of approximately **0.207**. However, following this peak, the average reward

for Q-learning epsilon-greedy starts to decline, indicating convergence towards *sub-optimal* policies. This decline suggests that the agent may have started to exploit certain actions or states that yield immediate but *sub-optimal* rewards, ultimately hindering further improvement in performance.

In contrast, both the *standard Q-learning* and *Double Q-learning algorithms* exhibit a different pattern. The average rewards for these algorithms remain consistently low and almost constant throughout the training period. This pattern suggests that, unlike *Q-learning epsilon-greedy*, which balances exploration and exploitation, these algorithms predominantly focus on exploration without effectively leveraging the knowledge gained. The consistently low average rewards indicate that while exploration is occurring, there is no exploitation of the learned knowledge, leading to stagnation in performance.

While Q-learning epsilon-greedy demonstrates significant improvement in average rewards, it's crucial to note that "overtraining" the model may also have disadvantages. Our experiments indicate that the optimal number of episodes to train the model in our environment is around 3000, where we observe the most desired results. Beyond this point, the average rewards may start to decline as the model overfits to the training data or converges to sub-optimal policies. Therefore, striking a balance between exploration and exploitation within a suitable training duration is essential for achieving optimal performance in reinforcement learning tasks.

Evaluating the Performance Multi-armed Bandits Simulation

After 1 million iterations, the multi-armed bandit algorithm demonstrated remarkable efficiency in selecting the three bandits with the highest probabilities. The algorithm's exploration-exploitation balance allowed it to focus primarily on the more rewarding options. This efficient selection process highlights the effectiveness of the chosen hyperparameters in guiding the algorithm towards optimal decision-making.

The probabilities of each bandit are provided in Table 4.4.

Bandit	**Probability**
Bandit 1	0.91
Bandit 2	0.93
Bandit 3	0.98
Bandit 4	0.23
Bandit 5	0.08

TABLE 4.4 – Probabilities of multi-armed bandits

For a visual representation of the bandit selection process and the convergence towards the optimal bandits, we refer to Figure 4.4.

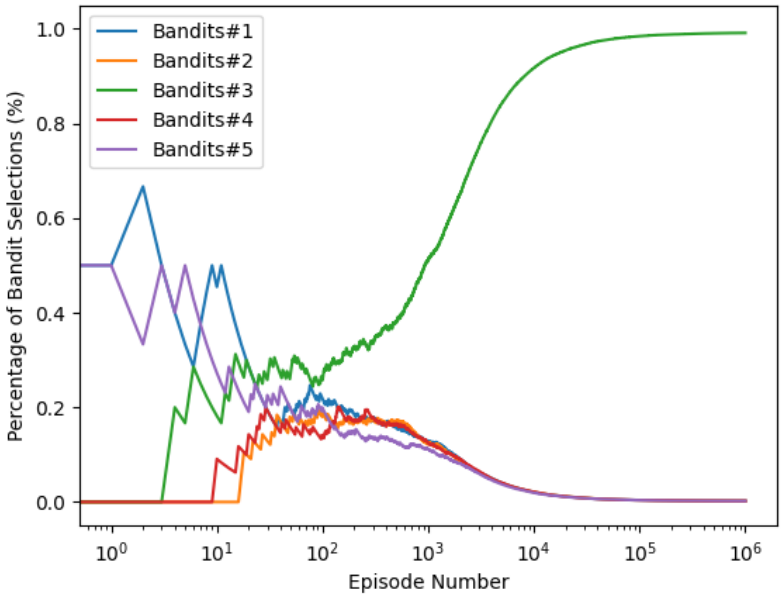

FIGURE 4.4 – Evolution of Bandits Selection.

This plot provides insight into how the algorithm's selections evolve over time, demonstrating its ability to identify and exploit the highest-probability bandits.

Conclusion

Model-based algorithms exhibit remarkable performance, swiftly converging to the optimal policy with relatively fewer training episodes. Our experiments underscored their efficiency and ability to learn rapidly. However, despite their effectiveness, the practicality of model-based approaches may be limited in scenarios where computational efficiency is paramount. While they offer fast convergence, their reliance on explicit models can restrict their applicability, particularly when compared to model-free methods.

On the other hand, our experiments highlight the importance of selecting appropriate hyperparameters in reinforcement learning algorithms. We observed that different algorithms can achieve remarkably close performances when tuned correctly. For instance, while Sarsa demands a high number of training episodes to converge, Q-learning epsilon-greedy demonstrates the opposite behavior, achieving optimal performance with fewer training iterations. Despite its complexity, this approach proves advantageous as it accelerates training, saving time and computational resources. Thus, the judicious selection of hyperparameters plays a crucial role in maximizing the efficiency and effectiveness of reinforcement learning algorithms.

General Conclusion

In this project, we aimed to explore the world of Reinforcement Learning and address the various methods and the ability of RL to solve complex decision-making problems. We discussed the fundamental concepts of RL, including the agent, environments, policy, states, actions, and rewards. Additionally, we introduced a various range of Reinforcement Learning techniques, from tabular techniques in Reinforcement Learning such as multi-armed bandits, finite Markov processes, dynamic programming, temporal difference learning and Monte Carlo methods. These methods are the best for problems with discrete states. Policy iteration and value iteration are different examples of those strategies. Policy iteration improves an initial policy through alternately comparing and enhancing it till convergence, while value iteration computes the optimal value function via iterative updates. These methods provide simplicity and guarantees ; however, they face scalability challenges in large or continuous state and action spaces because of their tabular representation. We also introduced Approximation Techniques in Reinforcement Learning, which included On-policy prediction, on-policy control, and off-policy methods. These types of methods provide scalability to deal with massive or continuous states and action spaces. Instead of tabular representations, they use function approximation strategies like neural networks or linear functions. By mastering compact representations of value functions or policies, they allow RL algorithms to address high-dimensional input spaces efficiently. However, approximation mistakes can pose challenges in convergence and stability. The integration of generative AI impacts Reinforcement Learning by helping in mere data generation and objective maximization tasks. In RL for mere generation, tools like Generative Adversarial Networks (GANs) help us to create diverse and realistic data, improving the agent's learning process and efficiency. Also, combining RL with generative models allows agents to make maximization tasks easier, for example, optimizing non-quantifiable features like creativity or aesthetics.

However, training huge models like the Large Language Models (LLMs) or open-world video games presents a huge difficulty because single computers, even those with great overall performance capabilities, can hardly do the task. This implies that it can only be accomplished by large IT companies with access to supercomputers and several GPU processors. They have so benefited greatly from the development of cutting-edge AI technology. Smaller companies and people without access to those kinds of resources find it difficult to compete in this race. Encouraging collaboration across various businesses within the AI field and making

those sources more available are imperative if we are to ensure that everyone has an honest chance of contributing to AI developments. In this manner, we are able to make certain that the development of AI isn't always simply constrained to massive businesses and that everyone can take advantage from it. In conclusion, Reinforcement Learning is an exponentially increasing domain within the field of artificial intelligence. With the introduction of new techniques and algorithms, RL continues to expand, making its way to new opportunities across multiple domains. From a general perspective, RL is being adopted in a lot of sectors such as robotics, finance, and healthcare. Companies are now exploring the services offered by RL to automate complex tasks and find optimal solutions.

APPENDIX A

Tabular-solution Algorithms Pseudo-code

In this appendix, we will present our pseudo-code to all the algorithms we have covered. Ideally, we don't want any part of the main content of our work to become obsolete in the future. Technology changes, standards change and eventually the code could become unrunnable. That's why we will be presenting pseudo-code instead of the actual implementation. Pseudo-code provides a high-level representation of the algorithm's logic without being tied to a specific programming language or implementation details. However, for those interested in the full implementation, the Python code can be found on GitHub at the following link :

`https://github.com/Mahdyy02/reinforcement-learning-implementation`

By separating the pseudo-code from the main content, we ensure that this work remains relevant and adaptable to changes in technology and standards over time.

A.1 Model-based RL Algorithms

A.1.1 Policy Improvement Implementation

```
1
2  FUNCTION compute_state_value(state)
3      # Estimate the state's worth
4      IF state equals 47 THEN
5          RETURN 0
6      END IF
7      action = policy[state]
8      FOR each state_tuple in env.unwrapped.P[state][action]
9          probability, next_state, reward, info = state_tuple
10         # Recursively compute the value of the next state
11         RETURN reward + gamma * compute_state_value(next_state)
12     END FOR
13 END FUNCTION
```

Listing A.1 – State Value Computation

```
FUNCTION compute_q_value(state, action)
    # Calculate Q-value function also known as action-value function
    IF state equals terminal_state THEN
        RETURN 0
    END IF
    # Get the transition information for the given state-action pair
    prob, next_state, reward, info = env.unwrapped.P[state][action][0]
    # Compute the Q-value for the next state recursively
    RETURN reward + gamma * compute_state_value(next_state)
END FUNCTION

# Initialize an empty dictionary to store Q-values
Q = empty dictionary

# Compute Q-values for all state-action pairs
FOR each state in range(env.observation_space.n)
    FOR each action in range(env.action_space.n)
        Q[(state, action)] = compute_q_value(state, action)
    END FOR
END FOR

# Convert dictionary keys to strings for JSON serialization
data_str_keys = {json.dumps(k): v for k, v in Q.items()}
# Write the Q-values to a file in JSON format
json.dump(data_str_keys, file, indent=4)
```

Listing A.2 – Q-Value Computation and Storage

```
FUNCTION policy_improvement(policy)
    # Open a file to store the improved policy
    improved_policy_file = open("improved_policy.json", "w")

    # Initialize an empty dictionary for the improved policy
    improved_policy = {state: 0 for state in range(env.observation_space.n)}

    # Initialize a dictionary to store Q-values for all state-action pairs
    Q = empty dictionary

    # Iterate over all states
    FOR each state in range(env.observation_space.n)
        # Find the action with the maximum Q-value for the current state
        max_action = action that maximizes Q-value for current state

        # Update the improved policy with the action that maximizes Q-value
        improved_policy[state] = max_action
    END FOR

    ## Write the improved policy to the file in JSON format
    Write improved_policy to improved_policy_file in JSON format
    # Return the improved policy
    RETURN improved_policy
END FUNCTION
```

Listing A.3 – Loading Improved Policy

A.1.2 Value Iteration

```
FUNCTION value_iteration():
    # Initialize value function v and policy
    v = {state: 0 for state in range(env.observation_space.n)}
    policy = {state: 0 for state in range(env.observation_space.n - 1)}
    threshold = 0.001

    # Open files to store results
    value_iteration_improved_policy_file = open("
   value_iteration_improved_policy.json", "w")
    value_iteration_improved_value_function_file = open("
   value_iteration_improved_value_function.json", "w")

    # Perform value iteration until convergence
    WHILE True DO:
        # Initialize a new value function
        new_v = {state: 0 for state in range(env.observation_space.n)}

        # Iterate over all states
        FOR state IN range(env.observation_space.n - 1) DO:
            # Find the action with the maximum Q-value
            max_action, max_q_value = get_max_action_and_value(state, v)

            # Update the value function and policy
            new_v[state] = max_q_value
            policy[state] = max_action

        # Check for convergence
        IF all(abs(new_v[state] - v[state]) < threshold for state in range(
   env.observation_space.n)) THEN
            BREAK // Exit the loop if convergence is achieved

        # Update the value function
        v = new_v

    # Return the optimal policy and value function
    RETURN policy, v
END FUNCTION
```

Listing A.4 – Value Iteration

A.2 Training Loop

```
FUNCTION training_loop(num_episodes):
    # Loop over the specified number of episodes
    FOR episode IN range(num_episodes):
        # Reset environment to start a new episode
        state = env.reset()
        # Select an action randomly
        action = env.random_action()
        # Initialize episode termination flag
        done = False
```

```
10
11          # Loop until episode termination
12          WHILE NOT done DO:
13              # Take action and observe next state and reward
14              next_state, reward, done = env.step(action)
15              # Select next action randomly
16              next_action = env.random_action()
17              # Update Q-table based on observed transition
18              Q[state, action] = (1 - alpha) * Q[state, action] + alpha * (
    reward + gamma * Q[next_state, next_action])
19              # Update current state and action for next iteration
20              state, action = next_state, next_action
21      END FOR
22  END FUNCTION
```

Listing A.5 – Training Loop

In our approach, we aim to maintain the training loop structure while adapting each `update_q_table` function to its corresponding algorithm. This strategy allows us to preserve the integrity of the training process while accommodating the unique requirements and updates of each algorithm. By isolating the algorithm-specific logic within the `update_q_table` function, we ensure modularity and flexibility in our implementation. This approach also streamlines the development process, making it easier to manage and debug individual components. As a result, we can efficiently experiment with different algorithms without overhauling the entire training pipeline, enabling rapid iteration and exploration of our different algorithms.

A.3 SARSA Algorithms

A.3.1 Basic SARSA Q-value Update Function

```
1 FUNCTION update_q_table(state, action, reward, next_state, next_action)
2     old_value = Q[state, action]  % Old Q-value of the current state-action
    pair
3     next_value = Q[next_state, next_action]  % Q-value of the next state-
    action pair
4     Q[state, action] = alpha * (reward + gamma * next_value) + (1 - alpha) *
      old_value  % Update Q-value using SARSA update rule
5 END FUNCTION
```

Listing A.6 – SARSA Update

A.3.2 Expected SARSA Updating Q-value Update Function

```
1 FUNCTION update_q_table(state, action, reward, next_state)
2     expected_q = mean(Q[next_state])  % Compute the expected value of Q-
    values for the next state
3     Q[state, action] = (1 - alpha) * Q[state, action] + alpha * (reward +
    gamma * expected_q)  % Update Q-value using Expected SARSA update rule
4 END FUNCTION
```

Listing A.7 – Expected SARSA Update

A.4 Q-learning Algorithms

A.4.1 Basic Q-learning Algorithm Q-value Update Function

```
FUNCTION update_q_table(state, action, reward, next_state)
    old_value = Q[state, action]  % Old Q-value of the current state-action pair
    next_max = max(Q[next_state])  % Maximum Q-value in the next state
    Q[state, action] = alpha * (gamma * next_max + reward) + (1 - alpha) * old_value  % Update Q-value using Q-learning update rule
END FUNCTION
```

Listing A.8 – Q-learning Update

A.4.2 Q-learning Epsilon-greedy Q-value Update Function

```
# Define a function for epsilon-greedy action selection
FUNCTION epsilon_greedy(state):
    # With probability epsilon, select a random action
    IF rand() < epsilon THEN
        action = sample_action()  # Select a random action
    ELSE:
        action = argmax(Q[state, :])  # Select the action with the highest Q-value
    END IF
    RETURN action

# Define a function for Q-learning with an epsilon-greedy strategy
FUNCTION q_learning_epsilon_greedy_loop(num_episodes):
    # Loop over the specified number of episodes
    FOR episode IN range(num_episodes):
        # Reset the environment to start a new episode
        state, info = reset_environment()
        done = False

        # Loop until the episode is done
        WHILE NOT done DO:
            # Select an action using epsilon-greedy strategy
            action = epsilon_greedy(state)
            # Take a step in the environment
            next_state, reward, terminated, truncated, info = step_environment(action)
            # Update the Q-table based on the observed transition
            update_q_table(state, action, reward, next_state)
            # Update the current state for the next iteration
            state = next_state
            # Check if the episode is terminated or truncated
            done = terminated OR truncated

        # Update epsilon for the next episode
        epsilon = max(min_epsilon, epsilon * epsilon_decay)
END FUNCTION
```

Listing A.9 – Q-Learning with Epsilon-Greedy Strategy

A.4.3 Double Q-learning Algorithms Q-value Update Function

```
FUNCTION update_q_table(state, action, reward, next_state):
    # Randomly choose one of the two Q-functions
    i = randomly_choose(0, 1)
    # Determine the best action using the selected Q-function
    best_next_action = determine_best_action(Q[i][next_state, :])
    # Update the Q-value for the chosen action
    Q[i][state, action] = (1 - alpha) * Q[i][state, action] + alpha * (
   reward + gamma * Q[1 - i][next_state, best_next_action])
END FUNCTION
```

Listing A.10 – Double Q-Learning

A.5 Monte Carlo Algorithms Implemented

A.5.1 First-visit Monte Carlo Implementation

```
# Function for first-visit Monte Carlo method
FUNCTION first_visit_mc(num_episodes):
    # Initialize Q-table, returns_sum, and returns_count arrays
    Q = ZEROS(num_states, num_actions)
    returns_sum = ZEROS(num_states, num_actions)
    returns_count = ZEROS(num_states, num_actions)

    # Loop over the specified number of episodes
    FOR i FROM 1 TO num_episodes DO:
        # Generate an episode
        episode = generate_episode()
        # Initialize an empty set to track visited state-action pairs
        visited_states_actions = EMPTY_SET()

        # Loop over each (state, action, reward) tuple in the episode
        FOR EACH (state, action, reward) IN episode DO:
            # Check if the state-action pair has been visited for the first
   time in the episode
            IF (state, action) NOT IN visited_states_actions THEN:
                # Update the sum of returns and count of visits for the
   state-action pair
                returns_sum[state, action] += SUM(reward FOR EACH (state,
   action, reward) IN episode FROM CURRENT INDEX)
                returns_count[state, action] += 1
                # Add the state-action pair to the set of visited pairs
                visited_states_actions.ADD((state, action))

    # Calculate the average returns for each state-action pair and update
   the Q-table
    FOR EACH (state, action) IN returns_count DO:
        IF returns_count[state, action] != 0 THEN:
            Q[state, action] = returns_sum[state, action] / returns_count[
   state, action]

    # Save the Q-table to a JSON file
```

```
    SAVE Q TO JSON FILE

    # Return the updated Q-table
    RETURN Q
END FUNCTION
```

Listing A.11 – First-visit Monte Carlo

A.5.2 Every-visit Monte Carlo Implementation

```
# Function for every-visit Monte Carlo method
FUNCTION every_visit_mc(num_episodes):
    # Initialize Q-table, returns_sum, and returns_count arrays
    Q = ZEROS(num_states, num_actions)
    returns_sum = ZEROS(num_states, num_actions)
    returns_count = ZEROS(num_states, num_actions)

    # Loop over the specified number of episodes
    FOR i FROM 1 TO num_episodes DO:
        # Generate an episode
        episode = generate_episode()

        # Loop over each (state, action, reward) tuple in the episode
        FOR EACH (state, action, reward) IN episode DO:
            # Update the sum of returns and count of visits for the state-action pair
            returns_sum[state, action] += SUM(reward FOR EACH (state, action, reward) IN episode FROM CURRENT INDEX)
            returns_count[state, action] += 1

    # Calculate the average returns for each state-action pair and update the Q-table
    FOR EACH (state, action) IN returns_count DO:
        IF returns_count[state, action] != 0 THEN:
            Q[state, action] = returns_sum[state, action] / returns_count[state, action]

    # Save the Q-table to a JSON file
    SAVE Q TO JSON FILE

    # Return the updated Q-table
    RETURN Q
END FUNCTION
```

Listing A.12 – Every-visit Monte Carlo

A.6 Multi-armed Bandits Implemented

```
FUNCTION epsilon_greedy():
    r = random_uniform()
    IF r < epsilon THEN
        arm = random_integer(0, n_bandits)
    ELSE
        arm = argmax(values)
    END IF
    RETURN arm
END FUNCTION

n_bandits = 5
true_bandit_probs = random_uniform(n_bandits)
n_iterations = 500000
epsilon = 0.99
min_epsilon = 0.01
epsilon_decay = 0.999

counts = zeros(n_bandits)
values = zeros(n_bandits)
rewards = zeros(n_iterations)
selected_arms = zeros(n_iterations)

FOR i = 1 TO n_iterations DO:
    arm = epsilon_greedy()
    reward = random_uniform() < true_bandit_probs[arm]
    rewards[i] = reward
    selected_arms[i] = arm
    counts[arm] += 1
    values[arm] += (reward - values[arm]) / counts[arm]
    epsilon = max(min_epsilon, epsilon * epsilon_decay)

FOR i = 1 TO n_bandits DO:
    PRINT("Bandit#" + i + " -> " + true_bandit_probs[i]:.2f)
END FOR
```

Listing A.13 – Every-visit Monte Carlo

References

1. Yu, L., Zhang, W., Wang, J., Yu, Y. (2016). "SeqGAN : Sequence Generative Adversarial Nets with Policy Gradient."
2. Franceschelli, G., Musolesi, M. (2023). "Reinforcement Learning for Generative AI : State of the Art, Opportunities and Open Research Challenges."
3. Le, H., Wang, Y., Gotmare, A. D., Savarese, S., Hoi, S. C. H. (2022). "CodeRL : Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning."
4. Christiano, P. F., Leike, J., Brown, T. B., Martic, M., Legg, S., & Amodei, D. (2023). Deep Reinforcement Learning from Human Preferences. OpenAI, DeepMind.
5. Tambwekar, P., Dhuliawala, M., Martin, L. J., Mehta, A., Harrison, B., & Riedl, M. O. (2023). Controllable Neural Story Plot Generation via Reward Shaping. School of Interactive Computing, Georgia Institute of Technology, Department of Computer Science, University of Kentucky.
6. Christiano, P., Leike, J., Brown, T. B., Martic, M., Legg, S., Amodei, D. (2017). "Deep Reinforcement Learning from Human Preferences."
7. Ramamurthy, R., Ammanabrolu, P., Brantley, K., Hessel, J., Sifa, R., Bauckhage, C., Hajishirzi, H., Choi, Y. (2022). "Is Reinforcement Learning (Not) for Natural Language Processing : Benchmarks, Baselines, and Building Blocks for Natural Language Policy Optimization."
8. Sutton, R. S., Mahmood, A. R., White, M. (2016). "An Emphatic Approach to the Problem of Off-policy Temporal-difference Learning." Journal of Machine Learning Research.
9. Howard, R. (1960). "Dynamic Programming and Markov Processes." MIT Press, Cambridge, MA.
10. Barto, A. G., Bradtke, S. J., Singh, S. P. (1995). "Learning to Act Using Real-time Dynamic Programming." Artificial Intelligence.
11. Hugging Face.
12. OpenAI Gymnasium. (2017).
13. OpenAI ChatGPT-3.5. (2023).

14. Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning : An Introduction.* MIT Press.
15. Towards Data Science. *Solving the Multi-Armed Bandit Problem.*