



Investigating Behavioural Intelligibility In Reinforcement Learning

J Fouché



[orcid.org/ 0009-0004-1005-7489](https://orcid.org/0009-0004-1005-7489)

Dissertation accepted in fulfilment of the requirements for the degree
Master of Engineering in Computer and Electronic Engineering
at the North-West University

Supervisor: Prof PA van Vuuren

Graduation: May 2026

Abstract

In this dissertation, an investigation is launched into the fundamentals of reinforcement learning, aiming to answer the question: *How can first-principles implementations of RL agents enhance our understanding of their behaviour and improve intelligibility in controlled environments?*

This investigation is addressed by outlining the overarching mathematical concepts, the theoretical agent framework, the structure of the controlled environment, and the intelligibility methods employed.

Subsequently, four selected agents—PPO, Q-Learning, REINFORCE, and DDQN—are implemented using transparent, first-principles approaches. Each implementation is described in detail. The agents are then deployed within the environment, trained, and experimentally evaluated with respect to key features of their decision-making and learning behaviour. These experiments demonstrate that, through these investigative techniques, it is possible to observe and confirm theoretical aspects of the agents in a practical and transparent manner, thereby contributing to answering the research question.

In conclusion, by uniting the mathematical concepts with the theoretical understanding of the agents and presenting them in a transparent manner, one can train agents in a controlled environment and investigate them using intelligibility techniques. This approach provides significantly improved insight into the behaviour of these agents and lays the groundwork for future tests to be conducted on these fundamental implementations.

Keywords: Reinforcement Learning; Intelligibility; Interpretability; Explainability; First-Principles Implementations; Agent Behaviour Analysis; PPO; Q-Learning; REINFORCE; DDQN.

Contents

1	Introduction	1
1.1	Background and motivation	1
1.2	Scope and limitations	2
1.3	Research objectives	3
1.4	Structure of the report	4
2	Foundations of reinforcement learning	6
2.1	Markov decision processes	7
2.2	Policy	8
2.3	Value function	8
2.4	Bellman equations	9
2.5	Dynamic programming	10
2.6	Monte Carlo methods	10
2.7	Temporal difference learning	11
2.8	Policy gradient methods	11
2.9	Policy gradient theorem	12
2.10	Advantage functions	13
2.11	Entropy regularization	13
2.12	Exploration vs exploitation	14
2.13	Conclusion	16

3	Reinforcement learning algorithm taxonomy	17
3.1	Model-Free vs Model-Based	19
3.2	On-Policy vs Off-Policy	20
3.3	Value-Based vs Policy-Based	23
3.4	Overview of selected algorithms	24
3.4.1	REINFORCE	24
3.4.2	Q-Learning	25
3.4.3	Double Deep Q-Network (DDQN)	25
3.4.4	Proximal Policy Optimization (PPO)	25
3.4.5	Selection justification	25
3.5	Conclusion	26
4	Experimental setup	27
4.1	Environments used	27
4.1.1	Multi-Armed Bandit	27
4.1.2	Taxi	28
4.1.3	Frozen Lake	29
4.1.4	CartPole	31
4.1.5	Snake	32
4.2	Experimental method	33
4.3	Conclusion	34
5	Behavioural analysis and intelligibility	35
5.1	Intelligibility	35
5.2	Intelligibility methods	36
5.2.1	Performance metrics	36
5.2.2	Policy visualization	38

5.2.3	State visitation	39
5.2.4	Decision boundaries	39
5.2.5	Consistency	41
5.2.6	Feature heatmaps	42
5.3	Conclusion	43
6	Agent implementations	45
6.1	Code design	45
6.2	Agents	51
6.2.1	REINFORCE	52
6.2.2	Q-Learning	58
6.2.3	DDQN	61
6.2.4	PPO	67
6.3	Conclusion	73
7	Experimental design and execution	75
7.1	Implementation validation	76
7.1.1	Experimental question	76
7.1.2	Hypothesis	76
7.1.3	Methodology	76
7.1.4	Results	78
7.1.5	Conclusion	78
7.2	Policy visualization	79
7.2.1	Experimental question	79
7.2.2	Hypothesis	79
7.2.3	Methodology	79
7.2.4	Results	81

7.2.5	Conclusion	88
7.3	Learning rate impact	88
7.3.1	Experimental question	88
7.3.2	Hypothesis	88
7.3.3	Methodology	89
7.3.4	Results	90
7.3.5	Conclusion	91
7.4	Exploration consistency	92
7.4.1	Experimental question	92
7.4.2	Hypothesis	92
7.4.3	Methodology	93
7.4.4	Results	94
7.4.5	Conclusion	95
7.5	State visitation analysis	96
7.5.1	Experimental question	96
7.5.2	Hypothesis	96
7.5.3	Methodology	97
7.5.4	Results	99
7.5.5	Conclusion	103
7.6	Feature selection experiment	104
7.6.1	Experimental question	104
7.6.2	Hypothesis	104
7.6.3	Methodology	104
7.6.4	Results	106
7.6.5	Conclusion	110
7.7	Example	112

7.8	Conclusion	113
8	Conclusion	115
8.1	Summary	115
8.2	Main question	116
8.3	Future research directions	117
8.4	Final conclusion	118
A	Code	126

List of Figures

2.1	Markov Decision Processes	7
2.2	Comparison of Exploration Strategies	15
3.1	Overview of RL Categories	18
3.2	Graph of Reinforcement Learning Algorimt's Relative Popularity	19
4.1	Multi-Armed Bandit example reward distribution for 10 arms	28
4.2	Taxi Environment	29
4.3	Frozen Lake Environment	30
4.4	CartPole Environment	31
4.5	Snake Environment	32
5.1	Performance analyser output example from the fund_rl library	37
5.2	Policy visualisation output example from the analysis conducted using the fund_rl library	38
5.3	State visitation output example from the analysis conducted using the fund_rl library	40
5.4	Decision boundary output example from the analysis performed using the fund_rl library	41
5.5	Consistency analysis example using the fund_rl library	42
5.6	PCA heatmap output example generated using the fund_rl library	43
5.7	Feature heatmap output example generated using the fund_rl library	44

6.1	Example implementation using <code>fund_rl</code> to train a Q-learning agent on the <code>Taxi-v3</code> environment.	47
6.2	Flow diagram matching the implementation steps in the Q-learning example.	47
6.3	Combined flow of the training and evaluation functions.	48
6.4	Flow of the <code>Run</code> function.	50
6.5	Abstract base class <code>TAgent</code> used for implementing individual agents.	51
6.6	Class diagram of the REINFORCE agent showing its main components and their relationships.	53
6.7	Flow diagram illustrating the action selection process in the REINFORCE agent, showing how actions are chosen based on the current state and policy.	54
6.8	Flow diagram of the update procedure for the REINFORCE agent, illustrating how the policy is updated based on observed transitions and rewards.	55
6.9	Flow diagram showing the training process of the REINFORCE agent, including initialization, episode loop, and policy updates.	57
6.10	Class diagram of the Q-learning agent.	58
6.11	Flow diagram of the action selection process in the Q-learning agent.	59
6.12	Flow diagram of the Q-value update procedure.	60
6.13	Class diagram of the DDQN agent, illustrating key parameters and structural components.	62
6.14	Flow diagram of the action selection process in the DDQN agent.	63
6.15	Flow diagram of the update procedure in the DDQN agent.	64
6.16	Flow diagram of the training procedure for the DDQN agent.	66
6.17	Class diagram of the PPO agent, illustrating its key parameters, memory components, and neural network architecture.	68
6.18	Flow diagram of the action selection process in the PPO agent.	69
6.19	Flow diagram of the update procedure in the PPO agent.	70
6.20	Training flow for the PPO agent.	74
7.1	Flow diagram of the implementation validation experiment.	77

7.2	Filtered reward curves for all agents in the FrozenLake environment.	78
7.3	Flow of the experimental procedure for policy visualization and agent training.	80
7.4	Policy visualization of the Random (masked) agent before training.	82
7.5	Initial policy visualization of Q-Learning and DDQN agents before training ($\epsilon = 1.0$).	82
7.6	Initial policy visualization of the REINFORCE agent before training. PPO exhibits a similar initial distribution.	83
7.7	Q-Learning agent policy after 500 training episodes ($\epsilon \approx 0.8$).	83
7.8	DDQN agent policy after 500 training episodes ($\epsilon \approx 0.01$).	84
7.9	REINFORCE agent policy after 500 training episodes.	85
7.10	PPO agent policy after 500 training episodes.	85
7.11	Final policy of the Q-Learning agent after 2000 episodes ($\epsilon \approx 0.24$).	86
7.12	Final policy of the DDQN agent after 2000 episodes.	86
7.13	Final policy of the REINFORCE agent after 2000 episodes.	87
7.14	Final policy of the PPO agent after 2000 training episodes.	87
7.15	Flow diagram of the learning rate impact experiment.	89
7.16	Performance comparison of DDQN and PPO agents under different learning rate settings: $0.1\times$, $1.0\times$, and $10\times$ the default rate.	91
7.17	Flow diagram of the exploration consistency experiment.	93
7.18	Exploration consistency of the REINFORCE agent for different entropy factors.	95
7.19	Exploration consistency of the PPO agent for different entropy factors. . . .	96
7.20	Flow diagram illustrating the state visitation experiment process.	98
7.21	State visitation distribution during training.	100
7.22	PCA-based feature heatmap of state visitation during training.	100
7.23	Feature heatmap distribution during training.	101
7.24	State visitation distribution during evaluation.	101
7.25	PCA feature heatmap across all evaluated states.	102

7.26	Feature distribution heatmap for all evaluated states.	102
7.27	PCA heatmap of the top 100 most visited states.	103
7.28	Feature heatmap of top 100 most visited states.	103
7.29	Flow diagram of the feature selection experiment procedure.	105
7.30	Initial decision boundary of the DDQN agent prior to training.	106
7.31	Initial decision boundary of the PPO agent prior to training.	107
7.32	Post-training decision boundary of the DDQN agent.	107
7.33	Post-training decision boundary of the PPO agent.	108
7.34	Evaluation performance of the DDQN agent over 100 episodes.	108
7.35	Evaluation performance of the PPO agent over 100 episodes.	109
7.36	Decision boundary of the PPO agent for position vs. pole angle.	109
7.37	Decision boundary of the PPO agent for position vs. angular velocity.	110
7.38	Decision boundary of the PPO agent for position vs. velocity.	110
7.39	Evaluation performance of the Q-Learning agent trained without the position feature over 100 episodes, demonstrating improved control due to reduced state space complexity.	110
7.40	Evaluation performance of the Q-Learning agent trained with all features over 100 episodes, showing lower performance compared to the feature-reduced agent due to larger state space complexity.	111
7.41	Performance of a simplified PPO agent using only six linear decision boundaries.	111
7.42	State visitation distribution of the REINFORCE agent over 2000 episodes. The agent rarely visits many states, indicating insufficient exploration.	112
7.43	Training results after modifying the environment to include negative rewards for falling into holes. The REINFORCE agent eventually achieves optimal performance after initial poor results.	113

Table of abbreviations

A table containing a list of abbreviations the will be used throughout text.

AI	Artificial Intelligence
RL	Reinforcement Learning
XAI	Explainable Artificial Intelligence
MDP	Markov Decision Process
DP	Dynamic Programming
TD	Temporal Difference
UCB	Upper Confidence Bound
Q-Learning	Q-Learning
SARSA	State-Action-Reward-State-Action
MC	Monte Carlo
PPO	Proximal Policy Optimization
A2C	Advantage Actor-Critic
DQN	Deep Q-Network
DDPG	Deep Deterministic Policy Gradient
TD3	Twin Delayed DDPG
SAC	Soft Actor-Critic
REINFORCE	Policy Gradient Algorithm
EMA	Exponential Moving Average

Chapter 1

Introduction

1.1 Background and motivation

Modern artificial intelligence (AI), particularly methods based on neural networks, has seen rapid growth in research activity and adoption across many scientific disciplines in recent years [1]. Some achieved ground-breaking results. Notable examples include systems such as ChatGPT [2], OpenAI Five [3] and AlphaGo [4]. A significant portion of these breakthroughs, especially in the domain of reinforcement learning (RL), have occurred in game-based environments [5].

One reason for this trend is that games offer a safe, controlled, and easily replicable environment in which to train and evaluate agents [6]. Since many games are already implemented digitally, they provide an accessible and practical playground for reinforcement learning research [6]. However, despite the successes observed in such environments, real-world applications of these AI systems still face substantial challenges [7].

A prominent example of such a challenge is the so-called *black box problem* [8]. Because many modern AI agents rely on neural networks, their internal decision-making processes are often opaque [9]. While these agents may perform well on average, it becomes difficult to understand or predict their behaviour in edge cases or exceptional scenarios. A practical illustration of this issue was observed during the OpenAI Five Dota 2 demonstration match, where the AI, despite winning the series 2–0, occasionally performed questionable or outright bizarre actions during gameplay [3].

AI systems that function as black boxes cannot be relied upon in high-stakes domains such as medical surgery or autonomous vehicles [7]. Even if an AI agent performs well on average, a single unintended action—such as causing harm to a patient due to a misjudgment—would be unacceptable. As a result, an entire area of research has emerged to address this concern, known as *interpretability* and *explainability*, often collectively referred to as *explainable artificial intelligence* (XAI) [10].

This dissertation aims to address the black-box problem in reinforcement learning; however, it adopts a non-conventional approach to this issue. Instead of focusing on the opaque nature of neural networks and attempting to explain the agent’s individual decision-making, this dissertation focuses on the fundamental principles upon which these algorithms are built, the features and side effects they produce due to the rules governing their learning, and how this relates to the mathematical and implementation processes of the agents.

This analysis is conducted using existing intelligible techniques that shed light on the internal workings of the agents—including the black-box components—and other side effects resulting from implementation details.

Through this approach, the study seeks to illuminate the internal mechanisms of AI agents, ultimately contributing to improved understanding, debugging, and development of such systems. In the subsequent sections of this work, precise definitions of both *interpretability* and *explainability* are provided. Until then, the term *intelligibility* serves as a general descriptor encompassing both concepts.

1.2 Scope and limitations

To gain a deeper understanding of the agents and the black-box nature of neural networks, the mathematics, algorithms, and implementation of these agents are closely related and critically important. Therefore, a custom library called `fund_rl` (link in Appendix A) will be created. This library will aim to bridge the gap between the mathematical formulation of the agents, the algorithms that define them, and their implementation. The library will focus on transparency and simplicity above performance, making it different from standard implementations and, where necessary, it will even sacrificing performance to preserve clarity. It provides tools to analyse agents in training environments by including functions and modules.

Supporting tools, such as mathematical libraries, neural network frameworks, plotting utilities, and environment wrappers, will be employed where appropriate. However, the core agent implementations will be developed from scratch. These implementations are not intended to be production-grade or achieve state-of-the-art performance; instead, they will prioritize simplicity and clarity. Simplified techniques and design shortcuts may be employed to facilitate ease of interpretation.

This approach is adopted to enable a more thorough investigation into the intelligibility of reinforcement learning agents by ensuring that their internal mechanisms remain as transparent and understandable as possible. In contrast with commonly used reinforcement learning libraries, `fund_rl` focuses on bridging the gap between the mathematical formulation, algorithm design, and implementation of agents, while providing tools for practical analysis and experimentation. Furthermore, this study will be limited to existing methods

for interpretability and explainability, with the aim of demonstrating how these methods can contribute to a better understanding of how reinforcement learning agents operate.

Additionally, only reinforcement learning tasks with discrete action spaces will be considered. This decision is based on the fact that some of the selected agents are either incompatible with or particularly difficult to configure for continuous environments. Therefore, only discrete environments will be included. However, as is common in mathematical and engineering disciplines, the techniques employed here should, in principle, be extendable to the continuous domain.

Lastly, this research will focus on a small subset of four pre-existing reinforcement learning algorithms. The goal is not to develop new algorithms, but rather to offer a clearer and more transparent perspective on existing agents and their decision-making policies.

1.3 Research objectives

Due to the complexity of the main research question, it is divided into several smaller sub-questions, each of which is addressed in the corresponding chapters.

Main Research Question:

How can first-principles implementations of RL agents enhance our understanding of their behaviour and improve intelligibility in controlled environments?

Sub-Questions:

1. How do reinforcement learning agents work?

To implement RL agents from first principles, it is essential to understand their underlying mechanisms. This topic is explored in **Chapters 2 and 3**. This question is addressed by referring to existing research on the subject and presenting it in the form of a literature review. This is followed by two conclusions, presented at the end of Chapter 2 and Chapter 3, which summarize and demonstrate how the question is addressed.

2. In what environments will the agents be trained?

It is important to examine the controlled environments in which these agents will operate and learn. This is discussed in **Chapter 4**. This question is addressed through a brief literature review in the context of the environments, followed by a conclusion presented at the end of Chapter 4.

3. What is intelligibility?

To investigate the intelligibility of RL agents, it is necessary to define what is meant

by the term. This definition and discussion are presented in **Chapter 5**. This question is addressed through a literature review, followed by a conclusion at the end of Chapter 5 outlining how the question is addressed.

4. **Which techniques are available to interpret the behaviour of an agent?**

Various techniques can be used to analyse and interpret agent behaviour. These are examined in **Chapter 5**. This question is addressed through a literature review, followed by a conclusion at the end of Chapter 5 outlining how the question is addressed.

5. **How can implementation and evaluation improve intelligibility?**

Investigating the implementation and evaluation of RL agents provides insight into how intelligibility can be enhanced. This is addressed in **Chapters 6**. This question is addressed by outlining the underlying mechanisms in detail, showcasing how the agents are implemented and how they operate in practice. This is then linked to the interpretability techniques discussed in the previous two chapters. A conclusion is provided at the end of Chapter 6, outlining how the literature on agent design and implementation can contribute to a better understanding of their internal workings.

6. **How did this process help improve the intelligibility of RL agents?**

This concluding question reflects on how the research process contributed to improving the intelligibility of RL agents. It is covered in **Chapters 7 and 8**. Chapter 7 contributes to the answer to this question by presenting a series of experiments utilizing the agents outlined earlier, in the environments with the techniques discussed previously. This is followed by an analysis of the results and a discussion at the end of Chapter 7. Using these experiments and the formalized results, a comprehensive conclusion is drawn in Chapter 8, completing the research and addressing the main question.

The validation and verification of each sub-question, as well as the overall research objective, are conducted either through a literature review supporting the claims or through the experiments presented in Chapter 7, which substantiate the literature outlined in each chapter. This process serves to bridge the literature with the practical implementation, thereby validating the claims made throughout the study.

1.4 Structure of the report

In **Chapter 2**, foundational concepts in RL are explored, along with general terminology relevant to the study. **Chapter 3** provides an overview of RL agents, including a concise explanation of how they function and the core principles behind them.

Chapter 4 discusses the controlled environments in which the agents will be trained, along with the setup of the experiments. In **Chapter 5**, the concept of interpretability

is examined in depth, including its definition within the context of this research and the methods used to improve it in RL agents.

Chapter 6 presents the implementation of the agent and begins establishing a link between the agent’s design and its observed behaviour. **Chapter 7** discusses the experimental results and findings. The experiments conducted in this chapter follow the experimental methodology outlined in Chapter 4. Several experiments are carried out in Chapter 7, each designed to address a specific experimental question. Each experiment provides answers to its corresponding question. Furthermore, Sub-question 6 is also addressed based on the outcomes of these experiments. Finally, **Chapter 8** concludes the report by summarising key insights and reflecting on how the research contributed to the interpretability of RL agents.

Chapter 2

Foundations of reinforcement learning

The following sections present the key concepts that form the foundation of reinforcement learning. This section focuses on providing the essential mathematical background necessary to understand how RL agents operate at a fundamental level. This foundation serves as a starting point for gaining deeper insight into the internal workings of agents.

Attempting to develop or improve artificial intelligence systems without a thorough understanding of their underlying mechanisms can lead to errors, suboptimal performance, and limitations in long-term progress [11]. Therefore, this section begins with an introduction to *Markov Decision Processes* (MDPs), which provide the formal framework for modeling decision-making problems in RL [12], and then discusses policies and value functions, which are essential for understanding the Bellman equation.

Four main solution methods are then introduced: *Dynamic Programming*, *Monte Carlo Methods*, and *Temporal-Difference Learning*. In addition, a separate class of approaches—*Policy Gradient Methods*—is presented. Subsequently, three important concepts within policy gradient methods are discussed: the *Policy Gradient Theorem*, *Advantage Functions*, and *Entropy Regularization*. Finally, the fundamental differences between reinforcement learning and other paradigms such as supervised and unsupervised learning are examined. These concepts are essential for understanding how reinforcement learning agents function, and a basic understanding of them is required for further study [12].

2.1 Markov decision processes

Many reinforcement learning models use the Markov Decision Process (MDP)¹ [12]. To fully understand these models, it is crucial to first understand the MDP framework. The MDP is a foundational concept in reinforcement learning, providing a formal structure for defining the agent, its environment, states, actions, and rewards [12]. In the MDP, the agent is the decision-maker that takes actions within a defined environment. The environment is the context in which the agent operates, and it interacts with the agent in a mutually dependent system. The agent's actions cause changes within the environment, which in turn provides the agent with rewards based on those actions. The agent's objective is to maximize its cumulative reward over time [12].

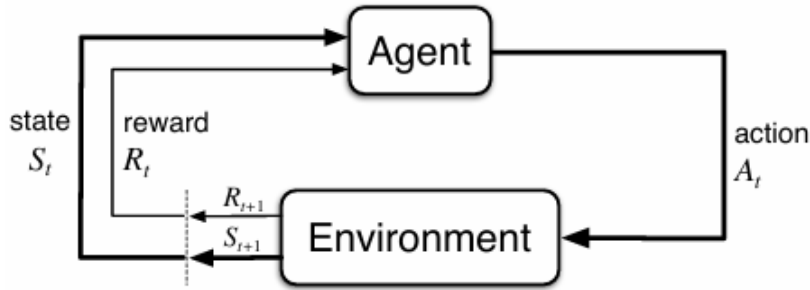


Figure 2.1: Markov Decision Processes

As illustrated in Figure 2.1, the MDP can be represented as a feedback loop where the agent's actions influence the environment, and the environment subsequently offers rewards [12].

The MDP operates in discrete time steps, where the agent's current state S_t serves as input. The agent then takes an action A_t , which causes the environment to evolve, transitioning the system to state S_{t+1} at time step $t + 1$ and yielding a reward R_{t+1} [12]. Many RL models are built on this structure, with the goal of maximizing the cumulative rewards obtained from the environment [12].

An example of formulating a control problem as a MDP is the *inverted pendulum* problem. In this scenario, the agent is typically a computer that controls a motor to apply force to a cart on which a pendulum is mounted.

The environment consists of the inverted pendulum system, including the cart and the surface on which it moves. A state in this system can include variables such as the cart's position and velocity, as well as the pendulum's angle and angular velocity.

The action space typically includes discrete actions such as pushing the cart to the left or to the right. The reward function can be defined such that the agent receives a reward

¹A Markov process is a stochastic process in which the future state is independent of the past, given the present state [13]. This means that if we have the full history of a given state, we can predict the current state. Moreover, if we have the current state, we can predict the future states regardless of the past.

of +1 for every time step that the pendulum remains upright within a specified angle threshold.

At each time step, the agent selects an action (e.g., "push cart left"), the environment responds by applying the corresponding force to the cart, and the dynamics of the system update accordingly. If the pendulum remains upright within the defined bounds, the agent receives a reward of +1. The agent then observes the next state configuration and selects a new action.

This process continues until the agent fails to keep the pendulum upright or reaches a predefined terminal condition. This formulation allows reinforcement learning algorithms to be applied effectively to learn a control policy for balancing the pendulum.

2.2 Policy

According to the Oxford Dictionary, a policy is defined as a principle or course of action that is adopted or proposed as desirable, advantageous, or expedient [14]. Within the domain of reinforcement learning, this definition is extended. A policy, denoted as π , is a mapping from a state to an action [12]. The optimal policy, represented as π_* , is the policy that maximizes the expected cumulative reward received by the agent [12].

A policy determines the behaviour of a reinforcement learning agent and may take various forms, such as simple functions, lookup tables, or complex search algorithms [12]. A stochastic policy is represented as $\pi(a | s)$, which indicates the probability of selecting action a in state s [12]. In contrast, a deterministic policy is expressed as $\pi(s) = a$, meaning that the agent consistently selects action a when in state s [12].

2.3 Value function

Nearly all reinforcement learning algorithms rely on value functions [12]. A value function, denoted as $v_\pi(s)$, represents the expected return starting from state s and following policy π for action selection thereafter [12]. This can be formally defined as [12]:

$$v_\pi(s) = \mathbb{E}_\pi(G_t | S_t = s) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s \right] \quad (2.1)$$

In Equation 2.1, G_t denotes the cumulative return obtained starting at time step t and continuing until the end of the episode² [12]. The parameter γ represents the discount factor, which decreases the weight assigned to future rewards in the return calculation

²An episode is a sequence of interactions between an agent and its environment, beginning from an initial state and terminating at a terminal state [15]

[12]. A value of γ close to 0 indicates that future rewards are considered less important, whereas a value close to 1 indicates that future rewards are considered nearly as important as immediate rewards. The function $v_\pi(s)$, referred to as the state-value function, assigns a value to each state under policy π [16].

There also exists an action-value function [16]. This function, denoted as $q_\pi(s, a)$, represents the expected return starting from state s , taking action a , and thereafter following policy π [16]. It can be formally defined as [16]:

$$q_\pi(s, a) = \mathbb{E}_\pi(G_t \mid S_t = s, A_t = a) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right] \quad (2.2)$$

The optimal state-value function $v_*(s)$ is defined as the maximum value over all policies for each state s , such that $v_*(s) \geq v_\pi(s)$ for all π [12, 16]. Similarly, the optimal action-value function $q_*(s, a)$ is the maximum expected return achievable by any policy, starting from state s , taking action a , and acting optimally thereafter [12, 16]. It is formally written as [12]:

$$q_*(s, a) = \mathbb{E} [R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \quad (2.3)$$

Equation 2.3 shows that the optimal action-value function is equal to the expected immediate reward after taking action a in state s_t , plus the discounted optimal state-value from the next state S_{t+1} [12, 16].

2.4 Bellman equations

The Bellman equations form the foundation upon which value function calculations are based [16]. The core idea behind the Bellman equation is that the value of a state is equal to the immediate reward received for being in that state, plus the expected discounted reward of the next state [16].

The state-value function and the optimal state-value function can both be expressed using their respective Bellman equations [12, 16].

$$v_\pi(s) = \mathbb{E}_\pi [R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \quad (2.4)$$

$$v_*(s) = \max_a \mathbb{E} [R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \quad (2.5)$$

Similarly, the action-value function and the optimal action-value function can also be expressed using their respective Bellman equations [12, 16].

$$q_\pi(s, a) = \mathbb{E}_\pi [R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a] \quad (2.6)$$

$$q_*(s, a) = \mathbb{E} \left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a \right] \quad (2.7)$$

By expressing the value functions recursively, as shown in Equations 2.4, 2.5, 2.6, and 2.7, their implementation in code becomes straightforward and efficient.

2.5 Dynamic programming

Dynamic programming encompasses a class of algorithms that compute the optimal policy for a given environment, assuming access to a perfect model of the environment and leveraging the structure of a Markov Decision Process [12, 17]. While theoretically valuable, such algorithms have limited practical applicability due to their reliance on an exact model of the environment—an assumption that is often unrealistic or infeasible in real-world scenarios [12, 17]. Moreover, dynamic programming methods typically incur significant computational costs [12, 17].

Consequently, many reinforcement learning approaches aim to approximate the results of dynamic programming without requiring a perfect model or incurring the same level of computational complexity [12].

2.6 Monte Carlo methods

Monte Carlo methods are a class of algorithms used to estimate the optimal value function through random sampling [12]. By collecting experience from interactions with the environment, these methods estimate expected returns and derive an optimal policy. A key advantage is that they do not require a model of the environment, relying solely on observed sequences of states, actions, and rewards [12]. However, Monte Carlo methods require complete episodes to update their estimates, making them unsuitable for infinite-horizon or non-terminating tasks [12].

A simple example of the update equation used in Monte Carlo simulations after acquiring an entire episode is [12]

$$V(s_t) \leftarrow V(s_t) + \alpha [G_t - V(s_t)] \quad (2.8)$$

In Equation 2.8, G_t represents the total cumulative reward obtained from time step t until the end of the episode. The parameter α denotes the step size, which controls the extent to which the value function is updated. By repeatedly interacting with the environment, obtaining trajectories (complete episodes consisting of state, action, and reward pairs),

and applying Equation 2.8, Monte Carlo methods can approximate the optimal value function and thereby discover the optimal policy [12].

2.7 Temporal difference learning

Temporal Difference (TD) learning marks a key advancement in reinforcement learning [12]. It combines the strengths of dynamic programming—updating after each transition—with the sampling-based approach of Monte Carlo methods, thereby eliminating the need for a model of the environment [12].

By modifying Equation 2.8, the dependency on G_t is removed, enabling updates to be performed at every time step t rather than only at the end of an episode. Simply discarding G_t , however, would strip the update rule of its meaning. Instead, G_t is replaced with the immediate reward received at time t , together with an additional term that estimates the expected future reward [12]. This modification yields the Temporal Difference update equation [12]:

$$V(s_t) \leftarrow V(s_t) + \alpha [R_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \quad (2.9)$$

In Equation 2.9, R_{t+1} denotes the reward obtained at time $t + 1$, and γ represents the discount factor applied to future rewards. By repeatedly sampling experience³ from the environment and updating the value function according to Equation 2.9, Temporal Difference learning approximates the optimal value function or action-value function [12].

2.8 Policy gradient methods

While certain reinforcement learning agents employ TD learning to estimate value functions, others utilize Policy Gradient methods to directly optimize policies [18]. These methods do not involve learning a value or action-value function; rather, the policy is parameterized by a set of parameters θ , which are adjusted to modify the agent’s behavior [12, 18].

The probability of selecting action a at time t in state s , under a policy parameterized by θ , is defined as [12]:

$$\pi_\theta(a \mid s) = \Pr(A_t = a \mid S_t = s, \theta_t = \theta) \quad (2.10)$$

³Section 2.12 presents a discussion on exploration and exploitation, including methods by which the agent samples experiences from the environment.

Let $J(\theta)$ denote a function that quantifies the quality of the policy [12, 19]. By applying gradient ascent, as expressed in Equation 2.11, the policy is iteratively improved [19]:

$$\theta_{t+1} = \theta_t + \alpha \nabla_{\theta} J(\theta_t) \quad (2.11)$$

In this equation, θ_t represents the current set of policy parameters at time step t , and θ_{t+1} represents the updated policy parameters after applying the gradient step. The learning rate α determines the step size for the update, while $\nabla_{\theta} J(\theta_t)$ denotes the gradient of the objective function with respect to the policy parameters.

The objective function $J(\theta)$ represents the expected return obtained under the policy π_{θ} , which is parameterized by θ [19]. By maximizing the expected return, the performance of the agent improves [19]. Formally, $J(\theta)$ can be defined as [18]:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_{\theta}} [R(\tau)] \quad (2.12)$$

Where $\tau = (s_0, a_0, r_1, s_1, a_1, r_2, \dots)$ denotes a trajectory (sequence of states s_t , actions a_t , and rewards r_{t+1}) generated by following policy π_{θ} . The notation $\tau \sim \pi_{\theta}$ indicates that the trajectory is sampled according to the policy π_{θ} , and $R(\tau) = \sum_{t=0}^{T-1} \gamma^t r_{t+1}$ denotes the cumulative discounted reward along the trajectory. The discount factor $\gamma \in [0, 1]$ reduces the weight of future rewards.

While $J(\theta)$ provides a theoretical objective for optimizing a policy, it cannot be directly computed or solved in practice due to its dependence on the full distribution of trajectories [18]. The final piece of the puzzle is the Policy Gradient Theorem, which establishes a mathematical connection between this abstract objective and a form that can be approximated from sampled data [12]. The theorem provides a way to compute an estimate of the gradient of $J(\theta)$, making it possible to implement policy optimization methods in practice [12].

2.9 Policy gradient theorem

The Policy Gradient Theorem provides a mathematical justification for rewriting the objective function defined in Equation 2.12 in a form that can be practically computed [12, 18]. This reformulation enables the calculation of the gradient of $J(\theta)$, which can then be used to update the policy parameters, as shown in Equation 2.11.

The Policy Gradient is given by [18, 19]:

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\tau \sim \pi_{\theta}} [R(\tau) \nabla_{\theta} \log \pi_{\theta}(\tau)] \quad (2.13)$$

Where $\tau = (s_0, a_0, r_1, s_1, a_1, r_2, \dots, s_{T-1}, a_{T-1}, r_T)$ denotes a trajectory generated by following the policy π_θ , and $\tau \sim \pi_\theta$ indicates that the trajectory is sampled according to the policy π_θ . The cumulative discounted reward along the trajectory is given by $R(\tau) = \sum_{t=0}^{T-1} \gamma^t r_{t+1}$, and $\pi_\theta(\tau) = \prod_{t=0}^{T-1} \pi_\theta(a_t | s_t)$ is the probability of the trajectory under the policy. Finally, $\nabla_\theta \log \pi_\theta(\tau)$ represents the gradient of the log-probability of the trajectory with respect to the policy parameters, which indicates how to adjust the parameters to increase the likelihood of high-reward trajectories.

From the Policy Gradient Theorem, the derivative of $J(\theta)$ formally depends on all possible trajectories [12]. However, since it is expressed as an expected value, it can be approximated by sampling individual trajectories [12]. By averaging over many sampled trajectories, this estimator converges to the true gradient given in Equation 2.13 [12, 18].

2.10 Advantage functions

A common component of policy gradient methods that employ the Policy Gradient Theorem to update the policy parameters θ is the use of an *advantage function* instead of raw returns [18]. The advantage function, denoted $A^\pi(s, a)$, assigns a numerical value to an action a in state s , indicating whether it performs better or worse than the average expected return of that state [16]. This technique reduces the variance of the policy gradient estimate and improves the stability of learning [18].

Several methods exist for computing the advantage function. A simple example is to subtract the mean return from the return of a specific action, serving as a baseline that preserves the unbiased nature of the gradient estimate [18]. Positive advantage values indicate that the action performs better than average, while negative values indicate worse-than-average performance [18].

2.11 Entropy regularization

Another important component of policy gradient methods is the concept of *entropy* of a probability distribution [18]. Entropy measures the randomness or uncertainty of a distribution [20]. A high entropy value indicates that the distribution is stochastic [20], meaning the policy selects actions with high variability. Conversely, a low entropy value indicates that the distribution is nearly deterministic [20], with the policy favoring certain actions strongly.

Entropy can be used to control the exploration behavior of a policy gradient agent [18]. By adding the negative of the entropy as a regularization term to the loss function, the agent is encouraged to maintain high entropy, thereby continuing to explore different actions rather than prematurely converging to a deterministic policy [18].

2.12 Exploration vs exploitation

One of the key distinctions between reinforcement learning and both supervised and unsupervised learning is the trade-off between exploration and exploitation [12].

In reinforcement learning, the agent must make 'sacrifices'. It cannot always take the action it believes to be the best (exploiting its current knowledge), because that action may not actually be optimal [12]. However, it also cannot always take exploratory actions (choosing suboptimal actions to broaden its knowledge) because this would prevent it from achieving optimal performance [12]. Therefore, there must be a balanced trade-off between exploitation and exploration [12, 19].

In the book *Reinforcement Learning: Second Edition* by Richard S. Sutton and Andrew G. Barto, several methods are outlined to balance the trade-off between exploration and exploitation [12]:

- **Greedy** - Always selects the action with the highest known reward, leading to fast exploitation but no exploration.
- **Epsilon-Greedy** - Chooses the best-known action most of the time but randomly explores with a small probability ϵ .
- **Upper Confidence Bound (UCB)** - Selects actions based on both expected reward and uncertainty, prioritizing less-explored actions.
- **Decayed Epsilon-Greedy** - Starts with a high exploration rate (ϵ) and gradually reduces it over time, transitioning from exploration to exploitation.
- **Optimistic** - Initializes action values optimistically high, encouraging initial exploration before converging to better choices.

For ease of comparison across methods, the experimental example presented in the book is extended to include all five approaches. The experiment is conducted over 1000 independent runs, with each run consisting of 5000 steps. The multi-armed bandit problem is used as the test environment (explanation of this environment is provided in Chapter 4.). In this setting, the agent must identify the optimal slot machine (arm) that yields the highest rewards. Each slot machine provides a reward that is normally distributed, and the means of the arms are also drawn from a normal distribution.

All agents are implemented using Q-learning, with the following configurations:

- **Optimistic initialization** - Uses an initial value of 1 to encourage exploration.
- **Fixed ϵ -greedy** - Selects actions using an ϵ value of 0.1.

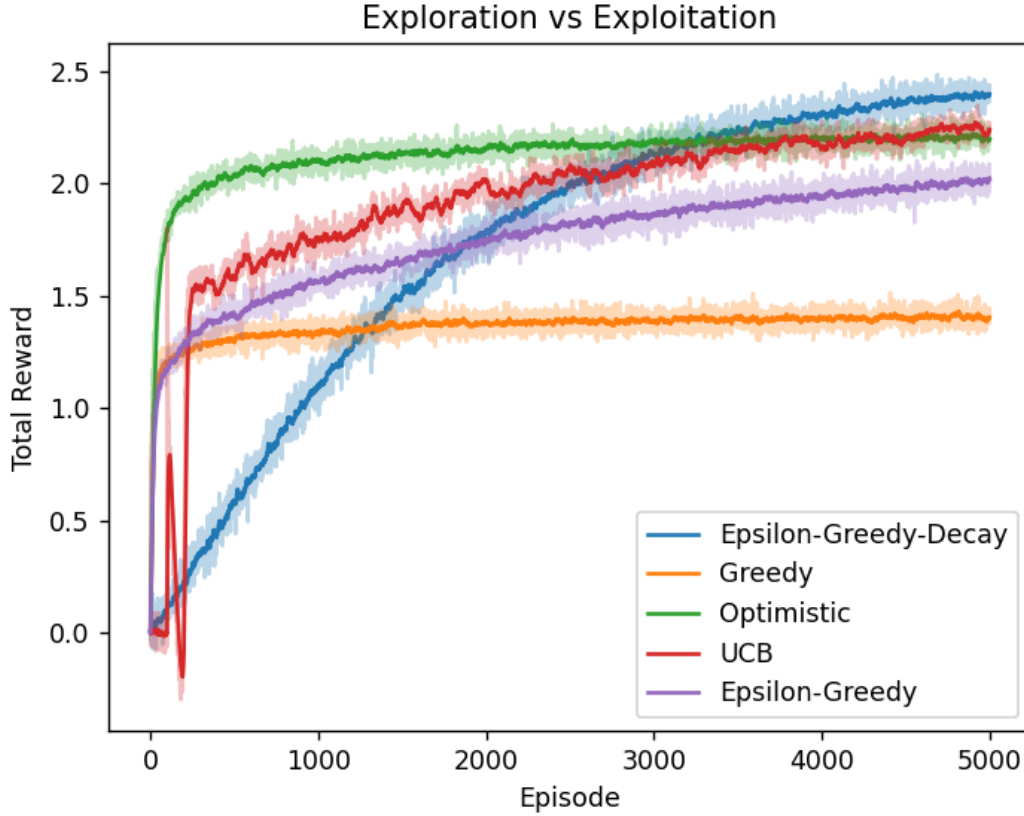


Figure 2.2: Comparison of Exploration Strategies

- **Decaying ϵ -greedy** - Starts with full exploration (100%) and decays gradually to 10% by episode 3500 (70% of total steps).
- **Upper Confidence Bound (UCB)** - Selects actions using a C value of 2.

The average results of these configurations are presented in Figure 2.2.

The results illustrate how each method demonstrates distinct learning behaviors, rather than identifying any one approach as universally superior. For instance, while the Greedy strategy shows strong initial performance by exploiting current knowledge, its lack of exploration limits long-term improvement. In contrast, methods like Optimistic and UCB initially explore more, resulting in delayed but significant gains. The epsilon-greedy strategies strike a balance between exploration and exploitation, improving steadily over time, with epsilon-decay shows continues improvement.

Importantly, these differences highlight the characteristic *shapes* of the learning curves rather than ranking methods absolutely. Each strategy has its own strengths depending on the nature of the environment. For example, UCB and epsilon-decay may be more suitable in stationary, episodic problems where long-term learning is essential [12]. On the other hand, Optimistic can be valuable in situations requiring early exploration followed by stable exploitation, while fixed epsilon-greedy maintains a consistent balance throughout

[12].

Ultimately, the choice of method should be guided by the specific needs of the task at hand. Among all, the epsilon-greedy approach remains one of the most commonly used in practice due to its ease of implementation and its effective balance between exploration and exploitation across a wide range of reinforcement learning problems.

2.13 Conclusion

This chapter focused on the fundamental concepts underpinning RL. It began with the Markov Decision Process, followed by the definition of the policies and value functions and the formulation of the value functions in terms of the Bellman equation. The chapter then examined dynamic programming and Monte Carlo simulation methods, discussing their applications as well as their limitations. Temporal difference learning was subsequently introduced as a combination of these two approaches.

The chapter also explored methods that did not rely on value functions, specifically policy gradient techniques. It described the policy gradient theorem and highlighted two common enhancements—advantage functions and entropy regularization—that improved the performance of policy gradient methods. Finally, the chapter considered a fundamental distinction between reinforcement learning and other areas of artificial intelligence, namely the exploration–exploitation tradeoff.

Understanding these basic concepts is crucial for a solid foundation in RL [11, 12]. Comprehending the underlying mathematics of these algorithms provides insight into how agents learn and behave, which is further illustrated in Chapter 3 [11, 12]. Without a fundamental understanding of RL, subsequent improvements, performance enhancements, and advanced techniques are prone to errors and may hinder the development of effective reinforcement learning systems [11].

Chapter 3

Reinforcement learning algorithm taxonomy

In this chapter, a taxonomy and overview of RL agents is presented, followed by a description of the selection criteria and the agents selected for further analysis. Each selected agent is accompanied by a motivation for its inclusion. A more in-depth discussion of these agents follows in Chapter 6.

There is a wide range of categories within RL, and classifying them can be challenging. RL algorithms can be categorized based on factors such as continuous or discrete states, continuous or discrete actions, or any combination of the two, as illustrated in [21].

Another approach to categorization is based on the type of implementation, such as Model-Based and Model-Free methods, as shown in [22], and [23]. Another categorization is between on-policy methods versus off-policy methods, or value optimization versus policy optimization methods [24].

While categorizing RL models based on input and output can be useful for selecting an appropriate algorithm for a specific case [24], many algorithms can be adapted to various input and output spaces through sampling, quantization, or by converting discrete signals into continuous signals using additional steps [25].

Thus, for this research, categorization is based on on-policy versus off-policy methods, with a subcategorization into value optimization versus policy optimization. It is important to note that it is neither practical nor feasible to discuss every reinforcement learning algorithm or to create an exhaustive taxonomy encompassing all of them [23].

Given the sheer number of algorithms, many of which may fall into multiple categories, it is not possible to comprehensively include them all [23]. Therefore, only a selection from each category will be mentioned and discussed, though it is important to acknowledge that far more algorithms exist beyond the ones highlighted here.

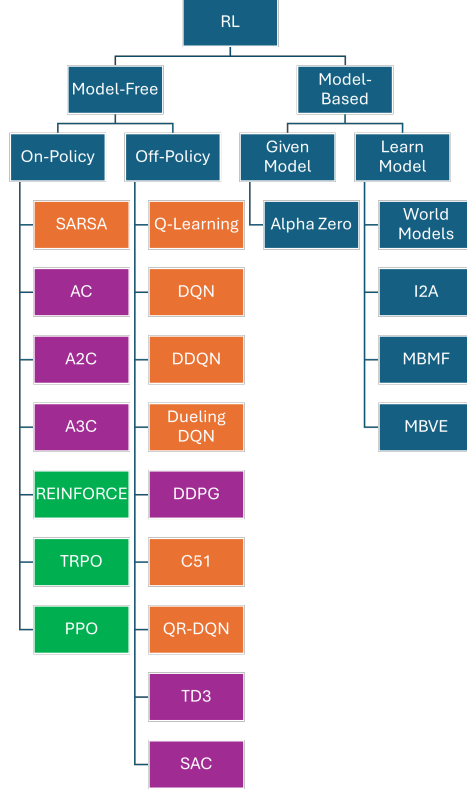


Figure 3.1: Overview of RL Categories

As shown in Figure 3.1, reinforcement learning is divided into two major categories: Model-Free and Model-Based. Each of these is further subdivided into two additional categories, resulting in four distinct groups into which different algorithms fall. Since model-free algorithms are the most common [23], this research focuses solely on model-free algorithms.

- An **orange algorithm** refers to a value-based method that optimizes a value function. These algorithms strongly overlap with off-policy methods.
- A **green algorithm** refers to a policy optimization method, which shows a strong correlation with on-policy approaches.
- A **purple algorithm** refers to a hybrid method that combines both policy optimization and value-based optimization.

Another important aspect of these algorithms is their relative popularity compared to each other. This relative popularity is outlined in Figure 3.2. Figure 3.2 is compiled from a simple query on the Scopus website [26], where the number of articles found is recorded. The full names of the algorithms, such as Soft Actor-Critic, are paired with their abbreviations, e.g., SAC. A table is then constructed by calculating the relative popularity of each algorithm, from which Figure 3.2 is derived.

From Figure 3.2, it is evident that Q-Learning is by far the most popular algorithm in recent years, followed by Deep Q-Learning and Actor-Critic methods. Other algorithms

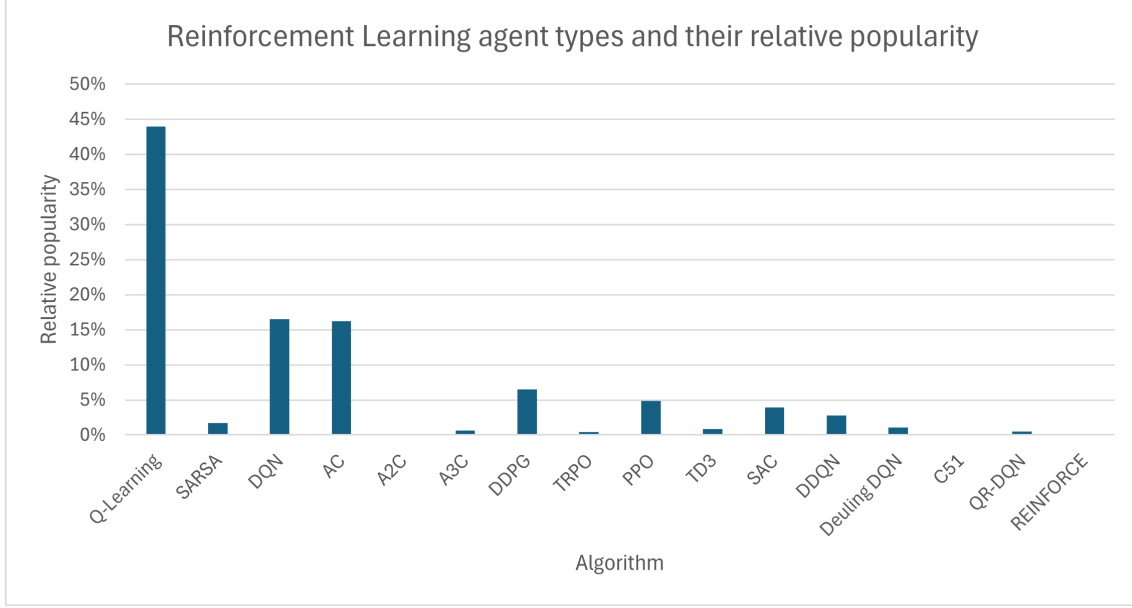


Figure 3.2: Graph of Reinforcement Learning Algorithm's Relative Popularity

receive less research attention, with some being hardly studied at all.

The query template used on Scopus is shown below.

```
TITLE-ABS-KEY ( WrittenOutName AND ShotHandName )
AND PUBYEAR > 2014
AND ( LIMIT-TO ( SUBJAREA , "ENGI" )
      OR LIMIT-TO ( SUBJAREA , "COMP" )
      OR LIMIT-TO ( SUBJAREA , "MATH" ) )
```

3.1 Model-Free vs Model-Based

The two primary categories of Reinforcement Learning, as shown in Figure 3.1, are Model-Free and Model-Based methods. In this context, models refer to representations of the environment and functions that predict its future states, capturing its internal workings [27]. Such models often employ physics equations, motion equations, differential equations, or other mathematical techniques to represent environments or systems [27].

Model-Based algorithms utilize a model of the environment to achieve a better understanding and more accurate predictions of future states [23]. These algorithms are generally observed to train faster and perform better than model-free algorithms [23]. This performance increase and faster training is because they can predict future states, leading to a more stable and improved approach [23].

However, models are not always available, and creating accurate models can be complex or even impossible, particularly in environments with many random factors [23]. Model-

free algorithms, which do not rely on a model of the environment, are more widely used due to their ease of implementation and applicability to a variety of problems [23]. In contrast, model-based approaches are more limited by the difficulty of creating accurate models [23].

3.2 On-Policy vs Off-Policy

Delving deeper into Model-Free Reinforcement Learning algorithms, the next logical distinction, as shown in Figure 3.1, is between On-Policy and Off-Policy methods.

An On-Policy Method refers to a method where an agent uses a specific policy to make decisions. After making a decision based on the policy, the agent evaluates the outcome of that decision and subsequently improves the policy. In this approach, the agent utilizes its own active policy to make decisions and learn from the results. Therefore, it is called an On-Policy method [12, 28].

- **SARSA** - SARSA is an on-policy temporal difference learning algorithm that uses a Q-table as its policy [29]. The Q-table maps states to actions and serves as the action-value function [29]. The agent selects actions based on the maximum value from the action-value function, while occasionally making random choices to encourage exploration [29]. SARSA is effective in small environments with fixed state and action spaces, as it requires a complete mapping of all state-action pairs [29]. It updates the value function using the Bellman equation (Equation 2.6). Over the past decade, only 2% of reports with "SARSA" in the title have been recorded in Scopus, indicating that while it is a well-known technique, other simple yet more effective methods, such as Q-Learning, are more commonly used.
- **AC** - The Actor-Critic (AC) algorithm is an on-policy reinforcement learning method that simultaneously optimizes the policy and the value function [30]. It consists of two components: the actor, which updates the policy, and the critic, which evaluates the quality of the policy by estimating the value function [30]. AC can handle both continuous and discrete state and action spaces, making it suitable for a wide range of environments [30]. As shown in Figure 3.2, this algorithm is relatively popular, with approximately 16% of relevant research papers focusing on the Actor-Critic method.
- **A2C** - The Advantage Actor-Critic (A2C) algorithm is an enhancement of the standard AC method [31]. Instead of the critic directly estimating the value function, A2C estimates the advantage function, which quantifies how much better or worse a particular action is compared to the average action [31]. This modification reduces variance in policy updates, resulting in more stable training [31]. A2C can be applied to both continuous and discrete environments [30]. However, as shown in Figure 3.2, none of the research papers in the referenced dataset focus on A2C.

- **A3C** - The Asynchronous Advantage Actor-Critic (A3C) algorithm is an improvement over A2C, utilizing multiple agents in parallel to update the same policy simultaneously [31]. This parallelization enables faster and more stable training [31]. A3C can be applied to both discrete and continuous state and action spaces [30]. According to the dataset shown in Figure 3.2, approximately 1% of research papers focus on the A3C algorithm.
- **REINFORCE** - REINFORCE represents the most basic form of on-policy policy optimization [32]. It operates on the principle that actions yielding higher rewards should be selected more frequently, while less effective actions should be chosen less often [12, 32]. REINFORCE can handle both continuous and discrete state spaces, as well as continuous and discrete action spaces [33]. As shown in Figure 3.2, REINFORCE is not a widely used algorithm, appearing in 0% of research papers over the past ten years. This limited prevalence is largely due to the emergence of simpler yet higher-performing reinforcement learning algorithms, such as PPO, which have dominated the field.
- **TRPO** - Trust Region Policy Optimization (TRPO) is an on-policy policy optimization algorithm that builds on policy gradient methods [34]. It updates the policy by taking the largest possible step while ensuring the change remains within a specified constraint, preventing overly large updates [34]. TRPO can handle both continuous and discrete state and action spaces and has been applied in a variety of environments [34]. However, due to its mathematical complexity and implementation challenges, it is not widely used, as reflected by the 0% of research papers over the past 10 years focusing on this algorithm (Figure 3.2).
- **PPO** - Proximal Policy Optimization (PPO) is an on-policy policy optimization algorithm similar to TRPO, but it incorporates a clipping function to prevent overly large updates [33, 35]. This mechanism balances policy updates, maximizing cumulative future rewards while maintaining training stability [33]. PPO is the default algorithm used by OpenAI due to its simplicity and strong performance [36]. It supports both continuous and discrete state and action spaces, making it suitable for various environments, particularly complex ones, where it performs comparably to state-of-the-art algorithms [37]. Over the past 10 years, approximately 5% of relevant reports, as shown in Figure 3.2, have focused on PPO, a notable proportion considering its relatively recent introduction.

In off-policy methods, the agent selects actions according to a policy known as the *behavior policy* [12]. The outcomes and evaluations of these actions are recorded and stored [12, 28]. The agent also maintains another policy, called the *target policy*, which is improved based on the data generated by the behavior policy [12, 28].

As illustrated in Figure 2.1, the behavior policy determines the actions sent to the environment, whereas the target policy receives the rewards and updates itself accordingly [12]. These two policies can share the same underlying value function [38]. Consequently,

updates to the target policy’s value function can also benefit the behavior policy if it relies on the same value function.

- **Q-Learning** - Q-Learning is an off-policy temporal difference learning algorithm, similar to SARSA [12]. It uses a Q-table as its policy, which contains a state-action mapping that serves as the action-value function [38]. The agent selects actions based on the maximum value provided by the action-value function, while occasionally making random choices to encourage exploration [38]. The action-value function is updated using the Bellman equation (Equation 2.6). Q-Learning is effective in small environments with fixed state and action spaces, as it requires a complete mapping of all state-action pairs [12, 38]. Over the past decade, approximately 44% of reports in Scopus with "Q-Learning" in the title highlight its significance as a research topic.
- **DQN** - The Deep Q-Network (DQN) algorithm builds upon the Q-Learning framework by employing a neural network to approximate the Q-values [39, 40]. DQN operates in both continuous and discrete state spaces but is typically applied to discrete action spaces [39]. According to the dataset compiled for Figure 3.2, approximately 16% of research papers reference DQN, reflecting its widespread popularity in the field.
- **DDQN** - The Double Deep Q-Network (DDQN) algorithm is an enhancement of the DQN method [41]. It employs two neural networks: one to estimate the Q-values for action selection and another to update these Q-values [41]. This dual-network architecture mitigates the overestimation bias commonly observed in the original DQN algorithm [41]. DDQN performs effectively in both continuous and discrete state spaces, as well as discrete action spaces [39]. According to the dataset referenced in Figure 3.2, approximately 3% of research papers mention DDQN, suggesting that while it is less prevalent than some other algorithms, it remains a well-regarded approach within the field.
- **Dueling DQN** - The Dueling Deep Q-Network (Dueling DQN) algorithm extends the DQN architecture by dividing the final layer of the neural network into two separate outputs [42]. One output estimates the value function, while the other estimates the advantage function [42]. The algorithm operates in both continuous and discrete state spaces with discrete action spaces [39]. According to the dataset compiled for Figure 3.2, approximately 1% of research papers mention Dueling DQN, indicating that while it is less prevalent, it maintains a notable presence in the literature.
- **DDPG** - The Deep Deterministic Policy Gradient (DDPG) algorithm is an off-policy reinforcement learning method that optimizes both the policy and value functions using an actor-critic framework, similar to the AC algorithm [43]. DDPG is designed to operate in both continuous and discrete state spaces with continuous action spaces, making it particularly suitable for continuous control tasks [43]. According to Figure 3.2, DDPG is referenced in approximately 7% of papers in the

dataset, highlighting its widespread adoption in reinforcement learning research.

- **C51** - The 51-Atom (C51) algorithm is a relatively recent reinforcement learning method that models the full return distribution rather than only the expected return [44]. This distributional approach allows C51 to outperform many state-of-the-art algorithms while using relatively few additional concepts, offering an alternative approximation method for reinforcement learning [44]. C51 can be applied to both continuous and discrete state spaces [44]. However, since its introduction around the same time as PPO, which achieved considerable popularity, C51 has not been widely adopted. As shown in Figure 3.2, 0% of the analyzed papers focus on C51, reflecting its limited prevalence in the literature.
- **QR-DQN** - The Quantile Regression Deep Q-Network (QR-DQN) algorithm is an extension of both C51 and DQN [45, 46]. It combines the distributional approach of C51 with the Q-value estimation framework of DQN [46]. QR-DQN is applicable to both continuous and discrete state spaces with discrete action spaces [45]. According to Figure 3.2, QR-DQN is slightly more prevalent than C51, though it still appears in fewer than 1% of research papers in the dataset.
- **TD3** - The Twin-Delayed Deep Deterministic Policy Gradient (TD3) algorithm is an off-policy actor-critic method that builds upon DDPG [47]. DDPG is prone to overestimating Q-values, and TD3 mitigates this issue by employing two Q-value functions to reduce overestimation [47]. TD3 operates effectively in both continuous and discrete state spaces with continuous action spaces [47]. According to Figure 3.2, TD3 is referenced in only 1% of papers, indicating relatively infrequent use compared to other algorithms.
- **SAC** - The Soft Actor-Critic (SAC) algorithm is an off-policy actor-critic method [48]. SAC optimizes a policy that jointly maximizes expected return and entropy, promoting both performance and exploration [48]. It is suitable for continuous or discrete state and action spaces, and can even operate in hybrid environments [48]. According to Figure 3.2, SAC is relatively popular, appearing in approximately 4% of papers in the dataset.

3.3 Value-Based vs Policy-Based

Another major distinction between RL agents lies in whether they follow a value-based, policy-based, or hybrid (actor-critic) approach [49, 50].

Value-based methods learn a value function that assigns scores to actions or states [12]. By repeatedly updating this evaluation function, the agent improves its estimation of which states or actions are good or bad [12]. If the approximation becomes nearly optimal, then selecting the action in each state that leads to the best next state results in an optimal policy [49].

In contrast, policy-based methods do not associate values with states [49]. Instead, they represent a probability distribution over actions [49]. As sequences of actions are executed and feedback is received, the probabilities of better actions are increased while the probabilities of worse actions are decreased [12]. Over time, this leads to an approximately optimal action distribution [12].

Hybrid methods use both approaches: the actor models a probability distribution over actions (the policy), and the critic estimates the value of being in a given state [50]. By working together, they improve both the action selection and the value estimates, gradually converging toward an optimal policy [50].

3.4 Overview of selected algorithms

This section a subset of four algorithms is selected for implementation during the experimental phase.

While model-based RL provides valuable insights, its implementation is often complex. Therefore, the focus remains on model-free approaches, which are more commonly used in practice [23]. Within this category, the scope narrows further to include both On-Policy and Off-Policy methods. To ensure a balanced evaluation, algorithms from both categories are selected. Specifically, two On-Policy and two Off-Policy algorithms are chosen.

The selection criteria further includes popularity—ensuring the inclusion of widely adopted methods—along with simplicity and foundational importance. Simpler algorithms often serve as the basis for more advanced techniques; therefore, understanding them is essential before progressing to more complex methods.

In addition, the selection considers diversity in application domains. Reinforcement learning is applied in areas such as games, control systems, and prediction tasks. Including algorithms from these domains enables a more comprehensive evaluation. Lastly, several of the selected algorithms incorporate neural networks, reflecting the increasing importance of deep reinforcement learning. The combination of neural networks and RL represents a significant research frontier, and evaluating such agents forms a critical part of this study.

The four selected algorithms are listed below, each followed by a rationale for its selection.

3.4.1 REINFORCE

The REINFORCE algorithm, introduced in 1992 by Ronald J. Williams [32], is an on-policy policy optimization method [12, 51]. Although it is not among the most widely used algorithms, as illustrated in Figure 3.2, REINFORCE provides a foundational approach upon which many policy optimization reinforcement learning methods are built [12]. Consequently, it has been selected for inclusion in this study. Furthermore, REINFORCE can

utilize neural networks, taking advantage of recent advances in deep learning [51].

3.4.2 Q-Learning

Q-Learning, first introduced by Watkins in 1989 and formally defined as an algorithm by Watkins and Dayan in 1992 [52, 53], is a value-based reinforcement learning method that belongs to the off-policy category [12]. Although the original Q-Learning algorithm does not utilize neural networks, its popularity—particularly through extensions such as DQN—demonstrates its foundational influence on many modern reinforcement learning methods [39]. Consequently, Q-Learning is included in this study due to its significance in the development of contemporary algorithms.

3.4.3 Double Deep Q-Network (DDQN)

The DQN algorithm was first introduced by DeepMind in 2015 [54, 55]. It was the first reinforcement learning algorithm capable of processing raw pixel sensory input and achieving human-level performance in Atari 2600 games [54, 55]. In 2016, DDQN was introduced as an improvement on DQN [56]. Although DDQN is not among the most widely used algorithms, as shown in Figure 3.2, its inclusion is justified due to its improved stability compared to DQN [56]. As a value-based method, DDQN belongs to the off-policy category [41].

3.4.4 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) was introduced by OpenAI in 2017 [37, 36]. It gained considerable attention due to its strong empirical performance and simplicity of implementation, effectively superseding its predecessor, Trust Region Policy Optimization (TRPO) [35]. PPO is a policy optimization algorithm that belongs to the on-policy category [35]. As shown in Figure 3.2, it is among the most widely used reinforcement learning algorithms. OpenAI has identified PPO as their preferred algorithm because of its robustness and general applicability [36]. PPO employs neural networks to represent both the policy and value functions, making it well-suited for modern deep reinforcement learning applications [33].

3.4.5 Selection justification

As demonstrated by the selected algorithms, the criteria established for this study are satisfied. The selection encompasses a combination of policy-based and value-based methods, on-policy and off-policy approaches, as well as algorithms with and without neural network integration. This ensures a diverse set of reinforcement learning strategies suitable

for experimental comparison.

3.5 Conclusion

This chapter began with an overview of the taxonomy of RL and presented several representative RL algorithms. It then discussed the distinction between model-based and model-free methods, followed by a comparison of on-policy and off-policy approaches, as well as policy optimization versus value optimization strategies.

The separation into categories and subcategories, along with the fundamental distinctions between them, is important because agents falling within a particular group are expected to share certain characteristics. This characterization helps identify features that may contribute to similarities in agent behavior, which is further investigated in Chapter 6.

Chapter 2 provided an overview of the mathematical concepts on which reinforcement learning agents were built. That chapter introduced the conceptual foundations that underlay these agents, enabling a clearer understanding of their internal functioning. By combining the mathematical principles discussed in Chapter 2 with the concepts presented here, the selected agents are implemented in Chapter 6.

Finally, four algorithms (Q-learning, DDQN, REINFORCE, and PPO) were selected for detailed study based on a set of criteria. Focusing on a limited set of algorithms allows for a more in-depth analysis in Chapter 6 and enables more diverse experimental comparisons in Chapter 7. The selected algorithms were chosen to ensure coverage across a broad range of reinforcement learning approaches.

Chapter 4

Experimental setup

This chapter describes the environments used in the experiments presented in Section 7. It explains the rationale for selecting each environment and highlights the variability and observability they introduce. The choice of environments is a critical aspect of the study; given their diversity, they are discussed separately and applied consistently throughout the experiments. Evaluation metrics are detailed in Section 5, outlining the methods employed to assess agent performance. Finally, the section provides an overview of the experimental procedures, explaining how each experiment is structured and conducted, with specific differences addressed in their respective sections.

4.1 Environments used

Below are the five environments selected for the experiments. Each environment presents a unique challenge and perspective that the agent must adapt to in order to effectively solve the given problem. For each environment, the input (i.e., the information the agent receives from the environment), the output (i.e., the action the agent must produce in response), the reward structure, and the transition dynamics of the environment are described.

4.1.1 Multi-Armed Bandit

The Multi-Armed Bandit (MAB) problem is a well-known and structured problem in the field of reinforcement learning [12]. It encapsulates the challenge of balancing exploration and exploitation when faced with multiple possible actions, each yielding uncertain rewards [12]. Effectively solving this problem demonstrates an agent’s ability to explore unknown options while exploiting known rewarding actions to maximize cumulative return [12]. This environment will not be used in the experiments in Chapter 7; however, it has been included here because the MAB environment is used in Chapter 3 to explain exploration versus exploitation.

The environment is defined as follows: the agent is presented with n different arms (i.e., slot machine levers), each associated with a fixed but unknown mean reward as shown in Figure 4.1 [12]. At each time step, the agent selects one arm to pull and receives a reward sampled from the distribution associated with that arm [12].

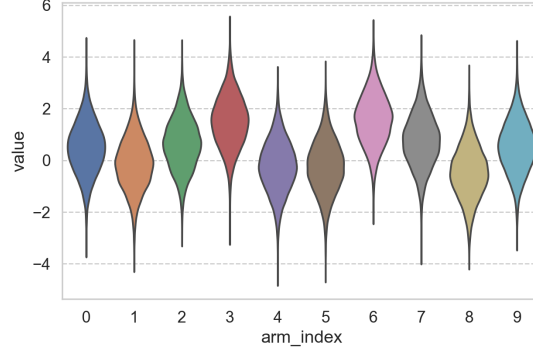


Figure 4.1: Multi-Armed Bandit example reward distribution for 10 arms

Goal: The agent aims to identify the arm with the highest expected reward using as few trials as possible, while also maximizing the total reward accumulated during the exploration process [12].

Input: None. No input is provided to the agent, as there are no observations available about the environment [12].

Output: An integer between 1 and n indicating the selected arm [12].

Reward: A reward sampled from the distribution associated with the selected arm [12].

Transition: The environment consists of a single state, and each episode terminates after one action, i.e., a single transition [12].

4.1.2 Taxi

The Taxi environment, developed by OpenAI Gym [57], consists of a 5×5 grid with walls separating some of the grid tiles as shown in Figure 4.2. Specific locations are marked near each corner of the grid [57]. A passenger is placed at a random location, and a destination is randomly assigned to another location. The agent must navigate the taxi to pick up the passenger and transport them to the designated destination without crashing [57].

This environment is simple and serves as a benchmark to verify whether all agent implementations have been successfully completed. Given its finite state space and action space, all agents should be able to solve the environment without significant difficulty [57].

Goal: The agent’s objective is to navigate to the passenger, pick them up, and transport them to the designated destination using the fewest number of steps, while avoiding crashes and ensuring correct drop-off [57].

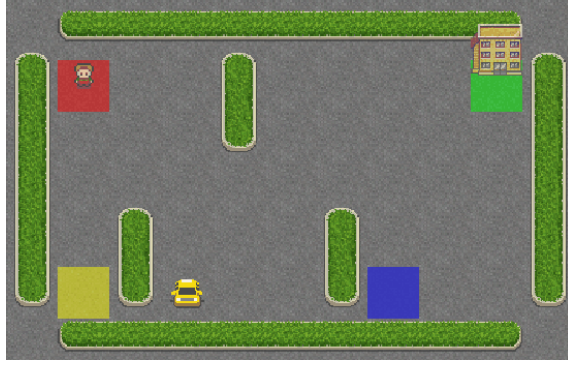


Figure 4.2: Taxi Environment

Input: A number between 0 and 404, corresponding to one of the 404 possible states in which the agent can find itself [57].

Output: At each step, the agent can perform one of six possible actions [57]:

1. Move south (down)
2. Move north (up)
3. Move east (right)
4. Move west (left)
5. Pick up the passenger
6. Drop off the passenger

Reward: The agent receives a reward of -1 for each step unless another reward condition is triggered [57]. A successful passenger drop-off yields a reward of $+20$ [57]. Executing an illegal **pickup** or **drop-off** action results in a penalty of -10 [57].

Transition: At each step, the selected action is executed, either moving the taxi in the intended direction or performing a pickup or drop-off operation [57]. The environment terminates upon a passenger drop-off [57].

4.1.3 Frozen Lake

The Frozen Lake environment is another environment provided by OpenAI Gym [58]. It consists of a 4×4 grid with a defined start position, a goal position as shown in Figure 4.3, and several holes distributed throughout the grid [58]. The agent must navigate through this environment to reach the goal while avoiding falling into the holes [58].

A key feature of this environment is its "slippery" nature. When the agent attempts to take an action, there is only a 33% chance that the intended action is executed, and a 66%

chance that the agent instead moves in a direction perpendicular to the intended one [58]. This introduces a significant amount of randomness into the environment.

Although the environment contains only 16 states (with one being the terminal state) and 4 actions, the randomness greatly increases the complexity of the problem. The agent must learn to generalize across episodes and discover the optimal path despite the inherent uncertainty. A strategy that succeeds once may not work again due to the non-deterministic transitions.



Figure 4.3: Frozen Lake Environment

Goal: Navigate to the goal location in the environment without falling into a hole [58].

Input: An integer between 0 and 15, representing the agent's current state [58].

Output: At each step, the agent must select one of four possible actions [58]:

1. Move left
2. Move down
3. Move right
4. Move up

Transition: The agent has a 33% chance of moving in the intended direction and a 66% chance of moving in a perpendicular direction [58]. The episode terminates if the agent falls into a hole or successfully reaches the goal [58].

4.1.4 CartPole

The CartPole environment is provided by OpenAI Gym [59]. It consists of a cart that can move along a one-dimensional track, with a pole attached to its center by an unactuated joint as shown in 4.4 [59]. The agent must apply forces to the cart, pushing it left or right, in order to prevent the pole from falling over [59].

This environment provides a clear and interpretable setting for evaluating agent performance in continuous control tasks, and allows for direct comparison with traditional control methods such as PID controllers.

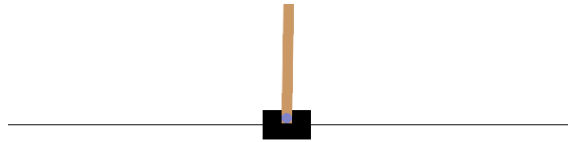


Figure 4.4: CartPole Environment

Goal: Keep the pole balanced upright on the moving cart for as long as possible [59].

Input: A four-dimensional vector representing the current state of the system [59]:

- Position of the cart
- Velocity of the cart
- Angle of the pole
- Angular velocity of the pole

Output: At each step, the agent must choose one of two possible actions [59]:

1. Push the cart to the left
2. Push the cart to the right

Reward: The agent receives a reward of 1 for every time step that the pole remains upright [59].

Transition: At each time step, the chosen action results in a force being applied to the cart, and the resulting system dynamics are simulated [59]. The environment terminates an episode under the following conditions [59]:

- The pole’s angle exceeds $\pm 12^\circ$
- The cart’s position exceeds ± 2.4 units (i.e., it reaches the edge of the visible track)
- The episode is truncated after 500 time steps

By using an established and well-understood environment, this task allows agents to be evaluated against known solutions and provides insight into their ability to handle classic control challenges.

4.1.5 Snake

The Snake environment is a custom-built control task based on the classic Snake game [60]. In this environment, the agent controls a snake that moves on a two-dimensional grid and must collect fruit [60]. Each time the snake eats a fruit, it grows longer, increasing the difficulty of survival [60]. The key challenge of this environment lies in its partial observability. Unlike previous environments that provide the agent with near-perfect state information, this custom Snake environment supplies only a limited observation, introducing uncertainty and requiring the agent to adapt.

This partial observability makes the environment especially useful for evaluating the differences between stochastic and deterministic policies and for highlighting an agent’s ability to operate under incomplete information.

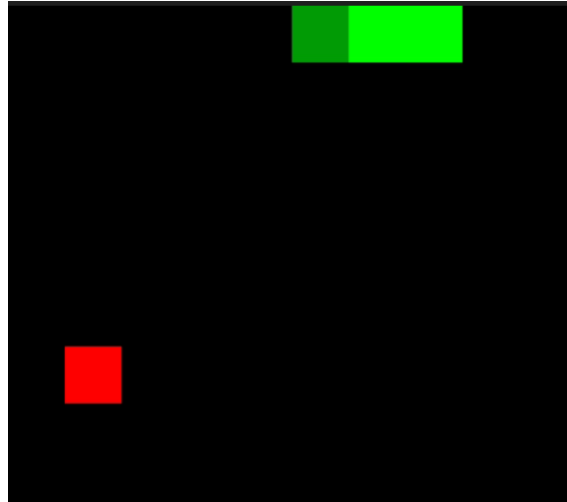


Figure 4.5: Snake Environment

Goal: Collect as many fruit as possible without dying [61].

Input: A 16-dimensional feature vector is constructed, consisting of four groups of four values, each corresponding to one of the four cardinal directions (up, down, left, right) relative to the agent’s current position. For each direction, the first value represents the number of grid cells reachable from the adjacent cell in that direction, computed using a flood-fill algorithm that treats walls and the snake’s body as obstacles. This value is

normalized by dividing by the total number of cells in the grid. The remaining three values form a one-hot encoding indicating the type of the first object encountered when traversing in that direction, which may be a wall, the snake’s body, or a piece of food.

Output: The agent selects one of the following four discrete actions at each step [61]:

1. Move up
2. Move left
3. Move right
4. Move down

Reward: The agent receives rewards based on the following criteria:

- +1 for collecting a fruit
- -0.01 per step to encourage efficient movement
- -1 upon death (colliding with itself or a wall)

Transition: At each step, the snake moves in the selected direction, and the environment updates accordingly [61]. The episode terminates if the snake’s head collides with a wall or with its own body [61].

By utilizing this custom environment, significant flexibility is granted over the reward structure and other environment parameters. In some experiments, the reward design or state representation may be modified to highlight specific learning behaviors or phenomena.

4.2 Experimental method

In this section, an overview of the method used to conduct each experiment is outlined. In Chapter 7, several experiments are conducted, each of which broadly follows the same defined procedure. Each experiment has a clearly defined experimental question that seeks to be answered by conducting the experiment. The question will be used to gain insights within the agents using the `fund_rl` library and the environment provided. By answering this question, a demonstration will be given on how insights can be gained on the agents with these methods, building towards answering the main research question. A corresponding hypothesis is then formulated as a proposed answer to that question. Following this, the experimental setup is described in detail, and the methodology for executing the experiment is provided.

Once the experiment is conducted, the results are collected and stored for analysis. An analysis of the results is then carried out, followed by a conclusion that evaluates the original hypothesis. Each experiment is implemented in code, encapsulated in a function that includes everything from the research question to the final conclusion.

The experimental code is available in the GitHub repository referenced in Appendix A.

4.3 Conclusion

This chapter established the foundation for all subsequent experimental work. It introduced the environments used, highlighting their diversity in terms of observability, control complexity, and stochasticity. These environments were carefully selected to evaluate agents across a broad spectrum of challenges.

The experimental method was also clearly defined, detailing how each experiment was structured — from the formulation of a research question and hypothesis to the setup, execution, and analysis.

Understanding the structure and operation of these environments is essential for conducting experimentation within controlled settings, as demonstrated in Chapter 7.

Altogether, this structured approach ensures that each experiment is reproducible, interpretable, and comparable, thereby laying the groundwork for the experiments presented in the following chapter.

Chapter 5

Behavioural analysis and intelligibility

The following chapter discusses the concept of intelligibility and various methods used to enhance it.

Intelligibility in RL is an important topic, particularly with the recent advances in deep reinforcement learning where deep neural networks are employed to implement learning algorithms [62]. However, the use of deep neural networks introduces several challenges and drawbacks [62]:

- They are more difficult to comprehend due to the complexity of the underlying neural network structures [9, 62].
- They are vulnerable to adversarial attacks, in which malicious users can manipulate the learned policy and cause it to perform unintended or undesirable actions. [62, 63].
- They are more challenging to train effectively [62, 64].

For these reasons, it is essential to understand how an RL agent learns and how its policy behaves, as this understanding can help address these challenges. The chapter begins by defining intelligibility and describing its main subcategories. Thereafter, it presents several methods that can be used to gain deeper insight into an RL agent’s learning process and to improve its overall intelligibility.

5.1 Intelligibility

Intelligibility is a central concept in the field of Explainable Artificial Intelligence (XAI) [62]. Although the terms explainability and interpretability are defined and interpreted in various ways across the literature, this study adopts the conventions outlined in [62]. Within

this framework, explainability refers to methods designed to generate insights into reinforcement learning models [62]. In contrast, interpretability is considered a passive property inherent to a model’s internal structure; that is, it is an attribute of the RL agent that arises from how its underlying structure is employed to learn a policy [62].

By distinguishing these two categories—explainability as a practical methodology and interpretability as a theoretical property—the framework enables a clearer and more structured analysis of each aspect [62]. The following section presents methods aimed at enhancing explainability. However, since many of these methods also provide insights into interpretability, they are collectively referred to as intelligibility methods [62].

5.2 Intelligibility methods

There are several well-defined intelligibility methods [65], including decision trees, saliency maps, region-sensitive Rainbow, counterfactual states, and others [65]. These methods are primarily designed to operate within RL agents that utilize visual or sensory inputs [65]. Furthermore, these techniques are generally aimed at local interpretability, focusing on explaining why a single action was taken [66]. In contrast, the present research focuses on methods at a global scale, which aim to explain the agent’s overall strategy and behavior [66]. Consequently, these existing methods are not always well suited for the simplified environments described in Chapter 4.

In this work, variations and simplifications of these techniques are employed alongside statistical and mathematical evaluation methods that are well established. This approach ensures that the applied intelligibility methods align appropriately with the simplicity of the environment used.

5.2.1 Performance metrics

The most fundamental approach to evaluating the performance of a RL agent is by tracking its performance metrics, with the most basic being the cumulative reward [12]. Since RL can be formulated as an optimization problem [12], and optimization problems generally aim to minimize a cost function, it follows that in RL, the objective is to maximize a corresponding reward function [12]. In this context, maximizing the cumulative reward serves as the central goal of the learning process [12]. Therefore, monitoring the cumulative reward provides valuable insight into whether the agent is improving its performance over time [12]. When exploring the agent’s strategy and behaviour, cumulative reward provides a measure of the effectiveness of the agent’s strategy and its overall behavioural performance when evaluated against achieving the goal.

As with any measurement process, there is always some degree of error and noise [67]. To account for this, both the mean cumulative reward and its standard deviation are re-

ported [68, 69], as they respectively indicate the accuracy and precision of the performance measurement to some degree [68].

Overall, because the objective of an RL agent is to maximize the cumulative reward over an episode, an increasing reward typically indicates that the agent is learning and improving [12]. This approach thus represents the standard and most direct method for assessing agent performance [12].

However, performance metrics encompass more than just the reward. They also include training time, training step count, cumulative reward, as well as the mean, maximum, minimum, and standard deviation of these values. If Agent A and Agent B both achieve the same cumulative reward, but Agent A completes training in 5 minutes while Agent B requires 30 minutes, then Agent A is arguably more efficient, particularly when training time is a concern.

In Figure 5.1, an example of a performance analysis is presented. In this example, a PPO agent is trained for 200 episodes in the CartPole environment. Throughout the training process, the reward and loss are tracked. In the top-left panel of the figure, the light-blue curve represents the raw reward values, while the darker blue curve represents the filtered version. The top-right panel shows the loss, where the light-red curve indicates the calculated loss and the dark-red curve indicates its filtered version. The bottom-left panel presents the cumulative mean reward, which steadily increases to a peak value of 200. The bottom-right panel displays the standard deviation of the reward, which gradually increases to a peak of 150, indicating a standard deviation in the reward of approximately 150 over the agent’s training period.

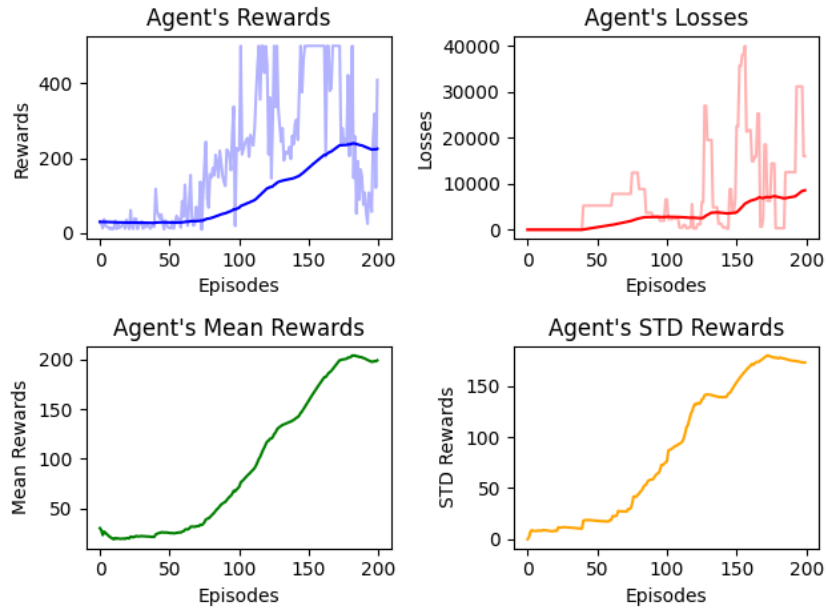


Figure 5.1: Performance analyser output example from the fund_rl library

5.2.2 Policy visualization

Policy visualization is an intelligibility method in RL that enables the policy of an agent to be displayed, thereby providing insight into the agent’s behavior in each state [12]. This technique is particularly valuable in environments with small state spaces—such as *FrozenLake*—where it is practically feasible to render every state alongside the action prescribed by the policy [12]. Since a policy maps states to actions, visualizing this mapping provides direct and explicit feedback on the agent’s intentions by converting the policy into an interpretable format [12, 70]. This approach is analogous to other feature-based intelligibility methods, which aim to make the policy more interpretable, such as those using imitation learning [70].



Figure 5.2: Policy visualisation output example from the analysis conducted using the `fund_rl` library

In Figure 5.2, an example of policy visualization is presented. A PPO agent is trained on the *FrozenLake* environment (without a slippery surface) over 237 episodes. The figure illustrates the distribution of the network across all states, with each state displaying the probability of selecting an action in each of the four directions according to the learned policy. This includes states that the agent has never visited (i.e., the four holes, which terminate the environment when reached), thereby demonstrating the policy that the

agent has learned up to this point.

5.2.3 State visitation

State visitation, also referred to as state frequency or state count, is a form of intelligibility method used in reinforcement learning [71]. In this method, the number of times a particular state is visited during training or evaluation is recorded and stored [71]. This approach provides a rich source of information for analysis and interpretation.

One of the first insights it offers is an overview of which states were visited during training, meaning it reflects the states with which the agent has interacted [71]. This information can be used to identify potential shortcomings in the agent’s learning process and highlight areas where additional training may be required [71]. It also indicates which states were most frequently visited, thereby revealing which states the agent saw most during training [71].

Furthermore, in control tasks, there typically exists a particular state or a set of states that represent the desired or stable operating conditions that the controller aims to maintain. State visitation analysis can therefore provide valuable information about how long and how often the agent remains within these preferred states, offering deeper insight into the agent’s stability, control performance, and behavioral tendencies.

Additionally, if a state or a subset of states exhibits a significantly higher visitation frequency compared to the surrounding states, this may indicate that the agent has formed a local loop within these states, repeatedly transitioning between them. Such behavior can reveal aspects of the agent’s strategy, such as the exploitation of a specific region of the state space, or it may suggest premature convergence to a suboptimal policy.

In Figure 5.3, an example of a state visitation analysis is presented. In this example, a Q-learning agent is trained for 2000 episodes in a non-slippery version of the *FrozenLake* environment. The states are numbered starting from 0 in the top-left corner, then 1, 2, 3, and so on, with the next row numbered 4, 5, 6, etc. As can be seen, several states are missing; these correspond to terminal states or states that the agent did not visit during training. Specifically, the missing states include the four holes and the goal state. The total number of states remains 16, as expected.

5.2.4 Decision boundaries

Decision boundaries are used to identify the input regions at which an agent changes its decision [72]. For example, in the inverted pendulum control problem, it is expected that the agent moves left when the pole leans to the left and moves right when the pole leans to the right [59, 72]. Therefore, the decision boundary for the pole angle is anticipated to lie near 0° , as this represents the point where the decision is expected to change [59, 72].

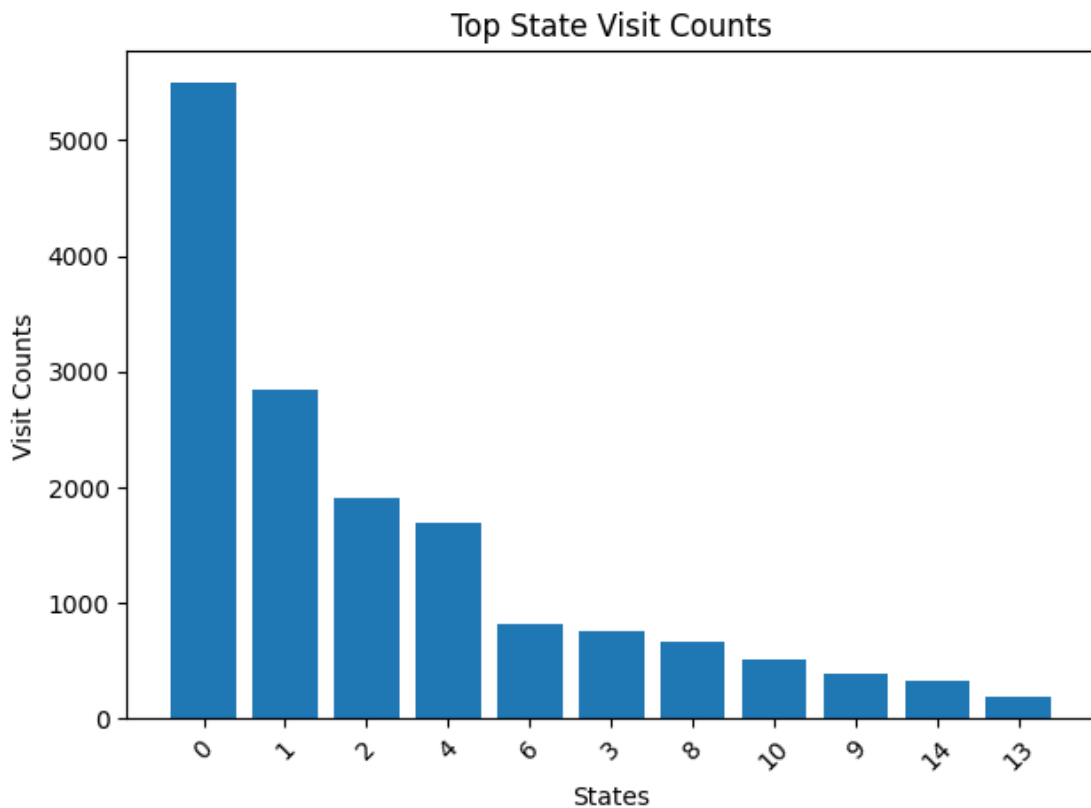


Figure 5.3: State visitation output example from the analysis conducted using the `fund.rl` library

Decision boundaries can also be interpreted as a simplified form of saliency maps. Saliency maps visually illustrate which parts of an image input the agent focuses on and which features are considered most influential in the decision-making process [65]. Similarly, decision boundaries provide insight into the relative importance of features. When a boundary is drawn between two features, one can visually infer which of the two features carries more weight in the agent’s decision process.

Analyzing decision boundaries assists in understanding how different features are utilized by the agent and provides valuable insight into how the agent may behave in previously unseen situations [72]. Furthermore, it reveals how the agent interprets various inputs, which is particularly useful in the context of behavior cloning and generalization [72].

In Figure 5.4, an example of a decision boundary analysis is presented for a PPO agent trained on the *CartPole* environment over 200 episodes. In this analysis, the decision boundary between velocity and position, ranging from -10 to 10, is examined. Blue regions indicate that the agent chooses to move right, while red regions indicate a choice to move left. Other colors represent the agent’s uncertainty in its action selection.

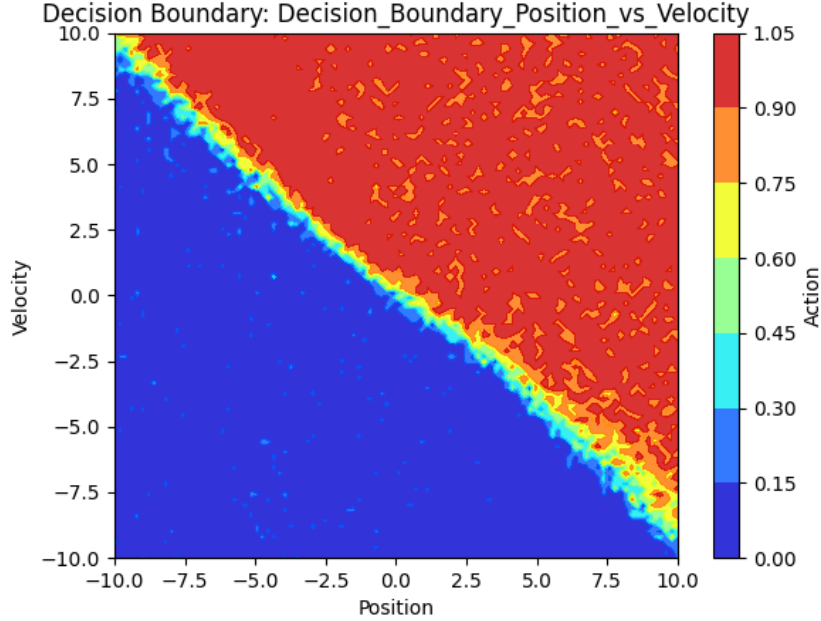


Figure 5.4: Decision boundary output example from the analysis performed using the `fund_rl` library

5.2.5 Consistency

Performance consistency in RL refers to an agent’s ability to repeatedly achieve a similar level of performance. Although this is often evaluated implicitly through performance metrics, the agent’s stability or consistency is essentially reflected in the standard deviation (STD) of the rewards obtained [68]. Since the STD represents the precision of performance, a lower STD indicates higher precision, meaning the agent consistently achieves similar performance levels [68].

In this study, consistency is defined as the agent’s stability throughout the training process—from initial exploration to the end of training—and measures how reliably the agent can reach the same level of performance. This concept aligns with interpretability objectives, as consistency is inherently influenced by the agent’s structure and hyperparameters rather than its raw performance alone, while also being affected by environmental factors [73].

For example, in the *CartPole* environment, an agent may consistently achieve similar scores because the rewards are deterministic [59]. In contrast, in the *FrozenLake* environment, the inherent stochasticity introduces randomness in rewards, resulting in a higher reward STD and, therefore, lower consistency [58].

A simple method for evaluating consistency involves training the same agent multiple times under identical settings in the same environment and recording the rewards [12, 68]. After several repetitions, the average reward and its standard deviation can be computed [68].

The STD serves as an indicator of consistency: a value of zero represents perfect consistency, whereas higher STD values indicate greater inconsistency [68].

In Figure 5.5, an example of a consistency analysis is presented. In this example, 100 PPO agents are trained over 200 episodes, and the average reward curve across all agents is computed. As shown in the figure, the dark-blue curve represents the mean reward of the agents, while the light-blue region indicates the area within one STD.

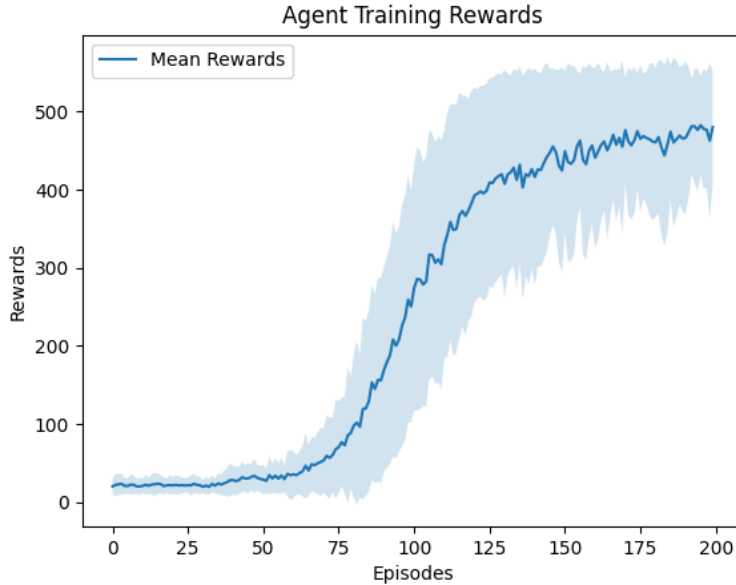


Figure 5.5: Consistency analysis example using the fund.rl library

5.2.6 Feature heatmaps

Saliency maps provide insight into which regions of an image input an agent focuses on most [70]. This is particularly useful in tasks involving image inputs [70]. However, in environments like the one used here, where no image data is involved, saliency maps offer little practical value. A similar technique can be applied instead, targeting individual features rather than image regions.

Heatmaps in reinforcement learning provide a visual method for analyzing how different features are utilized by the agent during training or evaluation. This can be achieved by maintaining a histogram for each feature across all encountered states and then plotting these histograms. If two features are considered together, a two-dimensional histogram, also known as a heatmap, can be generated. The resulting visualization outlines how the combination of those features contributes to the agent’s state space. Another approach is to capture all states and represent them on a two-dimensional grid, by taking the top two principal components. This provides a description of the 2D structure, and the two principal components can then be plotted against each other in a heatmap. This visualization illustrates how combinations of all features appear across the collected samples.

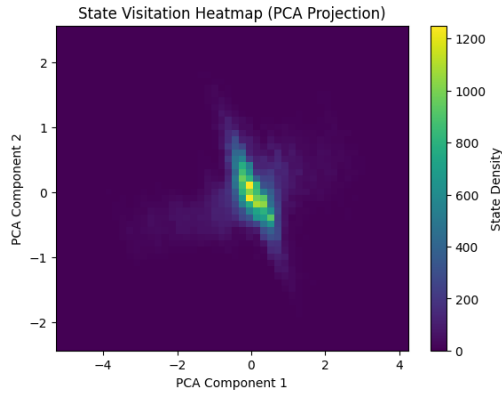


Figure 5.6: PCA heatmap output example generated using the `fund_rl` library

Heatmaps are valuable in several ways. For instance, they can help identify suboptimal policies or overlooked regions in the state space. If the heatmap reveals several bright, isolated points, it may indicate that the agent frequently visits a few distinct states without clear continuity between them. On the other hand, if the bright regions are concentrated and connected, it suggests the presence of a coherent central policy or trajectory that the agent follows consistently.

Overall, heatmaps offer an effective way to visualize the distribution of states and feature interactions, providing deeper insight into the agent’s behavior and decision-making process.

In Figure 5.6, an example of a PCA heatmap is presented, consisting of the two principal components of the states. Yellow regions indicate areas that are visited more frequently, while purple regions indicate less frequently visited states, with a gradient used to interpolate between these extremes.

In Figure 5.7, an example of a feature heatmap is shown. The diagonal panels display the histograms of each of the four features in the CartPole environment. The off-diagonal panels present 2D histograms or heatmaps illustrating the joint frequency of feature combinations, where yellow indicates more commonly visited combinations and purple indicates less frequent ones.

5.3 Conclusion

In this chapter, an overview of intelligibility was presented, including methods to understand a reinforcement learning agent both at a fundamental level, based on the agent’s structure (interpretability), and by utilizing additional tools to analyze a learned policy (explainability). Several intelligibility methods were subsequently discussed, with brief illustrative examples.

A proper understanding of how these agents function is essential for improving and build-

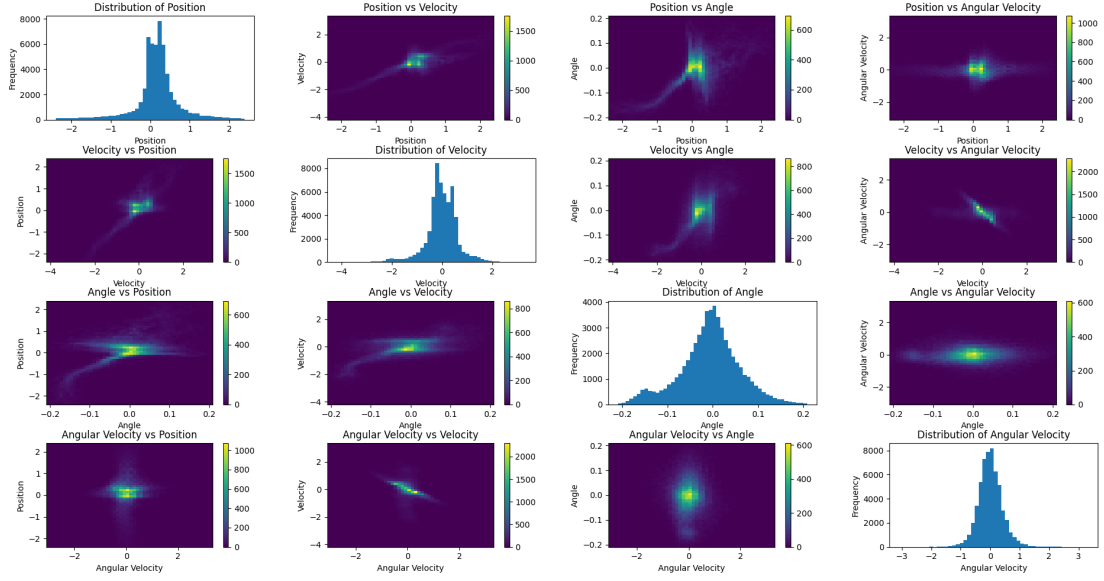


Figure 5.7: Feature heatmap output example generated using the fund_rl library

ing upon them. As previously noted, PPO is an improvement on REINFORCE, and DDQN is an improvement on Q-learning; both advancements were only possible due to a solid mathematical understanding of Q-learning and REINFORCE. Therefore, in order to further enhance DDQN and improve PPO, a clear understanding of their underlying mechanisms is paramount.

Chapter 6

Agent implementations

In the following chapter, the implementation of the code, as well as each of the individual AI agents used for this research, is outlined. The discussion begins with the code design philosophy, providing a brief overview of how the overall structure is organized and presented, and explaining the rationale behind specific design decisions. This is followed by the implementation of each of the four agents, demonstrating how the theoretical concepts in chapter 3 is translated into the actual implementation.

6.1 Code design

As part of this research, a custom library named `fund_rl` has been developed. This library has been designed with the goal of focusing on the fundamentals of reinforcement learning and improving the understanding of an agent’s internal workings, thereby enhancing explainability. To support these objectives, the library includes built-in capabilities for a variety of functions relevant to intelligibility.

Specifically, `fund_rl` provides multiple action selection methods, various agent implementations (including support for previously discussed interpretability techniques), and a buffer for storing rollouts. It also includes support for custom environments, evaluation techniques, neural network models, and sweep capabilities for optimizing hyperparameters. Additionally, it features tracing tools for tracking different metrics during training and evaluation, training utilities, and helper functions for filtering environments, wrapping them, and recording training sessions as videos.

Many components of `fund_rl` build upon existing libraries and use PyTorch for neural network models. The most unique aspect of this library lies in its agent implementations. Unlike libraries that focus on deploying highly optimized agents for industrial applications, `fund_rl` prioritizes simplicity and transparency. The agents are intentionally suboptimal by industry standards in order to make their internal processes easier to understand. This trade-off enables researchers and practitioners to gain significantly deeper insights into

agent behavior and the mechanics of reinforcement learning.

Throughout the remainder of this dissertation, flow diagrams are presented to illustrate the implementation logic, rather than displaying raw code snippets. It is important to note that each block in the flow diagrams does not represent an abstract coding concept; instead, each block explicitly and directly corresponds to either a single line of code or a small number of consecutive lines. The order of execution is preserved exactly as depicted in the diagrams, ensuring that the visual representation accurately reflects the underlying code structure.

The `fund_rl` library also includes a collection of example scripts demonstrating how to use the library in practice. For clarity, a visual example showing both the code and its corresponding flow diagram is presented below. In this example, a simple Q-learning agent is trained on the classic `Taxi-v3` environment.

Figure 6.1 and Figure 6.2 illustrate how the flow diagram directly maps to the corresponding lines of code in the implementation. This design approach ensures that each block in the flow diagram represents either a single line of code or a small, consecutive block, maintaining the exact execution order.

As shown in the figures:

- **F1 – Creates the environment:** The first step involves creating the `Taxi-v3` environment.
- **F2 – Initializes the agent:** The agent is then created and initialized. Its parameters can be configured; however, in this case, the default values are sufficient.
- **F3 – Initializes a tracker:** A tracker is initialized to monitor the agent’s performance during training.
- **F4 – Trains the agent:** The agent is trained over 2,000 episodes, with early stopping applied if it achieves an average reward of 5.00 or higher.
- **F5 – Disables the agent’s training:** The agent’s training phase is disabled, ensuring that its action selection policy performs the optimal action for evaluation.
- **F6 – Evaluates the agent’s performance:** The agent’s performance is evaluated across 100 episodes using the evaluation function.
- **F7 – Creates a new environment:** After evaluation, a new environment is created to visualize the agent’s learned behavior.
- **F8 – Executes the agent:** Finally, the agent is executed for 5 episodes in the new environment to visually demonstrate its learned policy.

This example demonstrates how each flow diagram in this research corresponds explicitly to a real implementation in `fund_rl`. The library can be downloaded from the GitHub

```

def Q_Learning():
    from fund_rl.utility.wrappers import Make_Environment
    from fund_rl.trackers.local import TLocal_Tracker
    from fund_rl.training.trainer import Train_Agent
    from fund_rl.evaluate.evaluation import Evaluate_Agent
    from fund_rl.utility.run import Run_Agent

    from fund_rl.agents.qlearning import TQ_Epsilon_Agent

    print("Q-Learning [Epsilon Greedy] Demo")

    # Create the environment
    Environment_Name = "Taxi-v3"
    Environment = Make_Environment(Environment_Name)

    # Initialize the Agent
    Agent = TQ_Epsilon_Agent(Environment)

    # Initialize the Tracker
    Tracker = TLocal_Tracker()

    # Train the Agent
    Train_Agent(Environment, Agent, Tracker, Episodes=2_000, Early_Stopping=5.00)

    # Disable training mode for evaluation
    Agent.Is_Training = False

    # Evaluate the Agent
    Evaluate_Agent(Environment, Agent, Tracker, Episodes=100)

    # Create the environment with rendering
    Environment = Make_Environment(Environment_Name, render_mode="human")

    # Run a few episodes with rendering
    Run_Agent(Environment, Agent, Episodes=5)

```

Figure 6.1: Example implementation using `fund_rl` to train a Q-learning agent on the `Taxi-v3` environment.

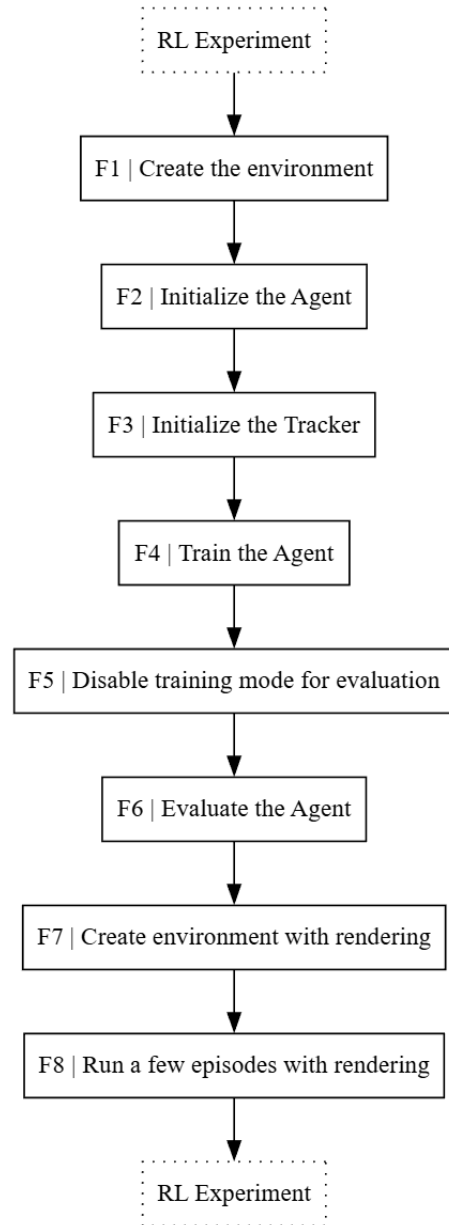


Figure 6.2: Flow diagram matching the implementation steps in the Q-learning example.

repository linked in the appendix. Users can visually inspect each implementation alongside its corresponding flow diagram for better understanding and practical insight.

Throughout the experiments, three essential functions are used frequently. For this reason, they are discussed briefly. The first two are the `Train_Agent` and `Evaluate_Agent` functions. These two operations are nearly identical; the primary difference is a safety check. In the training function, the safety check ensures that the agent is in training mode, whereas in the evaluation function, it ensures that the agent is not in training mode. This relationship is illustrated in Figure 6.3, where the flows have been combined due to their similarity.

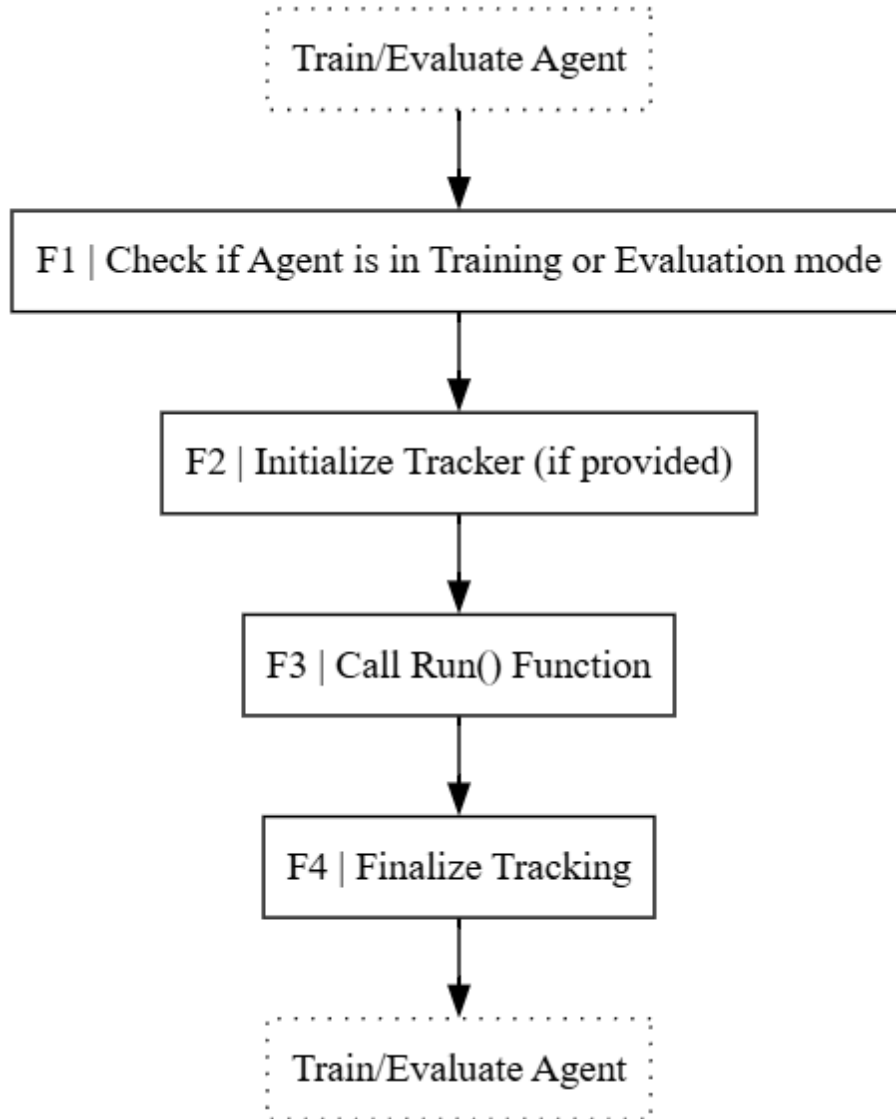


Figure 6.3: Combined flow of the training and evaluation functions.

The flow of the training and evaluation functions can be further broken down into the following four sub-functions:

- **F1 – Check training mode:** The function first checks whether the agent is operating in training or evaluation mode, depending on the intended phase of execution.

- **F2 – Initialize the tracker:** The tracker is initialized with default values or additional parameters, if provided.
- **F3 – Call the Run function:** The Run function is executed, which performs the Markov Decision Process (MDP) and runs the agent within the environment.
- **F4 – Finalize tracking:** Tracking is finalized once training is completed, provided that a tracker has been initialized.

The third essential function is the `Run` function, called in both the training and evaluation functions as F3, as shown in Figure 6.3. The `Run` function is responsible for executing the agent in an environment using the MDP. Its flow is illustrated in Figure 6.4 below.

As shown in Figure 6.4, the function performs the following steps:

- **F1:** Input validation.
- **F2:** Start the training loop.
- **F3:** Reset the environment.
- **F4:** Initialize variables to track rewards and steps.
- **F5:** Episode loop.
- **F6:** Render the environment, if applicable.
- **F7:** Agent selects action based on the current state.
- **F8:** Take action in the environment.
- **F9:** Update the agent with the transition.
- **F10:** Log data if a tracker is provided.
- **F11:** Update total reward.
- **F12:** Update the state.
- **F13:** Check if the episode is done.
- **F14:** Log additional episode data.
- **F15:** Update agent's reward attribute.
- **F16:** Log agent attributes.
- **F17:** Check for early stopping.

In the following section, each agent will be discussed with an outline of their important code sections.

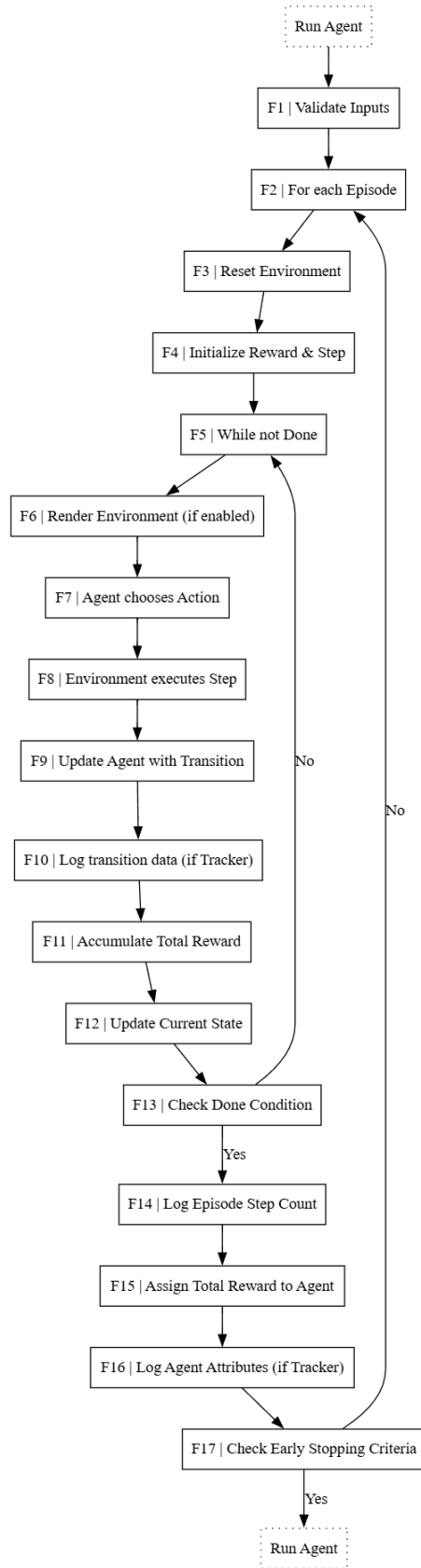


Figure 6.4: Flow of the Run function.

6.2 Agents

This section discusses the implementation details of each agent. It is important to note that these agents are not optimized for production use. Instead, they are implemented in a transparent and modular manner to facilitate a clearer understanding of their inner workings. Consequently, each agent follows a similar general structure. Due to this uniformity, certain agents—particularly PPO—do not leverage specific features that would otherwise enhance their performance.

Each agent is implemented by extending a base class named `TAgent`, as illustrated in Figure 7.1.

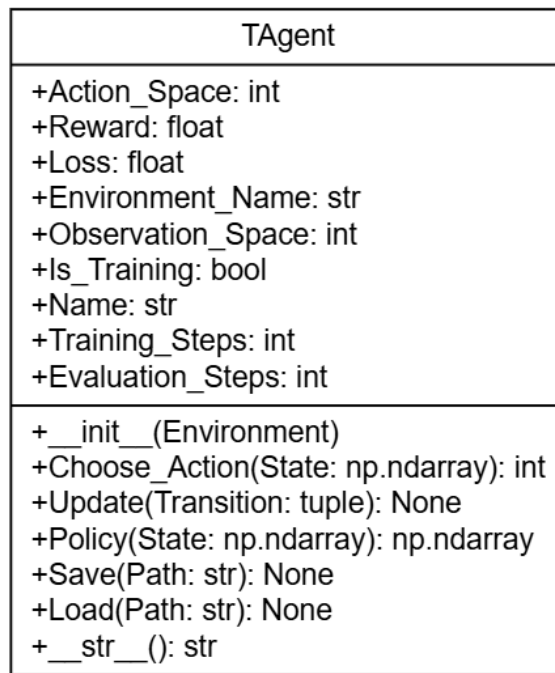


Figure 6.5: Abstract base class `TAgent` used for implementing individual agents.

As shown in the figure, the base class includes several built-in parameters essential to the operation of nearly any agent. These parameters include:

- **Action Space** – Represents the total number of actions available to the agent.
- **Reward** – Denotes the most recently recorded reward.
- **Loss** – Indicates the most recently recorded loss.
- **Environment Name** – Specifies the name of the environment in which the agent operates.
- **Observation Space** – Defines the shape of the observation space perceived by the agent.
- **Is Training** – Indicates whether the agent is currently in training mode.

- **Name** – Provides the identifier or name of the agent.
- **Training Steps** – Represents the total number of steps completed while in training mode.
- **Evaluation Steps** – Represents the total number of steps completed while in evaluation mode.

The class also defines a set of functions that must be implemented by each agent:

- `__init__()`: The constructor, which must accept at least the environment as an input (not relevant to the core implementation and therefore not discussed further).
- `Choose_Action(state: array) → int`: Receives a state and returns an integer representing the chosen action.
- `Update(Transition)`: Takes in a transition and performs the necessary updates.
- `Policy(State) → array`: Given a state, returns an array representing the probability distribution over possible actions (not relevant to the core implementation and therefore not discussed further).
- `Save()` and `Load()`: Functions for saving and loading the agent's state (not relevant to the core implementation and therefore not discussed further).
- `__str__()`: Overridden string method used to print the agent's settings (not relevant to the core implementation and therefore not discussed further).

The following sections present the specific implementation details for the `Choose_Action`, and `Update` functions for each individual agent. Additional utility functions used by the agents are also discussed as needed.

6.2.1 REINFORCE

This section discusses the implementation of the REINFORCE agent. It begins with the agent's class diagram, which illustrates its structure, additional functions, and key variables. Thereafter, the action selection, update procedure, and training functions are described in detail.

Class Diagram

Figure 6.6 presents the class diagram of the REINFORCE agent. From this diagram, it is evident that the agent includes several additional variables and two supplementary functions.

The additional variables are defined as follows:

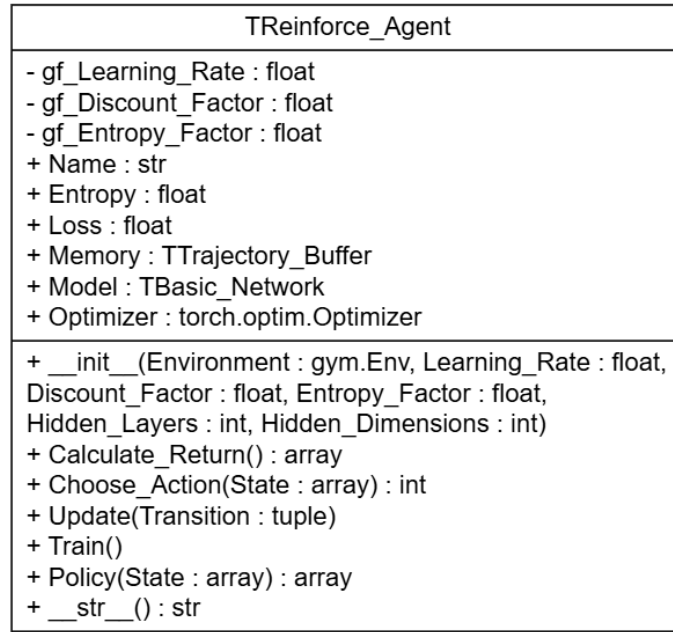


Figure 6.6: Class diagram of the REINFORCE agent showing its main components and their relationships.

- **Learning rate** - The learning rate used by the optimizer for training the neural network.
- **Discount factor** - The γ value used to calculate the cumulative discounted reward, as defined in Equation 2.1.
- **Entropy factor** - The coefficient determining the influence of entropy regularization when computing the agent's loss.
- **Entropy** - A variable used to store the computed entropy for monitoring and analysis purposes.
- **Loss** - A variable used to record the calculated loss during training for tracking and evaluation purposes.
- **Memory** - A trajectory buffer that stores all transitions of an entire episode, including state, action, reward, next state, and termination flag.
- **Model** - The underlying neural network that parameterizes the agent's policy. The weights and biases of this network define the policy function, as described in Equations 2.10, 2.13, and 2.12.
- **Optimizer** - The optimization algorithm used to update the neural network parameters via backpropagation.

Action Selection

Figure 6.7 illustrates the action selection process of the REINFORCE agent. As shown, this process follows four sequential functional steps.

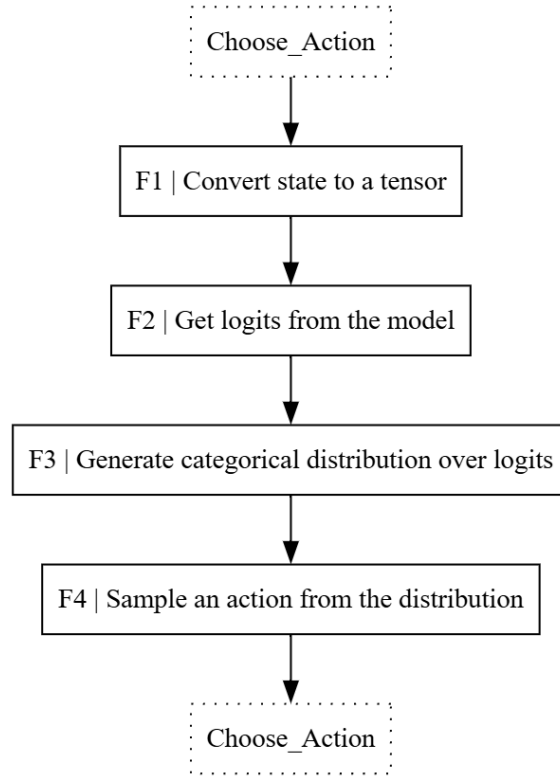


Figure 6.7: Flow diagram illustrating the action selection process in the REINFORCE agent, showing how actions are chosen based on the current state and policy.

The steps are described as follows:

- **F1 – State Conversion:** The current state is converted into a tensor format that can be processed by the neural network.
- **F2 – Obtaining Logits:** The neural network processes the state tensor and outputs the corresponding logits ¹, which represent the unnormalized action preferences.
- **F3 – Generating Action Probabilities:** The logits are transformed into a categorical probability distribution over possible actions.
- **F4 – Sampling Action:** An action is sampled from the resulting categorical distribution according to the computed probabilities.

¹Logits are the raw outputs from the neural network.

Update Procedure

Figure 6.8 illustrates the update procedure of the REINFORCE agent. In this diagram, it is evident that the agent calls a dedicated training function. This is necessary because REINFORCE is a Monte Carlo simulation, which requires a complete episodic trajectory before updating the agent's policy with all the collected data at once.

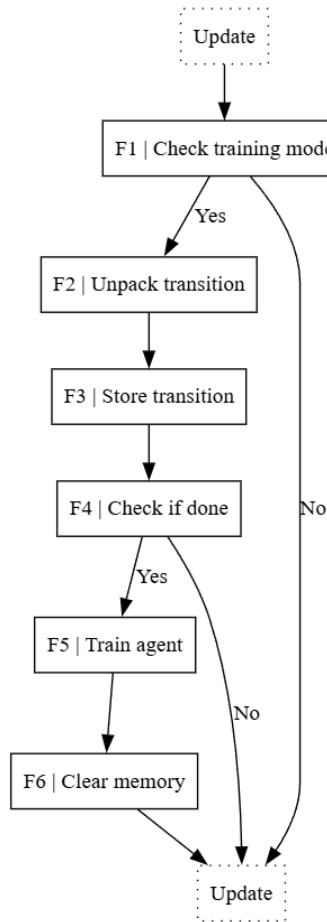


Figure 6.8: Flow diagram of the update procedure for the REINFORCE agent, illustrating how the policy is updated based on observed transitions and rewards.

The update procedure follows these steps:

- **F1 – Training Mode Check:** The agent first verifies whether it is operating in training mode. If it is not, the update procedure is skipped.
- **F2 – Unpack Transition:** The transition data are unpacked, allowing access to the individual components: state, action, reward, next state, and done flag.
- **F3 – Store Transition:** The transition is stored in the agent's memory buffer.
- **F4 – Check Episode Completion:** The agent verifies whether the current episode has been completed. If the episode is not complete, the remainder of the update function is skipped.

- **F5 – Train Agent:** Once the episode is complete, the agent calls the training function. This function uses all transitions collected in memory to update the policy network parameters.
- **F6 – Clear Memory:** After training, the agent clears its memory to prepare for storing transitions from the next episode.

Training Process

Figure 6.9 illustrates the training process of the REINFORCE agent. Although the training function appears lengthy, it consists of ten sequential functional steps executed in order to update the agent after a complete episode’s worth of data has been collected.

The cumulative discounted return for each time step t is defined as:

$$G_t = \sum_{k=0}^{T-t} \gamma^k R_{t+k} \quad (6.1)$$

The sequential steps in the training process are as follows:

- **F1 - Extract Trajectory:** All transitions stored in the agent’s memory buffer are extracted. The cumulative discounted returns G_t are calculated for each state using the return function.
- **F2 - Obtain Logits:** The neural network processes each state in the trajectory to obtain the corresponding logits.
- **F3 - Compute Action Probabilities:** The logits are transformed into action probabilities using a categorical distribution.
- **F4 - Compute Log Action Probabilities:** As required by the policy gradient theorem (Equation 2.13), the logarithm of each action probability is calculated.
- **F5 - Calculate Entropy:** The entropy of the action probability distribution is computed to encourage exploration.
- **F6 - Baseline Scaling:** In this implementation, the baseline is the mean of the returns, which is zero here. This step is included for completeness, as other implementations may use the baseline to reduce variance in the advantage estimates.
- **F7 - Compute Advantage:** The advantage is calculated as the difference between the return and the baseline.
- **F8 - Compute Loss:** The loss is computed using a modified version of Equation 2.13, resulting in the negative mean of the advantage multiplied by the log of the action probabilities.

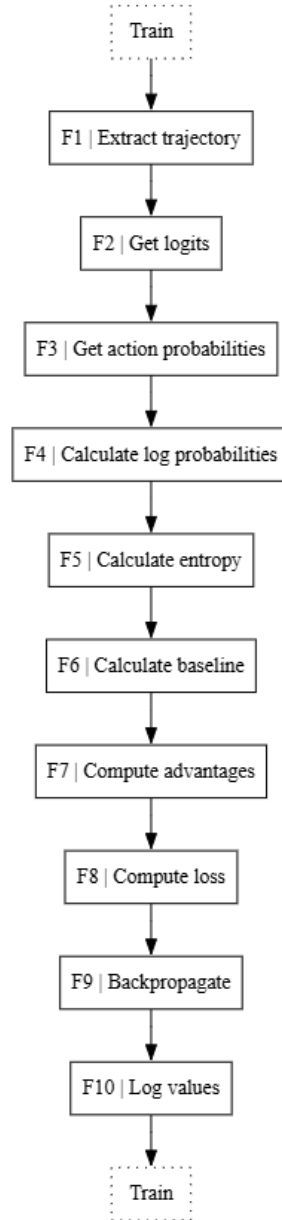


Figure 6.9: Flow diagram showing the training process of the REINFORCE agent, including initialization, episode loop, and policy updates.

- **F9 - Backpropagation and Optimization:** The optimizer performs backpropagation and updates the neural network parameters.
- **F10 - Logging:** Finally, the loss and entropy values are recorded for monitoring and analysis purposes.

This section outlined the implementation details of the REINFORCE agent and demonstrates how it utilized the policy gradient theorem. Being a Monte Carlo method, the agent collects complete episodic trajectories to train its neural network. Through this process, the agent iteratively improves its policy in order to converge towards the optimal policy.

6.2.2 Q-Learning

This section discusses the implementation of the Q-learning agent. It begins with the class diagram, followed by the action selection function, and concludes with the update function. There are two variations of the Q-learning algorithm, namely the ε -greedy Q-learning and the UCB Q-learning approaches. Since these variants only modify the action selection mechanism and not the core implementation of the agent, and because only the ε -greedy Q-learning method is used in the following sections, the discussion focuses solely on the ε -greedy Q-learning agent.

The Q-learning agent inherits from both a `QAgent` class and an `EpsilonSelection` class. The `QAgent` class, in turn, inherits from a `TabularAgent` class. However, this hierarchical structure can be flattened into a single Q-learning class for simplicity, and it is therefore presented in that form.

Class Diagram

The class diagram in Figure 6.10 illustrates the structure of the Q-learning agent, including its main components and functions.

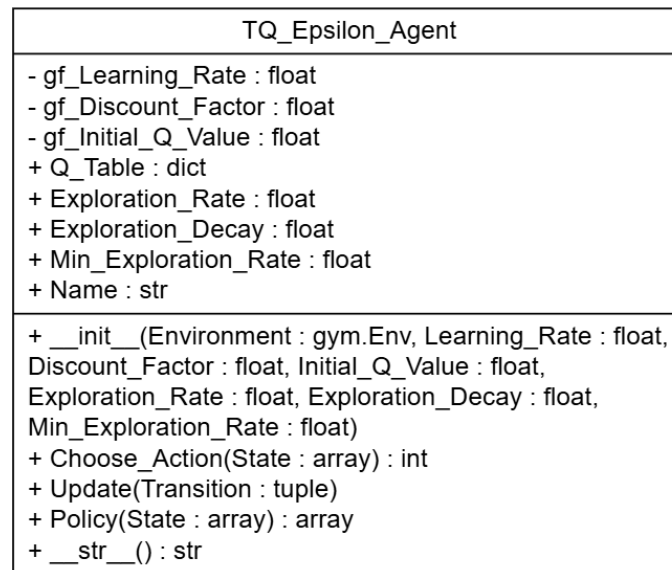


Figure 6.10: Class diagram of the Q-learning agent.

As shown, the Q-learning agent contains several additional variables, which are defined as follows:

- **Learning rate** - Corresponds to the parameter α in Equation 2.9.
- **Discount factor** - Represented by γ in Equation 2.9.
- **Initial Q-value** - Specifies the initial value assigned to each Q-value when a new state is encountered.

- **Q-table** - A table in which each state serves as an index, and the associated values are lists of all possible actions and their corresponding Q-values, thereby forming the action-value function.
- **Exploration rate** - Defines the ε -greedy action selection parameter.
- **Exploration decay** - Determines the rate at which the exploration rate decreases after each episode.
- **Minimum exploration rate** - Indicates the lowest permissible value that the exploration rate can reach during training.

Action Selection

The action selection process is illustrated in Figure 6.11. This diagram outlines how the agent determines which action to take by following four functional steps.

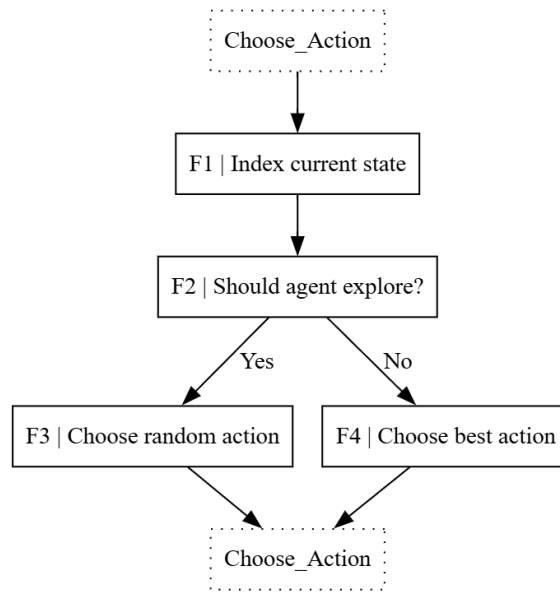


Figure 6.11: Flow diagram of the action selection process in the Q-learning agent.

The steps are as follows:

- **F1 – State Indexing:** The current state is converted into a suitable format that can be used as an index within the Q-table. This ensures that any state, regardless of its original data type, can be represented consistently in the dictionary structure.
- **F2 – Exploration Check:** The agent determines whether to explore or exploit. It generates a random value between 0 and 1. If this value is less than the exploration rate (ε) and the agent is in training mode, the exploration branch is followed.
- **F3 – Exploration:** If the exploration condition is met, the agent selects a random action from the available action set.

- **F4 – Exploitation:** Otherwise, the agent selects the action with the highest Q-value, as defined by the current action-value function in the Q-table.

Update Procedure

Figure 6.12 illustrates the update procedure of the Q-learning agent. It shows how the agent updates its Q-values based on observed state transitions and received rewards. Although the process consists of ten functional steps, these steps are executed sequentially without any conditional branching.

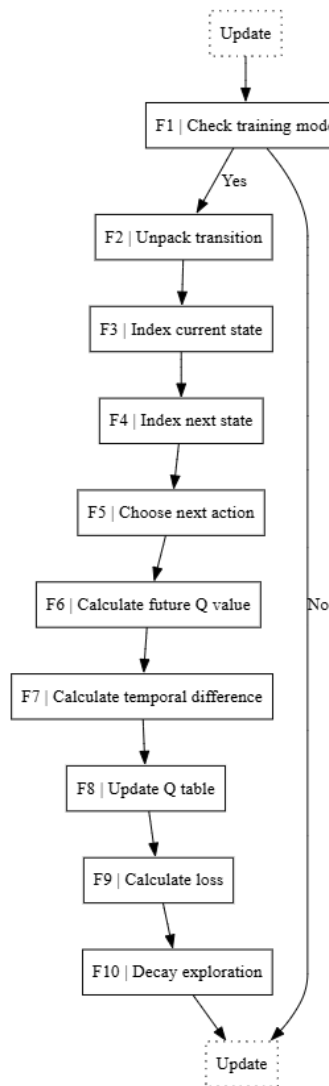


Figure 6.12: Flow diagram of the Q-value update procedure.

The sequential update process is described as follows:

- **F1 – Training Mode Check:** The agent first verifies whether it is operating in training mode. If it is not, the update procedure is skipped.
- **F2 – Unpack Transition:** The transition data are unpacked. This includes the

current state, the action taken, the reward received, the next state reached, and an indicator specifying whether the environment has terminated.

- **F3 – Index Current State:** The current state is converted into an appropriate index format for accessing the Q-table.
- **F4 – Index Next State:** The next state is similarly indexed to allow retrieval of its associated Q-values.
- **F5 – Select Next Action:** The action with the highest Q-value in the next state is selected.
- **F6 – Calculate Future Q-Value:** The future Q-value is retrieved from the Q-table using the next state and the selected next action.
- **F7 – Compute Temporal Difference:** The temporal difference (TD) error is calculated using the first part of Equation 2.9.
- **F8 – Update Q-Table:** The Q-table is updated according to the remainder of Equation 2.9, adjusting the Q-value for the current state–action pair.
- **F9 – Calculate Loss:** For monitoring purposes, the loss is computed as the squared temporal difference.
- **F10 – Decay Exploration Rate:** If the episode has terminated, the exploration rate is decayed according to the predefined decay factor.

This section outlines the implementation of the Q-learning agent and demonstrates how it utilizes the Q-table and temporal difference learning (Equation 2.9) to update and improve its state-action value function. By iteratively refining these values, the agent converges towards the optimal action-value function as defined by Equation 2.7.

6.2.3 DDQN

In this section, the implementation of the Double Deep Q-Network (DDQN) agent is discussed. The section begins with the class diagram outlining the agent’s parameters and additional functions. Thereafter, the agent’s action selection, update procedure, and training function are described in detail.

Class Diagram

Figure 6.13 presents the class diagram of the DDQN agent. As shown, the agent includes several additional parameters and one additional function compared to the base agent class.

The parameters are defined as follows:

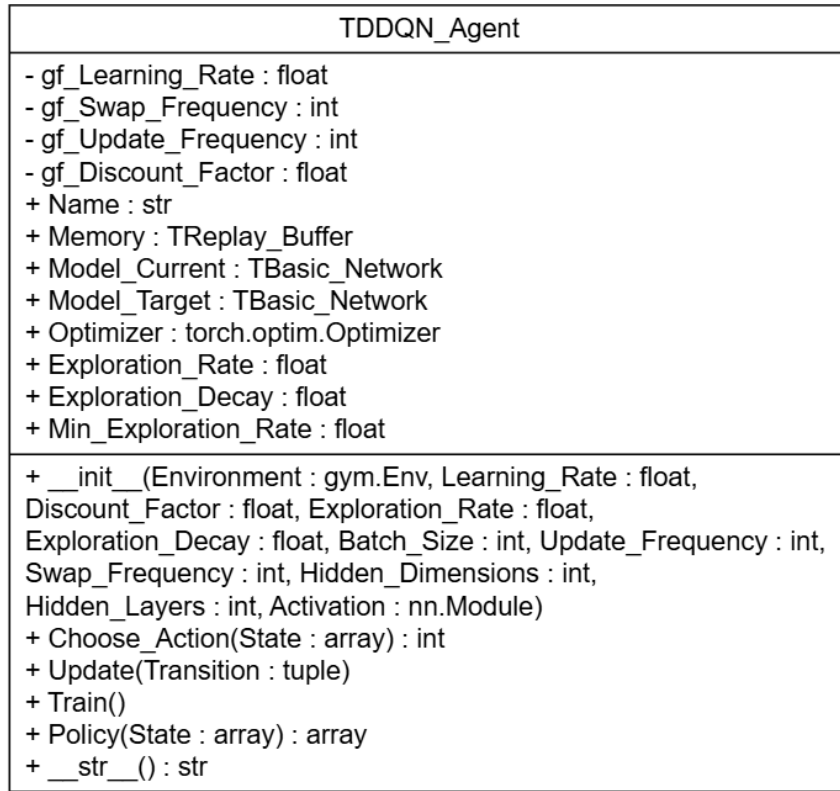


Figure 6.13: Class diagram of the DDQN agent, illustrating key parameters and structural components.

- **Learning Rate** – The learning rate of the optimizer used to train the neural network.
- **Swap Frequency** – The number of steps after which the target and current neural network models are synchronized. This mechanism distinguishes DDQN from standard DQN.
- **Update Frequency** – The number of steps after which the current neural network is updated using the target network. This is another characteristic feature of DDQN.
- **Discount Factor (γ)** – The discount rate applied in the temporal difference learning equation (see Equation 2.9).
- **Memory** – A replay buffer where the agent stores all transitions for experience replay.
- **Model (Current)** – The primary neural network used for action selection.
- **Model (Target)** – The target neural network used to update the current network. Unlike DQN, which uses a single network, DDQN maintains two separate networks to reduce overestimation bias.
- **Optimizer** – The optimization algorithm applied during backpropagation.

- **Exploration Rate (ϵ)** – The probability of selecting a random action under the ϵ -greedy exploration strategy.
- **Exploration Decay** – The rate at which the exploration rate decreases after each completed episode.
- **Minimum Exploration Rate** – The minimum threshold that the exploration rate can reach during training.

Action Selection

Figure 6.14 illustrates the action selection procedure of the DDQN agent. From this flow diagram, it is evident that the agent's decision-making process consists of four functional steps.

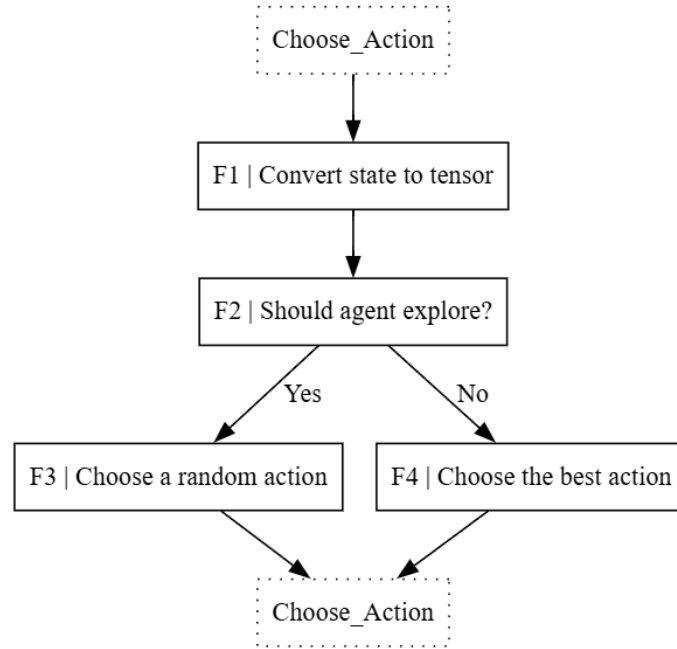


Figure 6.14: Flow diagram of the action selection process in the DDQN agent.

The action selection process proceeds as follows:

- **F1 – State Conversion:** The agent first converts the current environment state into a tensor so that it can be processed by the neural network.
- **F2 – Exploration Check:** The agent then determines whether to explore or exploit. A random number between 0 and 1 is generated, and if this value is less than the current exploration rate (ϵ) *and* the agent is in training mode, it will explore. Otherwise, it will exploit its learned knowledge.
- **F3 – Exploration:** If the agent chooses to explore, it selects a random action from the available action space.

- **F4 – Exploitation:** If the agent chooses to exploit, it selects the action associated with the highest Q-value as predicted by the current neural network.

Update Procedure

Figure 6.15 illustrates the update procedure of the DDQN agent. Similar to the REINFORCE agent, the DDQN agent utilizes a dedicated training function. However, unlike REINFORCE, the DDQN approach relies on a replay memory buffer from which samples are drawn for training. The update procedure therefore consists of five functional steps executed where appropriate.

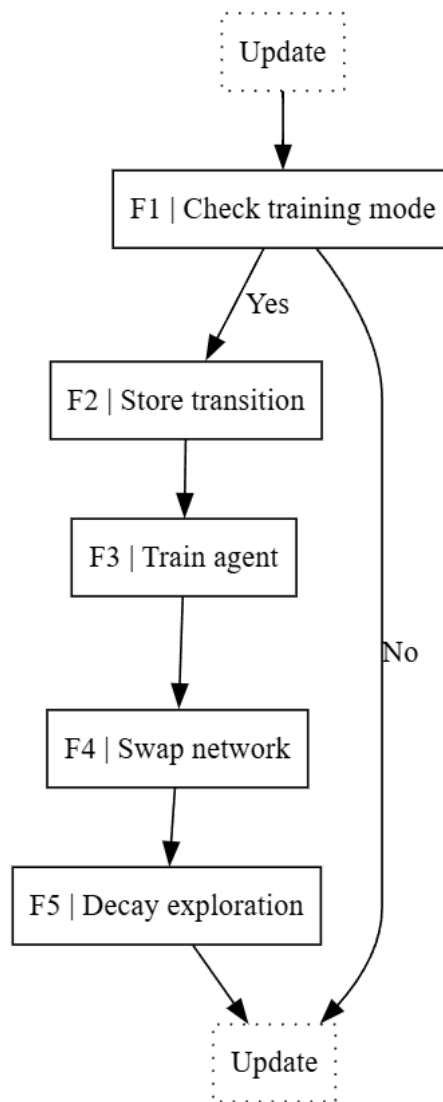


Figure 6.15: Flow diagram of the update procedure in the DDQN agent.

The update procedure proceeds as follows:

- **F1 – Check Training Mode:** The agent first verifies whether it is currently in training mode. If not, the update function is skipped.

- **F2 – Store Transition:** The current transition (state, action, reward, next state, and done flag) is stored in the replay memory buffer.
- **F3 – Train Agent:** A check is then performed to determine whether it is time to train the agent. If the number of steps since the last update meets or exceeds the predefined update frequency, the training function is executed using sampled experiences from memory.
- **F4 – Swap Networks:** The agent checks whether the weights of the current network should be copied to the target network. This occurs after a set number of training iterations, defined by the swap frequency.
- **F5 – Decay Exploration Rate:** Finally, at the end of each episode, the exploration rate (ϵ) is decayed according to the exploration decay parameter, ensuring a gradual shift from exploration to exploitation over time.

Training Process

Figure 6.16 illustrates the training procedure of the DDQN agent. As shown, this process consists of a sequential series of eleven steps, all executed in order without any branching logic. The DDQN training procedure builds upon the Double DQN architecture by decoupling action selection and evaluation between the current and target networks, ensuring more stable learning.

The training process proceeds as follows:

- **F1 – Sample Batch:** A batch of transitions is randomly sampled from the experience replay memory. The batch size is determined by the parameter specified during the agent’s initialization.
- **F2 – Convert to Tensors:** Each element of the batch is converted into tensors and separated into individual arrays of states, actions, rewards, next states, and done flags to ensure proper formatting for neural network processing.
- **F3 – Get Current Q-Values:** The current Q-values for all sampled states are computed using the current neural network model.
- **F4 – Select Q-Values:** For each transition in the batch, the Q-value corresponding to the action taken is extracted from the current Q-values.
- **F5 – Compute Next-State Q-Values:** The next-state Q-values are computed using the *current* neural network model.
- **F6 – Determine Next Actions:** The next actions corresponding to the maximum Q-values from the current model are selected. This represents the action that the current policy would choose in each next state.

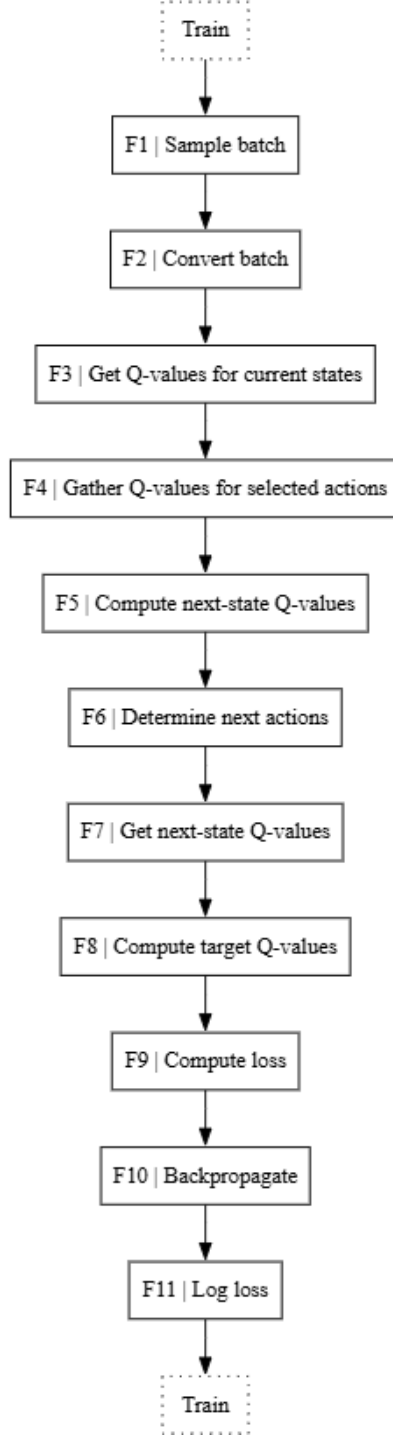


Figure 6.16: Flow diagram of the training procedure for the DDQN agent.

- **F7 – Obtain Target Q-Values:** Using the selected next actions, the target Q-values are obtained from the *target* neural network.
- **F8 – Compute Target Values:** The target Q-values are computed according to the temporal-difference learning principle (Equation 2.9):

$$Q_{\text{target}} = r + \gamma \cdot Q_{\text{target_next}}$$

where r is the observed reward, and γ is the discount factor.

- **F9 – Compute Loss:** The mean squared error (MSE) loss is calculated between the selected current Q-values and the computed target Q-values.
- **F10 – Backpropagation:** The optimizer performs backpropagation and an optimization step to update the parameters of the current neural network.
- **F11 – Log Loss:** Finally, the computed loss value is logged for monitoring and diagnostic purposes.

6.2.4 PPO

In this section, the implementation of the PPO agent is discussed. The discussion begins with its class diagram, outlining the agent’s parameters and additional functions. Thereafter, the action selection, update procedure, and training function are examined in detail.

Class Diagram

From Figure 6.17, it is evident that the PPO agent includes a large number of parameters as well as an additional training function, similar to the REINFORCE and DDQN agents.

The parameters are as follows:

- **Learning Rate:** Defines the step size used by the optimizer during gradient descent on the neural network.
- **Discount Factor (γ):** The discount rate used in the policy gradient equations when calculating the cumulative returns, as shown in Equation 2.13.
- **Policy Clip Value:** A unique PPO-specific parameter that limits the extent to which the new policy can deviate from the old policy during an update. This prevents excessively large policy updates and stabilizes training.
- **Value Function Coefficient:** Determines the contribution of the value function estimation term to the total policy loss.
- **Entropy Coefficient:** Controls the influence of the entropy term in the loss function. Higher values encourage greater exploration (by maintaining higher entropy), while lower values favor exploitation by reducing entropy.
- **GAE λ :**² The λ parameter used in the Generalized Advantage Estimation (GAE) function. This acts as a secondary discount factor controlling the bias-variance trade-off in advantage estimation.

² $\hat{A}_t = \delta_t + (\gamma\lambda)\hat{A}_{t+1}$, where δ_t is the temporal difference, γ is the discount factor, and λ is the GAE discount factor [74].

TPPO_Agent
- gf_Learning_Rate : float - gf_Discount_Factor : float - gf_Policy_Clip : float - gf_Value_Factor : float - gf_Entropy_Factor : float - gf_GAE_Lambda : float + Loss : float + Entropy : float - gi_Batch_Size : int - gi_Minibatch_Size : int - gi_Iterations : int + Name : str + Memory : TTrajectory_Buffer + Actor_Critic : TActor_Critic_Network_Separate + Optimizer : torch.optim.Optimizer
+ __init__(Environment : gym.Env, Learning_Rate : float, Discount_Factor : float, Hidden_Layers : int, Hidden_Dimensions : int, Batch_Size : int, Minibatch_Size : int, Iterations : int, Policy_Clip : float, Value_Factor : float, Entropy_Factor : float, GAE_Lambda : float) + Calculate_GAE(Rewards : array, Values : array, Dones : array) : array + Choose_Action(State : array) : int + Policy(State : array) : array + Update(Transition : tuple) + Train() + __str__() : str

Figure 6.17: Class diagram of the PPO agent, illustrating its key parameters, memory components, and neural network architecture.

- **Loss:** A variable used to store the most recently computed loss for logging and monitoring purposes.
- **Entropy:** Stores the most recently calculated entropy value for tracking purposes.
- **Batch Size:** Defines how many samples are collected before a training iteration begins.
- **Mini-Batch Size:** Specifies the number of samples drawn from the full batch during each mini-batch training iteration.
- **Iterations:** Determines how many epochs (complete passes) the training algorithm runs over the collected batch during one update.
- **Memory:** A trajectory buffer used to store transitions (state, action, reward, next state, and done flag) until a training cycle is triggered.
- **Actor-Critic Network:** Two neural network architectures consisting of an *actor* for the policy function and a *critic* for the value function estimation.

- **Optimizer:** The optimizer used to perform backpropagation and update the parameters of the actor-critic neural network during training.

Action Selection

In Figure 7.1, the action selection function of the PPO agent is illustrated. As evident from the figure, this procedure exactly follows that of the REINFORCE agent. This similarity highlights how both methods belong to the class of policy gradient algorithms, which directly learn a parameterized policy. The primary difference between the two lies not in how actions are selected, but rather in how the agents are trained and how their objective functions are optimized.

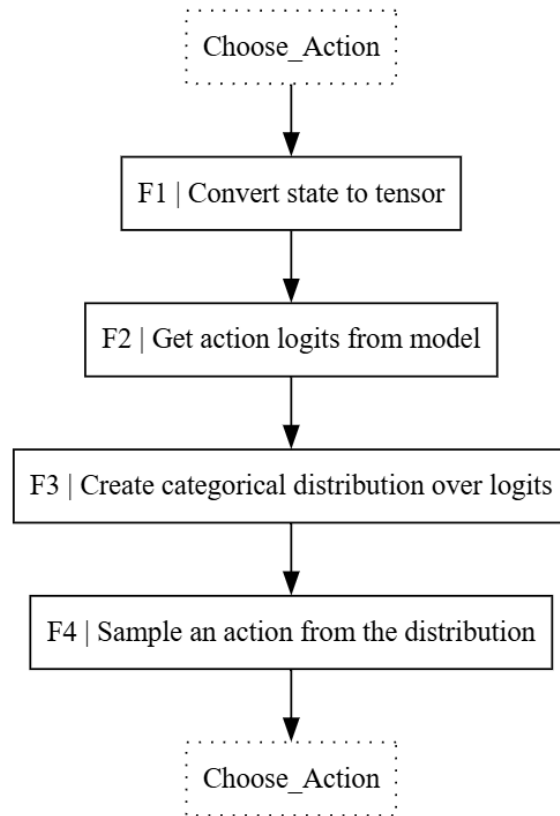


Figure 6.18: Flow diagram of the action selection process in the PPO agent.

The steps are described as follows:

- **F1 – State Conversion:** The current state is converted into a tensor format that can be processed by the neural network.
- **F2 – Obtaining Logits:** The neural network processes the state tensor and outputs the corresponding logits, which represent the unnormalized action preferences.
- **F3 – Generating Action Probabilities:** The logits are transformed into a categorical probability distribution over possible actions.

- **F4 – Sampling Action:** An action is sampled from the resulting categorical distribution according to the computed probabilities.

Update

In Figure 6.19, the update procedure of the PPO agent is illustrated. Similar to REINFORCE, PPO falls under the category of Monte Carlo methods, meaning it requires complete episodic rollouts to compute updates. However, this can make training unstable due to the long trajectories involved. To address this, PPO introduces a hybrid approach: instead of waiting for a full episode to complete, it performs updates after a certain number of samples (mini-batches), improving stability and sample efficiency.

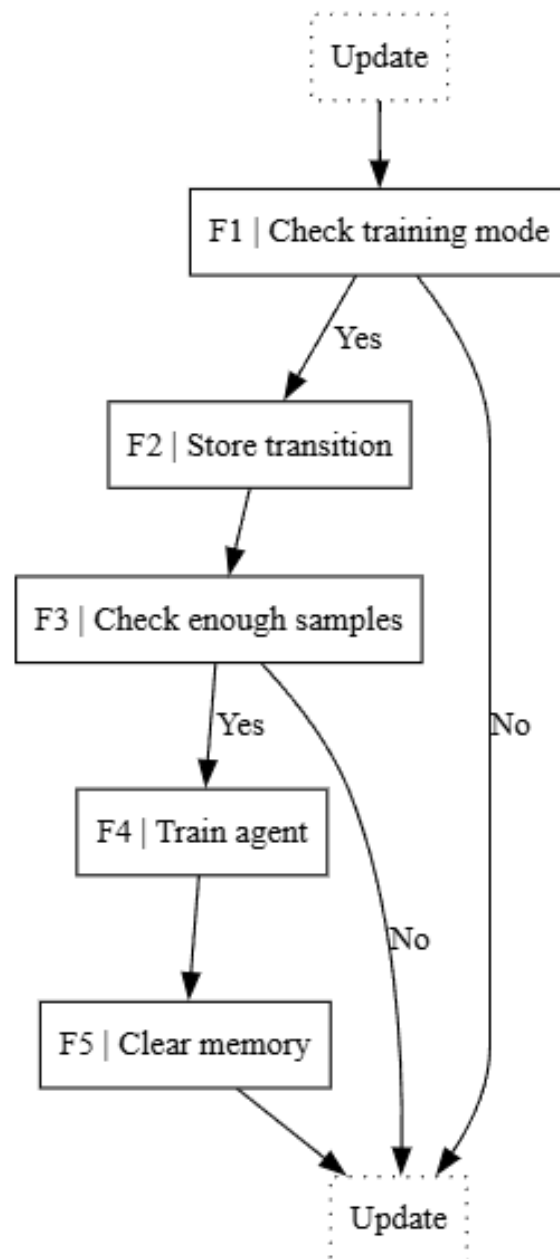


Figure 6.19: Flow diagram of the update procedure in the PPO agent.

F1 - Check Training Mode: The first step verifies that the agent is in training mode. If it is not, the remainder of the update procedure is skipped.

F2 - Store Transition: The received transition (state, action, reward, next state) is stored in memory for later sampling.

F3 - Check Samples: The agent checks whether enough samples have been collected to perform training. If not, the update process terminates early.

F4 - Train Agent: Once sufficient samples are available, the agent performs a training step using the collected data.

F5 - Clear Memory: After training, the memory buffer is cleared to make room for a new set of samples for the next training cycle.

Train

Figure 6.20 illustrates the flow of the training function. The reason for a separate training function for PPO, similar to DDQN or REINFORCE, is that PPO first collects samples before updating its policy. As shown, the training process for PPO is considerably longer than that of the other agents. It is important to note that this PPO implementation does not include all features of the Stable Baseline’s implementation; rather, it focuses on the core concepts that define PPO, including mini-batching and multiple training iterations, clipping the policy update to limit large changes, and using Generalized Advantage Estimation (GAE) to calculate advantages and returns. These represent the main differences between REINFORCE and PPO as implemented here.

The flow of the training process is as follows:

- **F1 - Prepare Batches:** The memory samples collected are extracted into separate arrays, which are converted into tensors compatible with the neural network.
- **F2 - Get Value Estimates:** Value estimates for the states are obtained from the neural network using the stored states. An additional final state, representing the next state of the last transition, is added to allow calculation of returns using GAE.
- **F3 - Compute Advantages and Returns:** Advantages and returns are computed using the GAE formula, incorporating reward-modified values and done flags.
- **F4 - Create Indices for Sampling:** A set of indices equal to the memory size is generated to allow shuffling of memory during training.
- **F5 - Get Old Action Logits:** The old action logits are obtained from the neural network using the stored states.
- **F6 - Convert Logits to Probabilities:** The logits from F5 are converted into probabilities using a categorical distribution.

- **F7 - Get Log Probabilities of Actions:** The log probabilities of the actions are calculated using the policy gradient theorem, similar to REINFORCE.
- **F8 - Reset Entropy and Loss:** Entropy and loss variables are reset to zero, as multiple training iterations will occur and cumulative loss and entropy will be tracked.
- **F9 - Iterate Over Training Steps:** For each training iteration, the following steps are executed.
- **F10 - Shuffle Indices:** The indices from F4 are shuffled to randomize the order of training data.
- **F11 - Iterate Over Mini-batches:** For each mini-batch, the following steps are executed:
- **F12 - Get Mini-batch Indices:** The indices for the current mini-batch are selected.
- **F13 - Extract Mini-batch Data:** The relevant data from memory (F1) is extracted for the mini-batch.
- **F14 - Get Current Action Logits:** Current action logits for the mini-batch states are obtained from the neural network. For the first mini-batch in the first iteration, these exactly match the old action logits. This serves as a check for implementation correctness.
- **F15 - Convert Current Logits to Probabilities:** The logits from F14 are converted into probabilities using a categorical distribution.
- **F16 - Get Log Action Probabilities:** The probabilities are converted to log probabilities for policy gradient calculations.
- **F17 - Compute Ratios:** The ratio of new to old log probabilities is calculated to limit excessive policy updates, a key component of PPO.
- **F18 - Normalize Advantages:** The mini-batch advantages are normalized.
- **F19 - Compute Clipped Loss:** Two losses are calculated—one standard and one clipped. The smaller of the two is used to update the policy, ensuring updates remain close to the original policy.
- **F20 - Get Value Estimates:** Value estimates for the mini-batch states are obtained from the neural network.
- **F21 - Compute Value Loss:** Value loss is calculated as the mean squared error between the network predictions and mini-batch returns.
- **F22 - Compute Entropy:** The entropy of the action probabilities is calculated.

- **F23 - Compute Total Loss:** The total loss is computed as the sum of the clipped policy loss, value loss (weighted by a value coefficient), and entropy loss (weighted by an entropy coefficient).
- **F24 - Backpropagation:** Backpropagation is performed and the optimizer updates the network parameters.
- **F25 - Accumulate Loss:** Loss is accumulated for tracking.
- **F26 - Accumulate Entropy:** Entropy is accumulated for tracking throughout training.

In this section, the implementation of the PPO agent is discussed. As evident, the agent's code is considerably longer than that of other agents, reflecting the original complexity of TRPO, which PPO aims to improve upon in terms of efficiency and stability. The highlights presented demonstrate how PPO uses the policy gradient theorem to update its policy and how it shares conceptual similarities with REINFORCE, as both algorithms belong to the same family of policy-gradient methods.

6.3 Conclusion

This chapter began with an overview of the code design philosophy and explained how the flow diagrams presented provided a direct mapping to the actual code implementations. Subsequently, the three most commonly used functions in this research were discussed, illustrating how each agent will be trained and evaluated, and how its actions will interact with the environment in accordance with the Markov Decision Process (MDP). Each of the four agents was individually described, beginning with a discussion of their attributes and any additional parameters they utilized, followed by an examination of the core functions necessary for their operation.

This discussion demonstrated an understanding of the mathematical concepts introduced in Section 2 and showed how the agent documentation and theoretical foundations from Section 3 were applied. Together, these elements formed the practical implementation of the agents presented in Chapter 6, providing a deeper and more functional understanding of their operation. The following sections illuminated the internal mechanisms of the agents, integrating investigative tools discussed in Chapter 5 with the environments presented in Chapter 4, thereby consolidating the foundational research into a coherent framework.

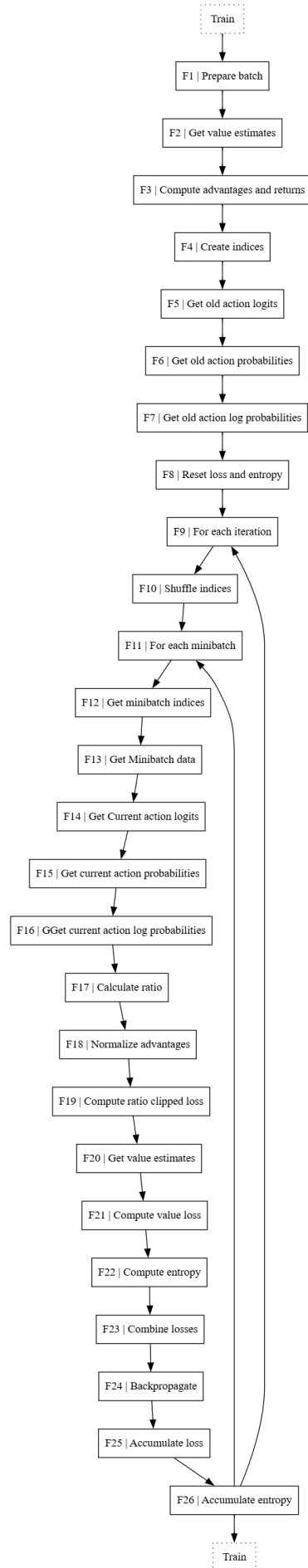


Figure 6.20: Training flow for the PPO agent.

Chapter 7

Experimental design and execution

In this section, the experiments conducted with the four agents—Q-Learning, DDQN, REINFORCE, and PPO—are discussed and executed within the environments outlined in Chapter 4. The analytical techniques described in Chapter 5 are applied to highlight specific characteristics and behaviors of each agent. These behaviors may arise from implementation details, internal structures, or patterns learned during training. The overall objective is to gain deeper insight into agent functionality by performing experiments based on the fundamental concepts outlined in Chapters 2 and 3, thereby contributing to the main research aims stated in Chapter 1.

The section begins with a foundational experiment, referred to as the *Implementation Validation Experiment*, designed to confirm that all agents have been implemented correctly. This is followed by the *Policy Visualization Experiment*, which utilizes policy visualization analysis to highlight notable behavioral features of the agents. Next, a *Learning Rate Impact Experiment* is conducted to examine differences in the agents’ sensitivity to changes in learning rate. Thereafter, an *Exploration Consistency Experiment* explores how policy-gradient-based algorithms approach exploration. Subsequently, a *State Visitation Experiment* demonstrates how the State Visitation Analysis and Feature Heatmap Analysis techniques can be combined to visualize policy behavior in a more complex environment.

The section concludes with an open-ended experiment titled the *Feature Selection Experiment*. In this experiment, the Decision Boundary Analysis method is applied to demonstrate how a policy can be visualized and potentially refined, enabling further evaluation or even partial policy reconstruction based solely on observed decision boundaries.

It is important to note that the goal of these experiments is to outline and interpret agent behaviors, characteristics, and properties, rather than to produce optimally performing agents. In many cases, only the final filtered reward is considered sufficient to confirm

that an agent has learned an acceptable level of behavior for demonstration purposes.

7.1 Implementation validation

This section presents the validation of the implemented agents described in Chapter 6. The goal is to determine whether the agents are implemented sufficiently to learn and outperform a random agent.

7.1.1 Experimental question

Is the implementation of the agents in Chapter 6 sufficient to allow them to learn and outperform a random agent?

7.1.2 Hypothesis

Given that the agents are based on existing implementations, the mathematical formalism presented in [12], and the results of initial tests, it is hypothesized that the agents are implemented adequately to demonstrate learning and to outperform a random agent.

7.1.3 Methodology

For this experiment, all four agents are evaluated in the FrozenLake environment. A deterministic version of the environment is used for simplicity instead of the slippery variant, the reason for this is that the slippery nature of the environment introduces significant variance, and in environments with sparse rewards, such as Frozen Lake, this merely increases the training time unnecessarily. Each agent interacts with the environment, and its results are plotted in Figure 7.2. The environment is wrapped in a utility function that converts the state representation from a single number between 0 and 15 into a one-hot encoded vector of length 16. This allows neural network-based agents to interpret the state more effectively. The overall execution flow is illustrated in Figure 7.1.

The methodology proceeds as follows:

- **F1 - Random Agent:** A masked random agent is created to ensure that only legal moves are executed (i.e., the agent cannot move into walls but may fall into holes).
- **F2 - Environment Creation:** The FrozenLake environment is initialized, and the state is one-hot encoded. The deterministic version of the environment is used for consistency.

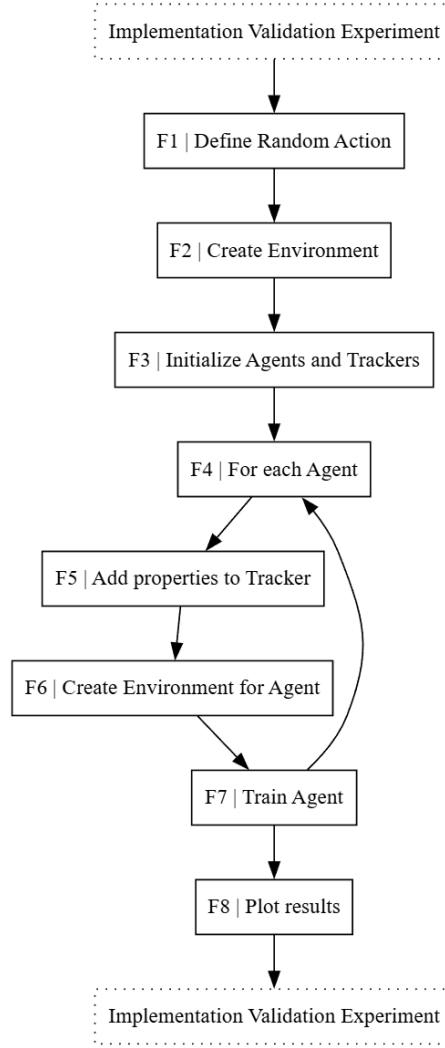


Figure 7.1: Flow diagram of the implementation validation experiment.

- **F3 - Agent Initialization:** All agents are instantiated, and a tracker is assigned to each. Hyperparameters are loaded from a predefined settings dictionary optimized for performance.
- **F4 - Agent Execution:** Each agent is executed in the environment according to the experimental loop.
- **F5 - Reward Tracking:** A filtered version of the reward is recorded in the tracker.
- **F6 - Environment Reset:** A fresh copy of the environment is created for each agent to ensure a clean training session.
- **F7 - Agent Training:** Each agent is trained for 2000 episodes with early stopping if the filtered reward reaches 1.0.
- **F8 - Results Visualization:** After all agents complete training, the tracked rewards are extracted and plotted.

7.1.4 Results

Figure 7.2 presents the filtered¹ rewards obtained by the agents during training. Q-learning, DDQN, and PPO achieve the optimal solution, resulting in early stopping of training. The REINFORCE agent demonstrates partial learning, reaching a maximum reward of approximately 0.25, this is further discussed in section 7.7. The random agent fails to reach a reward level of 0.1.

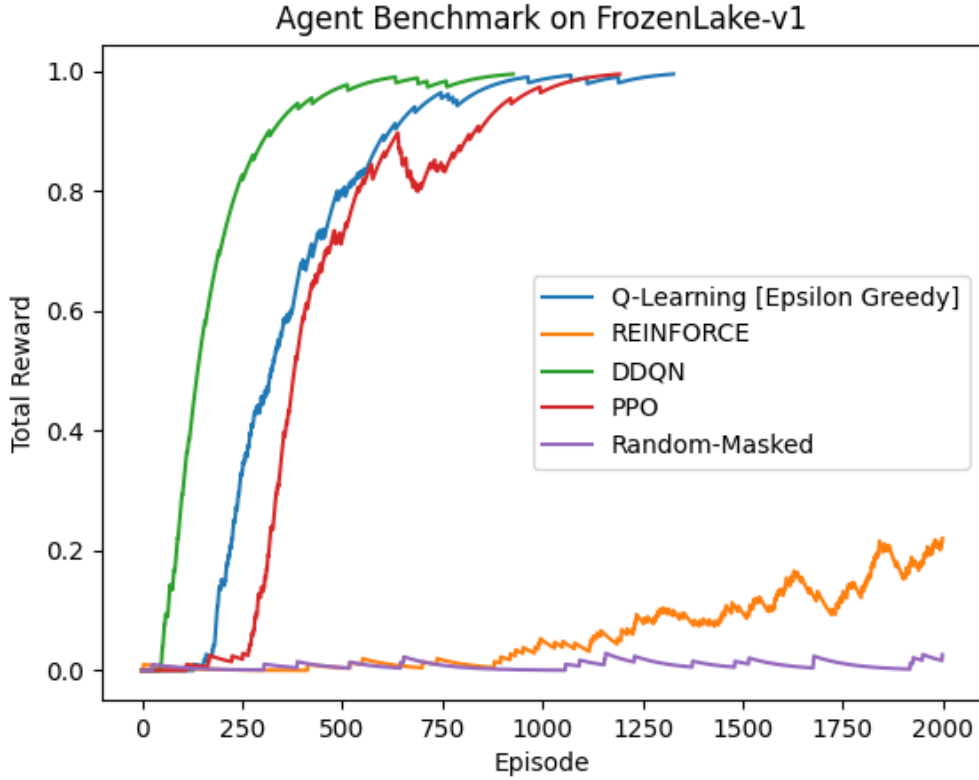


Figure 7.2: Filtered reward curves for all agents in the FrozenLake environment.

7.1.5 Conclusion

The results indicate that the implementations of PPO, DDQN, and Q-learning enable the agents to learn effectively and solve the environment, easily outperforming a random agent. The REINFORCE agent does not fully solve the environment but demonstrates learning and still outperforms the random agent. The limited performance of REINFORCE is attributed to its reliance on complete rollouts and sparse reward signals; the environment only provides a reward upon reaching the goal, resulting in insufficient learning samples.

¹Due to the stochastic nature of the agents' training and the plotting of multiple agents on the same axis, in many cases the results were filtered to smooth out noise and clarify the differences between each reward curve. The filter used is a simple exponential moving average (EMA) filter, defined as $x_t = (1 - \alpha)x_t + \alpha x_{t-1}$ with $\alpha = 0.99$.

In contrast, PPO benefits from multiple iterations and mini-batches, allowing it to leverage the same samples more effectively and achieve superior learning performance.

This experiment highlights how an agent’s performance can be tracked over its training cycles and demonstrates that the maximization of the reward, as described in Chapter 2, is practically occurring. The results are plotted and analyzed, providing insights into the agent’s performance and enabling assessment of whether the agent is improving over time.

The findings confirm that the hypothesis is correct: the agents are implemented effectively, allowing them to train, learn, and consequently outperform a random agent.

7.2 Policy visualization

In this section, the implementation validation experiment is extended by incorporating policy visualizations to analyze how each agent’s policy evolves throughout training. The goal of this experiment is to observe and compare how different agents adjust their decision-making strategies over time, providing insights into the learning dynamics of value-based and policy-based methods.

7.2.1 Experimental question

Is there a relationship between the underlying optimization type (i.e., value-based or policy-based) and the manner in which an agent’s policy changes throughout the training process?

7.2.2 Hypothesis

Yes. Given that policy-based optimization methods directly adjust the policy parameters gradually over time, while value-based methods compute state-action values and select the action with the highest estimated value, it is expected that value-based agents will exhibit more abrupt policy changes whenever a new action is assigned a slightly higher Q -value. In contrast, policy-based agents are anticipated to display smoother and more gradual policy updates during training.

7.2.3 Methodology

In this experiment, the same Random agent with an action mask was implemented as in the Implementation Validation experiment. Additionally, all four agents were used. Each agent was trained in increments of 500 episodes, for a total of 2000 episodes. Before starting, and after each set of 500 episodes, the agent’s current policy was visualized using

the Policy Visualization Analyzer.

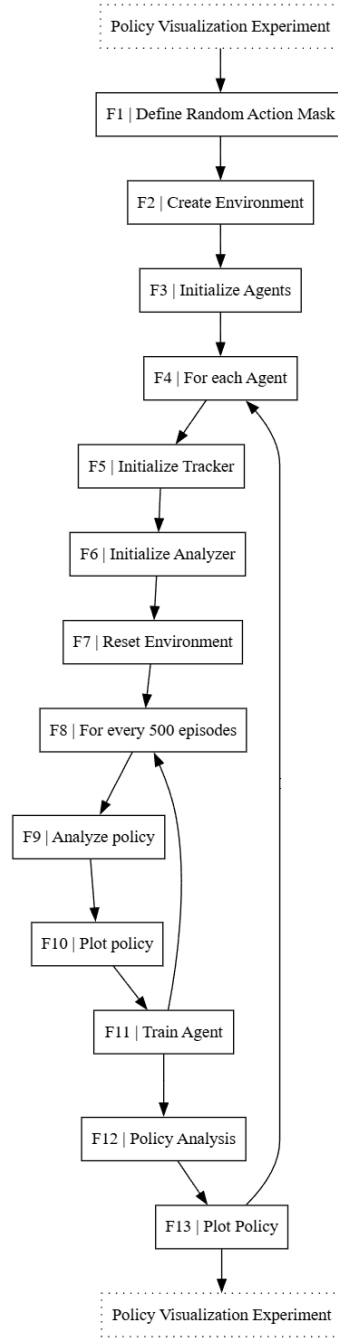


Figure 7.3: Flow of the experimental procedure for policy visualization and agent training.

Figure 7.3 illustrates the flow of the experiment:

- **F1 - Define random action mask:** The action mask function for the random agents was defined to ensure that the agents avoid walls.
- **F2 - Create environment:** The experimental environment was created.
- **F3 - Initialize actions:** The agents were initialized, and their initial states were stored in a dictionary.

- **F4 - For each agent:** The following loop was executed for each agent.
- **F5 - Initialize tracker:** A new tracker was initialized for performance monitoring.
- **F6 - Initialize analyzer:** The policy visualization analyzer was initialized.
- **F7 - Reset environment:** The environment was reset before training.
- **F8 - Train in increments:** The agent was trained for 500 episodes at a time, up to a total of 2000 episodes.
- **F9 - Analyze policy:** After each 500-episode increment, the agent’s updated policy was analyzed.
- **F10 - Plot policy:** The updated policy was plotted.
- **F11 - Train agent:** The agent continued training for the next 500 episodes.
- **F12 - Final policy analysis:** After completing all 2000 episodes, the final policy was analyzed.
- **F13 - Plot final policy:** The final policy was plotted and stored for further reference.

7.2.4 Results

Due to the structure of the experiment, a total of 25 policy visualization images are generated. However, it is neither practical nor necessary to display all of them. In several cases, two figures present identical or nearly identical results, and therefore, only one representative figure is included. Additionally, only figures that provide notable or meaningful insights are presented below. To begin, the policy of the random agent is examined for confirmation and visual comparison. The policy of the random agent is shown in Figure 7.4.

From Figure 7.4, it is evident that the agent primarily avoids the edges of the environment and otherwise selects actions randomly. Next, the initial policies of the Q-Learning and DDQN agents are examined before any training occurs. Since both employ the epsilon-greedy action selection strategy and begin with an exploration rate (ϵ) of 1.0, their initial policies are identical, as illustrated in Figure 7.5.

As shown in Figure 7.5, the epsilon-greedy policy results in uniformly random action selection across all states, with each action having an equal probability of 25%. The initial policies of the REINFORCE and PPO agents are then examined. Although minor differences exist, their overall policy structures are similar. The REINFORCE policy is shown in Figure 7.6.

As illustrated in Figure 7.6, both REINFORCE and PPO begin with approximately uniform action probabilities across all directions, though not exactly 25% per action. This variation is due to the random initialization of neural network weights.



Figure 7.4: Policy visualization of the Random (masked) agent before training.



Figure 7.5: Initial policy visualization of Q-Learning and DDQN agents before training ($\epsilon = 1.0$).

Mid-Training Policy Analysis

The policy of the Q-Learning agent after 500 training episodes, with an exploration rate of approximately 0.8, is shown in Figure 7.7.

As shown in Figure 7.7, the Q-Learning agent selects the highest Q-value action approx-



Figure 7.6: Initial policy visualization of the REINFORCE agent before training. PPO exhibits a similar initial distribution.



Figure 7.7: Q-Learning agent policy after 500 training episodes ($\epsilon \approx 0.8$).

imately 44% of the time and a random action 56% of the time. This high randomness suggests that the agent has largely learned the optimal policy but continues to explore to confirm its strategy.

The DDQN agent's policy after 500 episodes, with an exploration rate of approximately 0.01, is presented in Figure 7.8.



Figure 7.8: DDQN agent policy after 500 training episodes ($\epsilon \approx 0.01$).

As seen in Figure 7.8, the DDQN agent selects the optimal Q-value action approximately 99% of the time. This rapid convergence is due to its higher exploration decay rate. Similar to Q-Learning, the DDQN agent has effectively solved the environment by this stage.

However, it is worth noting that DDQN seemingly did not learn to avoid the walls, particularly those on the right side. This is due to the environment structure, which does not penalize the agent for attempting to move into a wall. Additionally, these states are rarely visited during the agent’s attempts to solve the environment, so the corresponding Q-values are not meaningfully updated. It is also possible that these Q-values are in fact very close to each other, and movement to the right is selected simply by a small margin.

In contrast, the REINFORCE agent’s policy after 500 episodes, shown in Figure 7.9, indicates that it has not yet stabilized on a preferred trajectory. The agent shows confidence primarily in the second-to-last state, where it selects the correct action with approximately 75% probability.

A similar pattern is observed for the PPO agent after 500 episodes (Figure 7.10). The PPO agent shows fewer tendencies to move into walls and begins to orient toward the goal in later states, though it has not yet fully converged.

This slower adaptation in PPO and REINFORCE arises from the fact that these policy-gradient methods incorporate exploration inherently within the policy network, rather than through an external exploration parameter.

Final Policy Analysis

The Q-Learning agent’s final policy, at 2000 episodes and an exploration rate of approximately 0.24, is shown in Figure 7.11.



Figure 7.9: REINFORCE agent policy after 500 training episodes.



Figure 7.10: PPO agent policy after 500 training episodes.

As seen in Figure 7.11, the Q-Learning agent maintains the same general policy structure but with reduced exploration. The core behavior remains stable.

The DDQN agent exhibits an unexpected change in its final policy (Figure 7.12). The agent begins to select suboptimal actions in the initial states, temporarily reducing overall efficiency.



Figure 7.11: Final policy of the Q-Learning agent after 2000 episodes ($\epsilon \approx 0.24$).

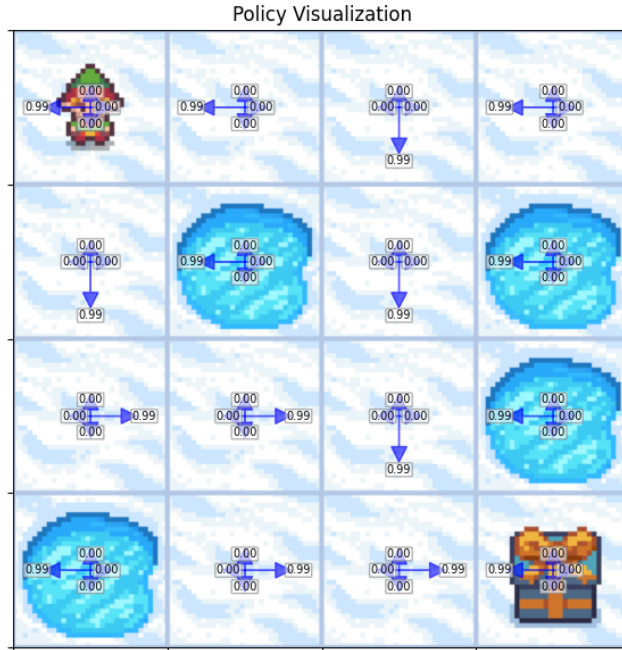


Figure 7.12: Final policy of the DDQN agent after 2000 episodes.

This phenomenon occurs because both Q-Learning and DDQN rely on the maximum Q-value to determine optimal actions. When multiple Q-values are nearly identical, even a minimal numerical difference (e.g., 10^{-8}) can cause sudden policy shifts. This sensitivity occasionally leads to instability and rapid policy changes.

In contrast, the REINFORCE agent's final policy (Figure 7.13) demonstrates improvement primarily in the later states, though it still struggles with early-stage decision-making.

Finally, the PPO agent's policy after 2000 episodes of training (Figure 7.14) shows a gradual and consistent convergence toward the optimal solution. As evidence shows, from episodes 0 to 500 and 500 to 2000, small changes occur over time, in contrast with DDQN, which makes large changes between episodes 0 and 500 and then only a few large changes here and there between episodes 500 and 2000.

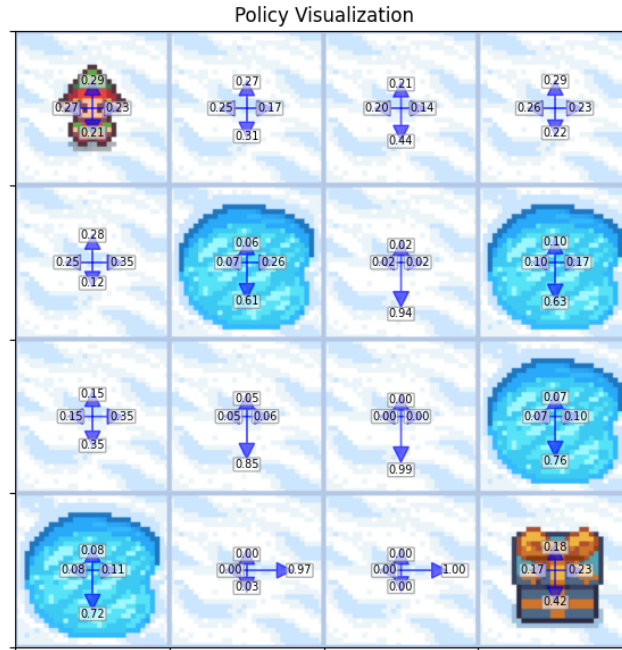


Figure 7.13: Final policy of the REINFORCE agent after 2000 episodes.

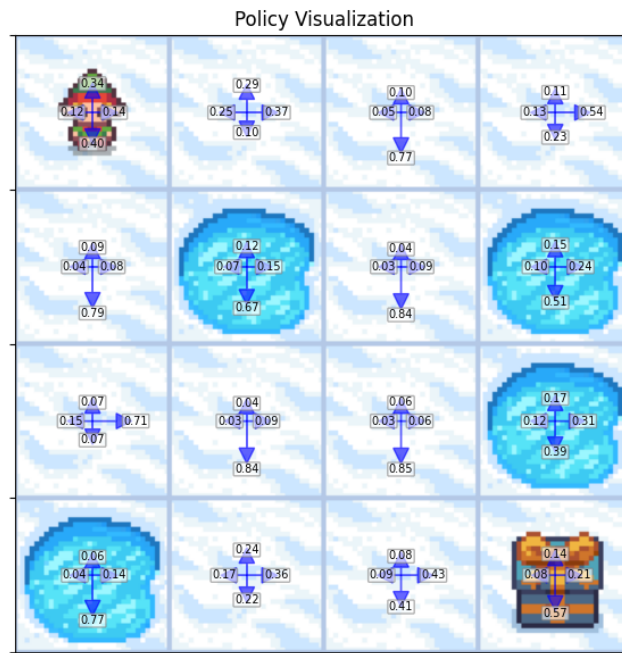


Figure 7.14: Final policy of the PPO agent after 2000 training episodes.

7.2.5 Conclusion

In conclusion, the results show that both DDQN and Q-Learning agents successfully solve the environment but exhibit some instability due to the sensitivity of Q-values, which can lead to rapid policy fluctuations. In contrast, the REINFORCE and PPO agents adjust their policies more gradually and do not exhibit sudden changes.

This experiment highlights how the policy visual analysis helps showcase the difference between policy optimization vs value optimization algorithms' internal differences and highlights them in the different ways they update policies. For Q-learning and DDQN, the policy changes dramatically even with very low levels of epsilon since epsilon is decoupled from the update, unlike PPO and REINFORCE which always make smooth transitions in a relative comparison.

Therefore, the hypothesis is confirmed: the agents are implemented sufficiently, demonstrating the ability to train, learn, and outperform a random agent. Furthermore, the results highlight a distinct difference in how policy optimization methods (PPO and REINFORCE) evolve their strategies compared to value-based optimization methods (Q-Learning and DDQN), enhancing the agent's intelligibility. .

7.3 Learning rate impact

In this section, the learning rate impact experiment is conducted. In this experiment, three variations of both the DDQN and PPO agents are examined: one using the default learning rate, one using $0.1 \times$ the default rate, and one using $10 \times$ the default rate. All other parameters are kept constant. The goal of this experiment is to determine the influence that changes in the learning rate have on each of these agents. The **Snake** environment is used for this experiment since, due to its partially observable nature, learning too quickly may lead to premature convergence.

7.3.1 Experimental question

Is there a difference between DDQN and PPO in terms of the impact that the learning rate has on their performance and training stability?

7.3.2 Hypothesis

Yes. Since the learning rate in PPO determines how rapidly the policy is updated, and since the policy itself inherently controls exploration, any change in the learning rate directly affects both policy adaptation and exploration. In contrast, for DDQN, exploration is decoupled from the learning rate because it is governed separately by the ϵ -greedy

mechanism. Therefore, a distinct difference in the sensitivity to learning rate changes is expected between the two methods.

7.3.3 Methodology

The methodology used to conduct this experiment is illustrated in Figure 7.15.

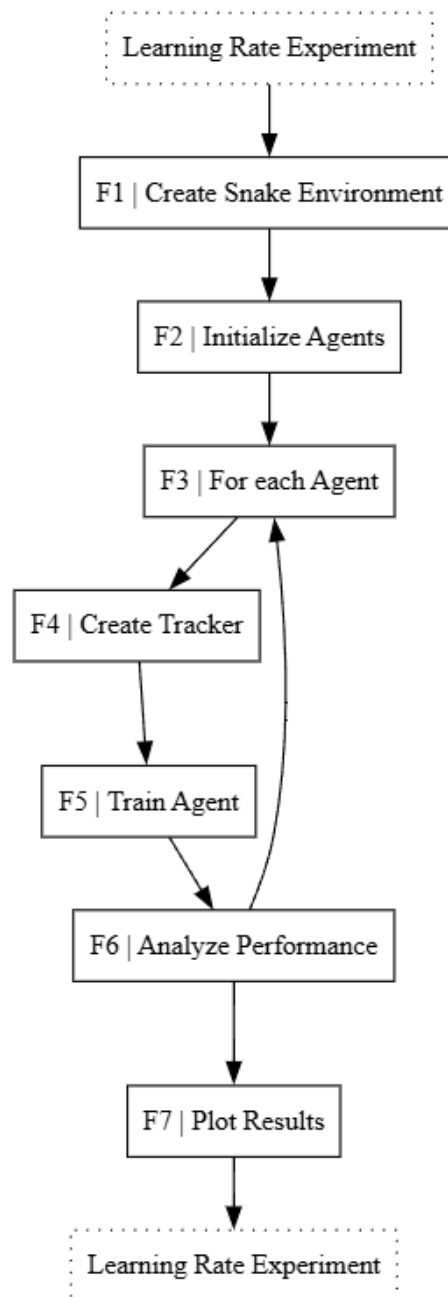


Figure 7.15: Flow diagram of the learning rate impact experiment.

- **F1 – Create Environment:** The Snake environment is initialized.
- **F2 – Initialize Agents:** Six agents are created — three DDQN and three PPO — each configured with identical parameters except for the learning rate variation.

Each agent is also assigned a performance analyzer.

- **F3 – For Each Agent:** The following loop is executed for each agent.
- **F4 – Create Tracker:** A tracker is created and connected to the analyzer.
- **F5 – Train Agent:** Each agent is trained once for 1000 episodes in the **Snake** environment.
- **F6 – Perform Analysis:** The performance analyzer evaluates the results after training.
- **F7 – Plot Results:** Once all agents have completed training, the results are compiled and plotted for comparison.

7.3.4 Results

The results of the experiment are shown in Figure 7.16. Of the six agents tested, only one failed to show meaningful improvement during training. Before analysing the results, some additional information regarding the figures is needed. First, the top-left plot (the reward curve) is again a filtered version of the agent’s rewards, using the EMA filter as described earlier. The mean reward curve (bottom-left) shows the cumulative mean reward. Therefore, for episode 1, the mean reward curve and the reward curve have the same value. For episode 2, the mean reward curve represents the average reward of episodes 1 and 2. For episode 10, the mean reward curve represents the average over episodes 1 to 10, and so on. The same logic applies to the standard deviation curve (bottom-right). For the first episode, there is only one reward, so the standard deviation is 0. For the tenth episode, there are ten rewards, and the standard deviation of those rewards is recorded.

Naturally, the mean reward curve increases rapidly at first and then increases more slowly as the number of samples grows. The standard deviation curve, on the other hand, starts at a small value and increases up to a certain point. This occurs because, early in training, the agent mainly receives a reward of 0, with little uncertainty, and as the reward distribution expands, the standard deviation increases accordingly. After some point—in this case, around episode 1000—the standard deviation begins to plateau as it reaches a maximum. With further training it will eventually decrease again.

There is also a noticeable spike in four of the agent’s loss curves (top-right). This spike is caused by the agent having collected enough samples and beginning to perform better; improved performance often leads to larger temporal-difference errors, producing a spike in loss before it gradually decreases again over time, similar to the behaviour observed in the standard deviation curve.

From Figure 7.16, it is evident that the change in learning rate has only a minor effect on the standard deviation of the agents’ performances. All agents converge to a similar range

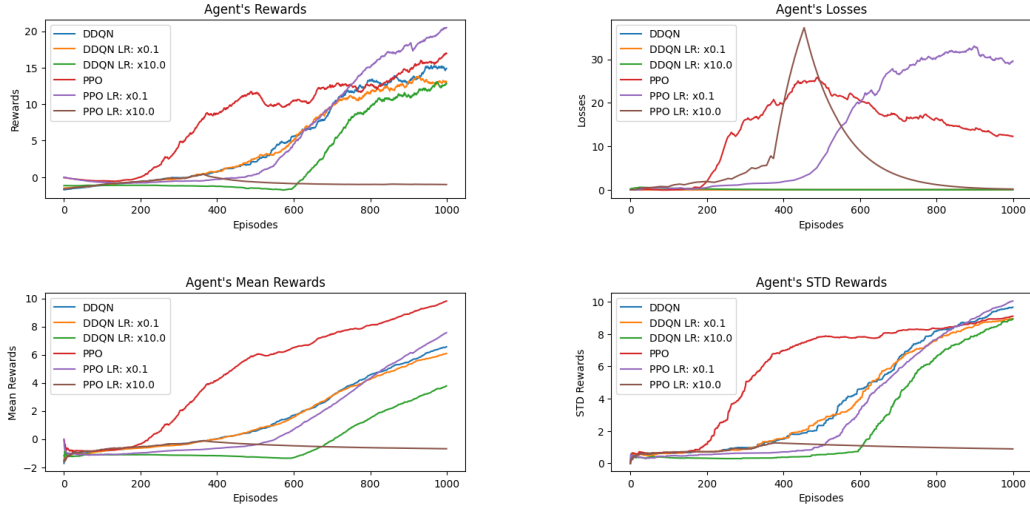


Figure 7.16: Performance comparison of DDQN and PPO agents under different learning rate settings: $0.1\times$, $1.0\times$, and $10\times$ the default rate.

of performance values. However, a clear difference is observed in the mean performance levels.

The mean reward curve in the bottom-left plot shows that the PPO agent with the default learning rate performs best, while the PPO agent with the $10\times$ learning rate performs worst, demonstrating a significant sensitivity to rate magnitude. In contrast, all DDQN agents perform similarly and cluster near the middle range of performance.

Examining the reward curves, it is observed that all agents follow comparable general trajectories. The PPO agent with the $10\times$ learning rate (red curve) learns faster initially but slows down significantly as training progresses, suggesting premature convergence. The PPO agent with the $0.1\times$ learning rate follows a similar trajectory but continues to improve steadily while others plateau. The DDQN agent with the $10\times$ learning rate (green curve) initially learns more slowly but ultimately reaches a performance level comparable to the others, highlighting DDQN's robustness to learning rate variation.

In summary, PPO demonstrates high sensitivity to learning rate changes, with smaller learning rates producing more stable learning and larger ones leading to instability or stagnation. DDQN, by contrast, remains comparatively stable and less affected by the same variations. This matches the theoretical foundations of these agents directly and demonstrates how the analysis technique was used to highlight this internal aspect of the agent.

7.3.5 Conclusion

In conclusion, the results clearly indicate a significant difference between PPO and DDQN in how they respond to changes in learning rate. These findings support the theoretical

expectation that PPO benefits from smaller learning rates and gradual policy updates, as its performance is tightly linked to policy adjustment stability. DDQN, on the other hand, exhibits greater robustness and tolerates a wider range of learning rates, consistent with its value-based optimization design. This allows for easier selection of the learning rate when choosing one for an agent, demonstrating how theory aligns with practical implementation. Therefore, the hypothesis is confirmed.

Additionally, this experiment showcased how an agent’s properties can be inspected, gaining insights into how changing a property of an agent helps understand its internal workings and performance during training, coupling that agent’s hyperparameters to the internal mechanisms used in Chapters 2, 3, and 6 to create the agents, and therefore extending the understanding and interpretation of the agent, enhancing the agent’s explainability.

7.4 Exploration consistency

In this section, the exploration consistency experiment is conducted. In this experiment, different variations of the REINFORCE and PPO agents are tested using three entropy coefficient values: 0, 0.01, and 0.1. While value-based agents can alter their exploration strategy through different action-selection policies (such as ϵ -greedy), policy optimization agents have more limited control over exploration. For these agents, exploration is primarily influenced by the entropy factor, since adjusting the learning rate also affects other aspects of training.

The goal of this experiment is therefore to examine the effect of varying entropy factors on the consistency of policy optimization algorithms. The agents are trained in the `CartPole-v1` environment, as it provides a simple and standardized benchmark where a reward of 500 is considered a solved threshold.

7.4.1 Experimental question

How does altering the entropy factor of a policy optimization agent affect its ability to consistently train from an untrained state to a high-performing policy?

7.4.2 Hypothesis

Changing the entropy factor is expected to influence the level of exploration. Increasing the entropy factor should enhance exploration, allowing the agent to more consistently reach similar states. However, this increase in exploration is also expected to reduce the average performance, since more random actions prevent the agent from fully exploiting learned knowledge. This effect is conceptually similar to ϵ -greedy action selection in value-based systems.

7.4.3 Methodology

Figure 7.17 illustrates the flow of the experiment.

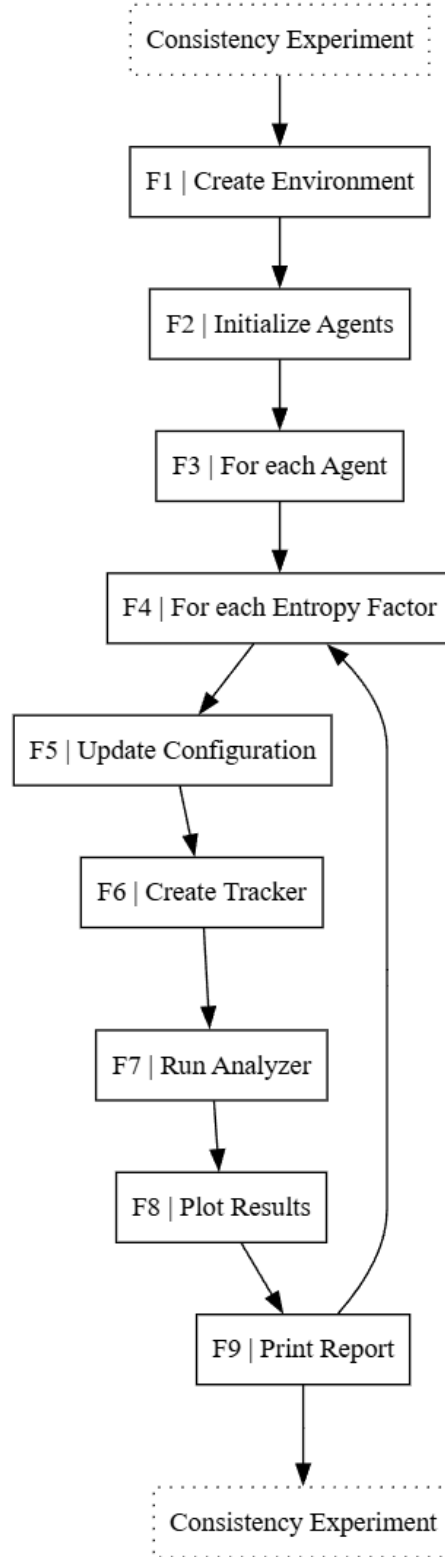


Figure 7.17: Flow diagram of the exploration consistency experiment.

- **F1 – Create environment:** The `CartPole-v1` environment is initialized.

- **F2 – Initialize agents:** Both REINFORCE and PPO agents are created with a copy of the default configuration.
- **F3 – Iterate over agents:** The following loop is executed for each agent.
- **F4 – Iterate over entropy factors:** For each of the three entropy factors (0, 0.01, and 0.1), the consistency analysis is repeated.
- **F5 – Update configuration:** The agent configuration is updated to reflect the new entropy factor.
- **F6 – Create tracker:** A new performance tracker is created for each iteration.
- **F7 – Train and average results:** The agent is trained for 200 episodes, repeated 200 times, and the average reward curve is computed.
- **F8 – Plot results:** The average results and standard deviation (STD) are plotted for comparison.
- **F9 – Generate report:** The experiment concludes with a summary and statistical analysis.

7.4.4 Results

Figures 7.18 and 7.19 present the results of the REINFORCE and PPO agents, respectively.

In Figure 7.18, it is evident that the REINFORCE agent struggles in this environment and shows low overall consistency. During the early stages of training, the 0 and 0.1 entropy factors exhibit high standard deviations, while the 0.01 factor produces the smallest deviation. As training progresses, the standard deviation for 0.1 decreases, ultimately becoming the smallest among the three, whereas the 0 factor shows the largest deviation. In terms of mean reward, a higher entropy factor correlates with a lower average performance.

In Figure 7.19, the PPO agent results demonstrate that a higher entropy factor significantly reduces mean performance—by nearly 40%. The corresponding increase in standard deviation indicates lower consistency. However, between episodes 50 and 80, the 0.01 entropy factor shows the smallest deviation, suggesting a temporary balance between exploration and stability.

Since the agent in many cases completely solves the environment and reaches the peak reward of 500, the standard deviation curve appears to hover around values based on the first standard deviation, occasionally producing results that extend above 500. In practice, this does not reflect the true behaviour of the agent. Therefore, these results slightly skew the perceived performance of the PPO agent. However, the general shape and overall meaning of the curve remain valid.

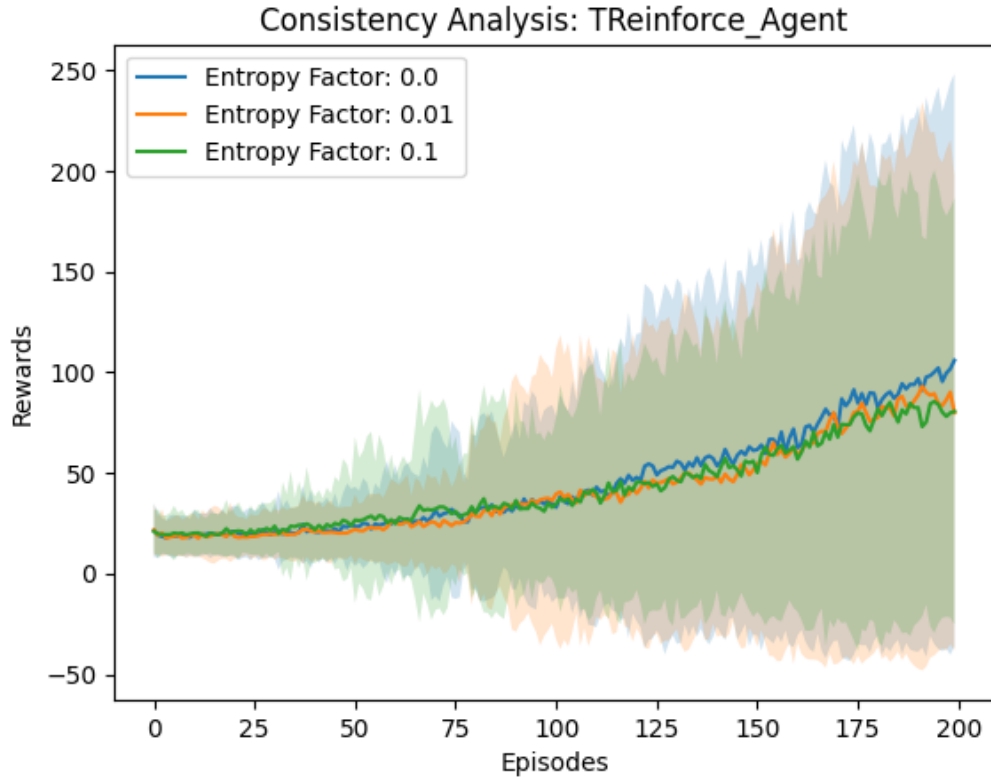


Figure 7.18: Exploration consistency of the REINFORCE agent for different entropy factors.

7.4.5 Conclusion

From these results, it is apparent that a large entropy factor decreases the agents' average reward. For REINFORCE, it improves consistency, while for PPO, it tends to reduce consistency, especially in later stages of training. This difference is expected, since PPO performs multiple optimization steps on sampled data, meaning that excessive exploratory actions can reduce performance by training on less optimal actions.

Overall, the findings support the notion that a small entropy factor can promote sufficient exploration without compromising stability. However, since the influence of entropy diminishes over time, an adaptive entropy schedule that decays as training progresses may provide optimal results. Therefore, the original hypothesis is only partially supported.

This experiment highlights how the property of entropy, for REINFORCE and PPO, was investigated using the analysis techniques. Combining the theory of Chapter 2 once again with the implementation of Chapter 6 highlights how an agent's properties can be investigated and the influence on the agent determined. This method can therefore be extended onto other properties or even new proposed properties to gain insights on the agent's behavior with that property, along with how that property affects the mechanisms internal to that agent enhancing the agent's explainability.

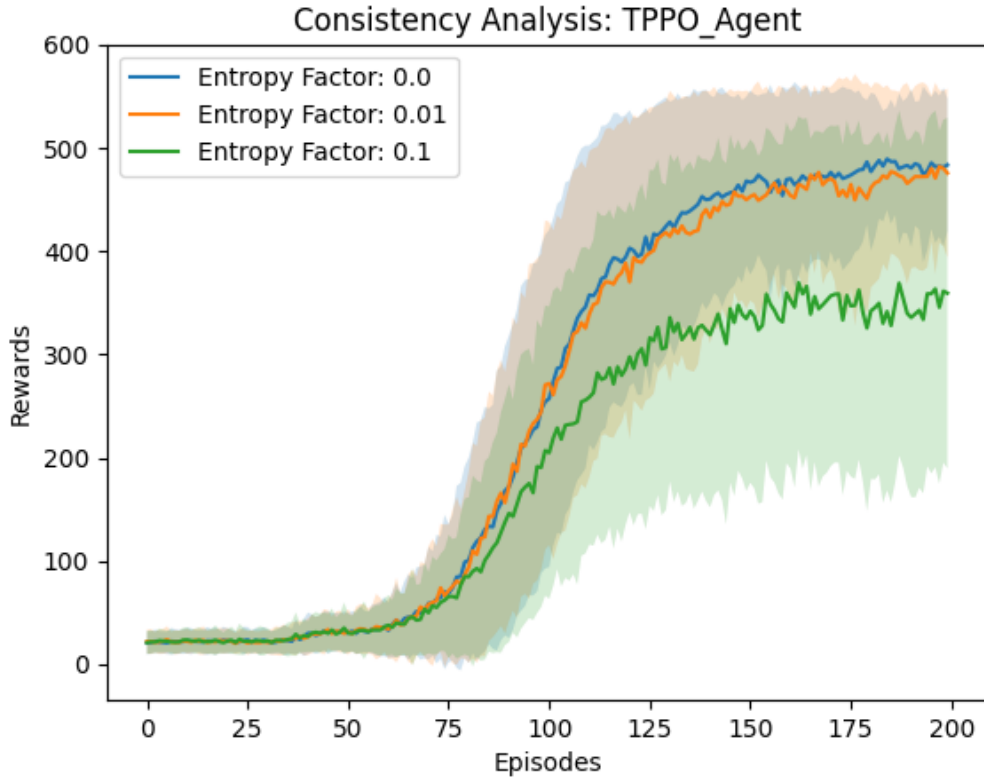


Figure 7.19: Exploration consistency of the PPO agent for different entropy factors.

7.5 State visitation analysis

In this section the state visitation experiment is conducted. In this experiment only the Q-learning agent is used and trained in the **Taxi-v3** environment. The goal here is to gain insight into the agent's policy or strategy in a more complex environment when direct policy visualization is not possible.

7.5.1 Experimental question

How can an agent's policy be visualized when direct policy visualization is not an option?

7.5.2 Hypothesis

When direct visualization of an agent's policy is not possible, the policy can still be inferred through indirect analysis methods. One common approach is state visitation analysis, which shows how frequently the agent visits different states of the state space. Although this does not reveal the full policy, it highlights the agent's behavioural tendencies and preferred trajectories.

To gain a more complete understanding, state visitation can be combined with additional techniques such as feature activation heatmaps. Together, these methods provide a meaningful approximation of the agent’s policy by revealing how states, features, and actions relate to one another during decision-making. Together, these indirect analysis methods make it possible to visualize an agent’s policy even when direct policy visualization is not available.

7.5.3 Methodology

In figure 7.20 the method used to execute the experiment can be seen.

- **F1 - State Decoder:** In order to extract meaningful features, the state must be decoded from a single integer to a taxi row, taxi column, passenger destination, and passenger current location. This step involves writing a decoding function.
- **F2 - Create Environment:** The Taxi-v3 environment is created.
- **F3 - Create Agent:** The Q-learning agent is created using its default settings.
- **F4 - Create Tracker:** A tracker is initialized for logging training metrics.
- **F5 - Create State Visitation Analyzer:** A state visitation analyzer is initialized.
- **F6 - Train Agent:** The agent is trained in the Taxi environment for 400 episodes.
- **F7 - State Analysis:** The visited states are analyzed.
- **F8 - Decode States:** The states are decoded using the function from F1.
- **F9 - Create Tracker:** A new tracker is created for the feature heatmap analysis.
- **F10 - Create Feature Heatmap Analyzer:** A feature heatmap analyzer is created with the bin size set to 5, matching the 5x5 grid of the environment.
- **F11 - Log States:** The decoded states are logged into the tracker for feature heatmap generation.
- **F12 - Analyze Feature Heatmaps:** The feature heatmaps are generated.
- **F13 - Plot Heatmaps:** The heatmaps are plotted to visualize the training phase.
- **F14 - Reset Environment:** The environment is reset for evaluation.
- **F15 - Set Agent Evaluation Mode:** The agent is set to evaluation mode.
- **F16 - Create Tracker:** A tracker is created for evaluation.
- **F17 - Create State Visitation Analyzer:** A new state visitation analyzer is initialized for evaluation.

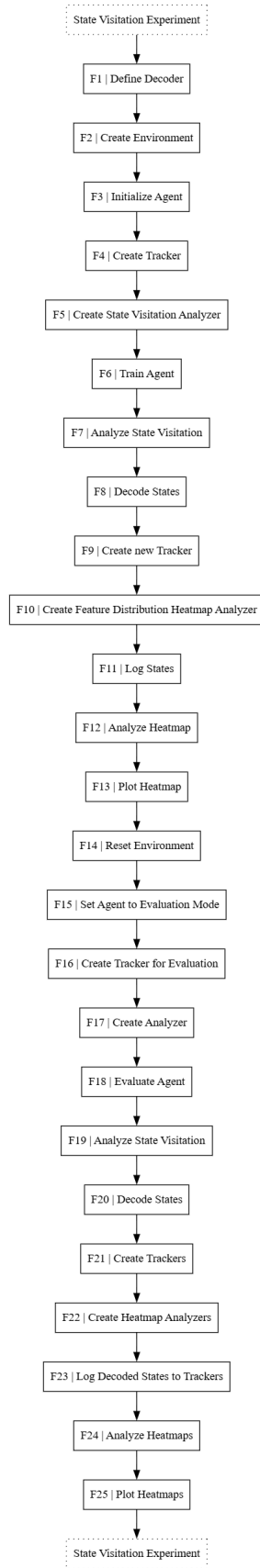


Figure 7.20: Flow diagram illustrating the state visitation experiment process.

- **F18 - Evaluate Agent:** The agent is evaluated for 5,000 episodes, matching the total training episode count.
- **F19 - Analyze State Visitation:** The states visited during evaluation are analyzed and plotted.
- **F20 - Decode States:** The evaluation states are decoded using the function from F1.
- **F21 - Create Trackers:** Two new trackers are created—one to track all decoded evaluation states and one for the 100 most visited states.
- **F22 - Create Heatmap Analyzers:** Two feature heatmap analyzers are created—one for all decoded states and one for the top 100 most visited states.
- **F23 - Log States:** The decoded states are logged using the trackers.
- **F24 - Analyze Heatmaps:** The feature heatmaps are analyzed for both cases.
- **F25 - Plot Heatmaps:** The heatmaps are plotted to visualize the final visitation distributions.

7.5.4 Results

In the following section, the results obtained from the analyzer are discussed. In Figure 7.21, the initial state visitation analysis can be observed. This analysis is conducted on all states visited during training. Examining the state distribution, it becomes apparent that three distinct sections emerge. The first section, located at the top, consists of states visited approximately 1200 times. The second section represents middle states with around 800 visits, and the third section gradually decreases from approximately 600 visits to below 100. However, without additional context, it is difficult to determine what each of these sections represents.

To gain further insight, the PCA-based feature heatmap for the state visitation is analyzed. As illustrated in Figure 7.22, it becomes evident that the resulting visualization does not provide meaningful information, suggesting that the state representation remains highly entangled and lacks clear structure.

Further analysis of the feature heatmaps in Figure 7.23 indicates that most feature activations are dominated by extreme values, producing little variation across features. This suggests that the agent’s policy remains unstable and unfocused during training. The training visitation patterns appear to consist of three distinct behavioral stages: an initial phase characterized by random exploration, a transitional phase with partial policy structure, and a final phase representing early signs of learned behavior. Since these stages overlap, the aggregated data provides limited interpretive value.

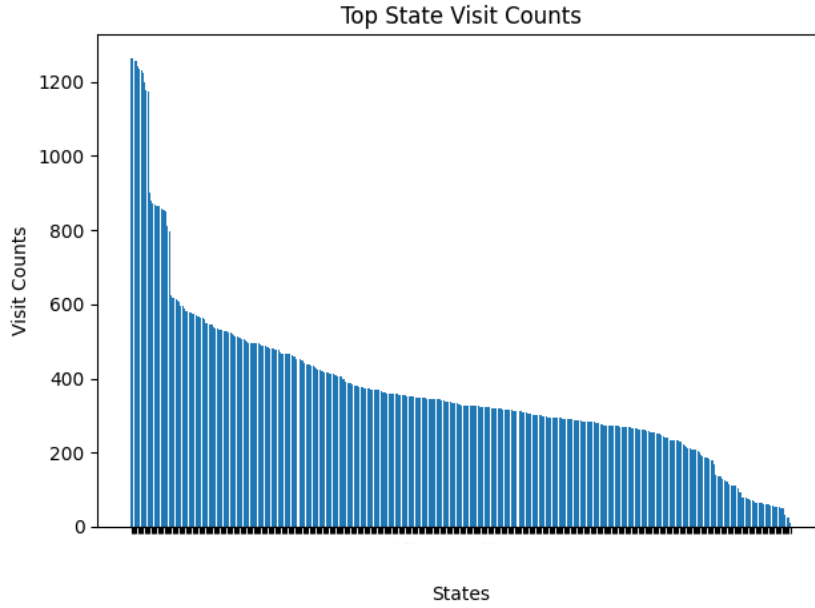


Figure 7.21: State visitation distribution during training.

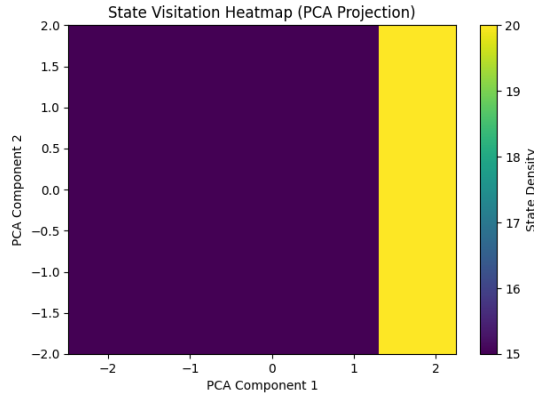


Figure 7.22: PCA-based feature heatmap of state visitation during training.

The histograms in Figure 7.23 indicates that each feature is distributed uniformly, showing that the agent receives an approximately equal combination of feature activations. This suggests that the training session has been thorough in its randomization, ensuring that the agent is exposed to a sufficiently diverse set of states to develop a proper understanding of the problem.

Focusing instead on the evaluation phase provides a clearer picture. As shown in Figure 7.24, the state visitation distribution during evaluation reveals three visible sections that correspond to the core, supporting, and peripheral states of the learned policy. The core states represent the most frequently visited and essential states for task completion, while the secondary and random states appear less consistent, possibly resulting from environmental variability and initialization effects.

The PCA heatmap in Figure 7.25 supports this observation, showing that most state

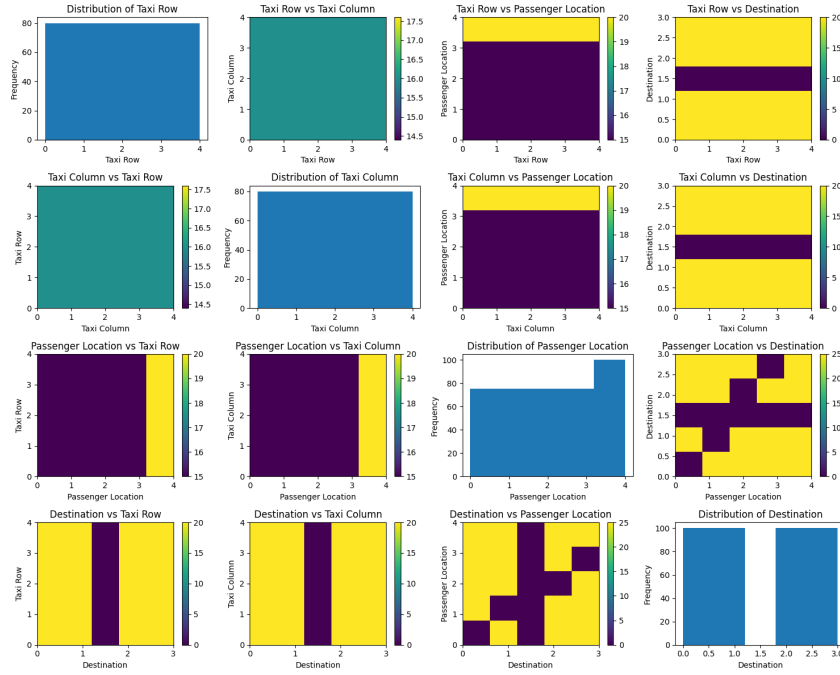


Figure 7.23: Feature heatmap distribution during training.

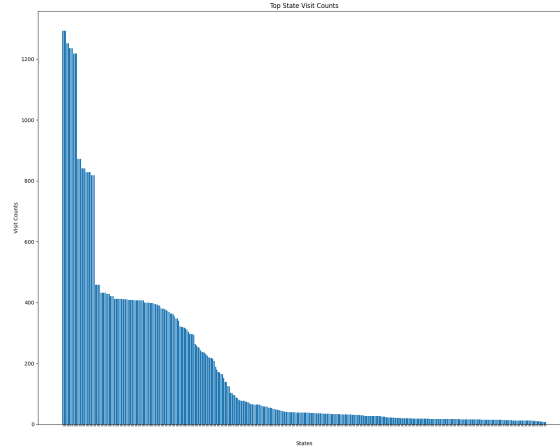


Figure 7.24: State visitation distribution during evaluation.

combinations converge toward a small number of dominant strategies, reflecting the agent’s consistent policy structure.

The corresponding feature distributions in Figure 7.26 demonstrate that the agent frequently operates along the central rows and within the first and fourth columns of the environment (Figure 4.2). This pattern is further confirmed by the heatmaps, where these regions exhibit the highest visitation intensity. Additionally, the agent spends considerably more time with the passenger outside the vehicle than inside, suggesting that it requires longer on average to locate and pick up passengers than to deliver them.

This analysis can be refined further by examining the top 100 most frequently visited states. As shown in Figure 7.27, these states define the agent’s core strategy. Two distinct

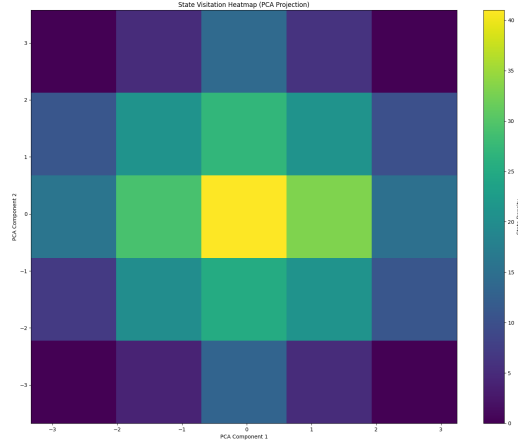


Figure 7.25: PCA feature heatmap across all evaluated states.

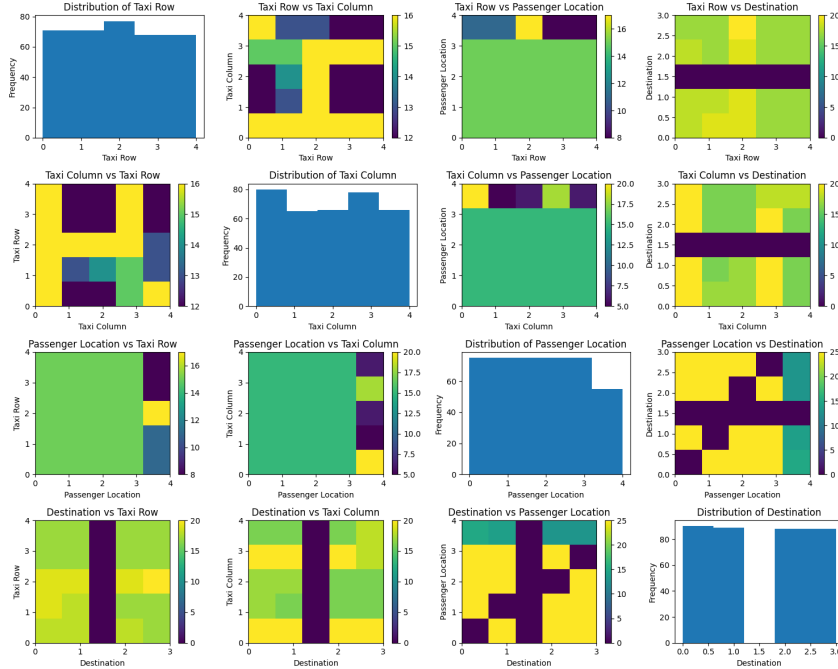


Figure 7.26: Feature distribution heatmap for all evaluated states.

clusters are visible, corresponding to the central row and the first and fourth columns, which the agent prioritizes when navigating the environment.

A closer look at the detailed feature heatmaps in Figure 7.28 confirms this behavior. The agent clearly favors the central row and border columns, reflecting optimal routing within the environment. Furthermore, the passenger distribution shifts significantly toward “in-vehicle”² states, indicating that most high-frequency states correspond to successful passenger delivery phases. This suggests that the agent’s training has effectively converged toward its goal of delivering a passenger.

²Passenger states 0, 1, 2, and 3 represent the four possible passenger locations, and state 4 indicates that the passenger is inside the vehicle.

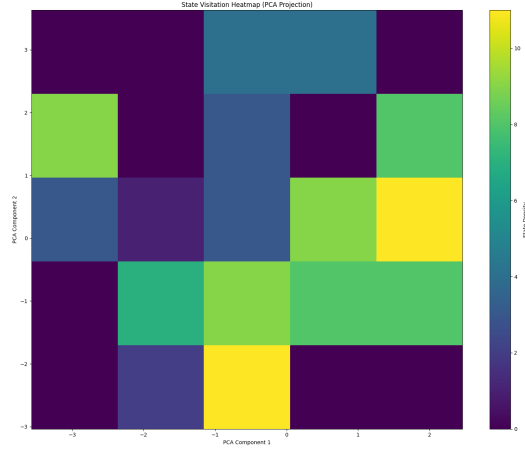


Figure 7.27: PCA heatmap of the top 100 most visited states.

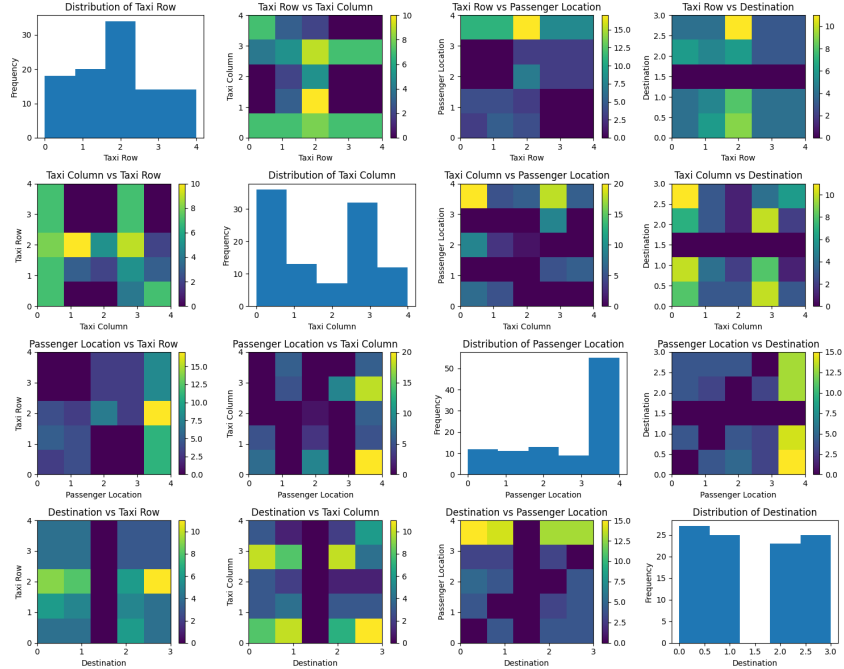


Figure 7.28: Feature heatmap of top 100 most visited states.

7.5.5 Conclusion

From the results, it is evident that the agent receives a well-balanced and distributed set of features from the environment, allowing it to experience a diverse range of states. Furthermore, the evaluation shows that the agent develops a consistent and clear strategy of picking up the passenger and delivering them efficiently by navigating the optimal route along row two and columns one and four. Therefore, even without directly visualizing the agent's behavior or policy, the underlying policy can be interpreted and examined effectively, even within a more complex environment. Hence, the hypothesis is confirmed to be correct.

This method directly illustrates the agent's strategy, or it can be derived using feature

heatmaps, as performed here. This demonstrates how an agent’s strategy can be interpreted and explained, enhancing the interpretability of the agent and making it clear how the agent is reasoning.

7.6 Feature selection experiment

In this section, the feature selection experiment is conducted. Four types of agents are utilized: DDQN, PPO, Q-Learning, and a Random agent with action masking, all within the CartPole environment. The goal of this experiment is to highlight the importance of input features and examine their influence on agent performance.

7.6.1 Experimental question

How does the selection of input features affect the performance and decision-making of reinforcement learning agents?

7.6.2 Hypothesis

It is expected that agents trained with more comprehensive feature sets will achieve better performance and more stable decision boundaries, while agents with limited or masked features will demonstrate reduced performance and suboptimal policy behavior.

7.6.3 Methodology

The methodology for this experiment is illustrated in Figure 7.29.

- **F1 - Create Environment:** The CartPole environment is initialized.
- **F2 - Create Agents:** DDQN and PPO agents are created and configured according to the experiment’s settings.
- **F3 - Execute Loop for Each Agent:** The following procedure is performed for each agent.
- **F4 - Create Tracker:** A tracker is initialized to log agent performance and training metrics.
- **F5 - Analyze Initial Decision Boundaries:** The agent’s decision boundaries are analyzed prior to training.
- **F6 - Train Agent:** The agent is trained in the CartPole environment for 200 episodes.

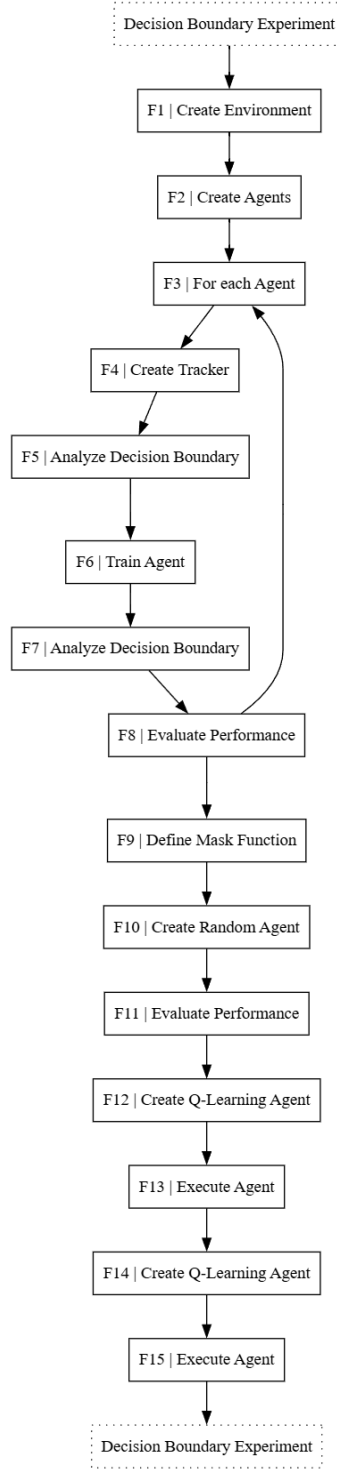


Figure 7.29: Flow diagram of the feature selection experiment procedure.

- **F7 - Analyze Post-Training Decision Boundaries:** After training, the agent's decision boundaries are analyzed and saved.
- **F8 - Evaluate Performance:** The agent's performance is evaluated following training.
- **F9 - Define Mask Function:** For the Random agent, a decision boundary from

the PPO agent is sampled, requiring the definition of an action masking function.

- **F10 - Create Random Agent:** The Random agent is initialized using the defined masking function.
- **F11 - Analyze Random Agent Performance:** The Random agent's performance is evaluated.
- **F12 - Create Q-Learning Agent:** A Q-Learning agent is initialized.
- **F13 - Train Q-Learning Agent:** The Q-Learning agent is trained on the CartPole environment for 100,000 episodes and subsequently evaluated over 100 episodes; results are recorded.
- **F14 - Create Reduced-Feature Agent:** An agent is created using only velocity, pole angle, and angular velocity, ignoring the position feature.
- **F15 - Train Reduced-Feature Agent:** This agent is trained for 100,000 episodes and evaluated over 100 episodes; results are recorded.

7.6.4 Results

In this section, the results of the feature selection experiment are discussed. As can be seen in Figures 7.30 and 7.31, the initial decision boundaries of the PPO and DDQN agents for a subset of features (angle vs. angular velocity) show that the agents are unsure which action to take (red for right or blue for left) and have not yet undergone any training. This aligns with expectations.

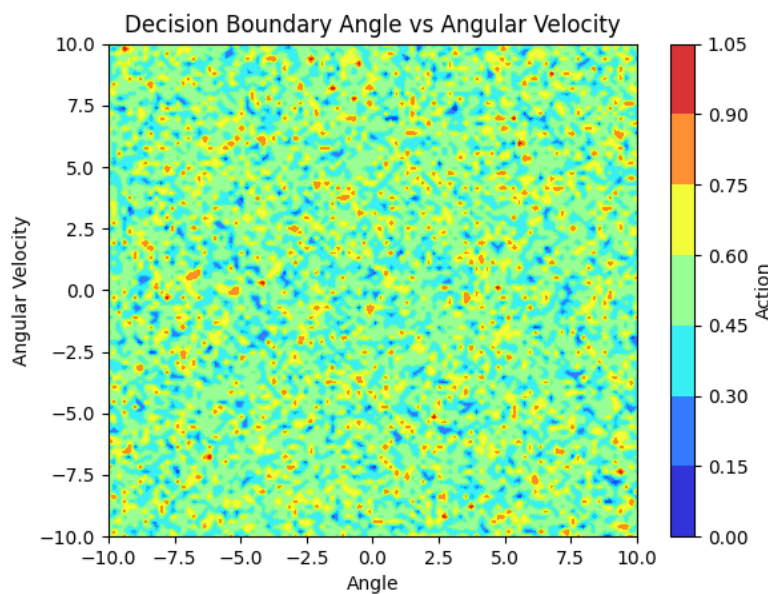


Figure 7.30: Initial decision boundary of the DDQN agent prior to training.

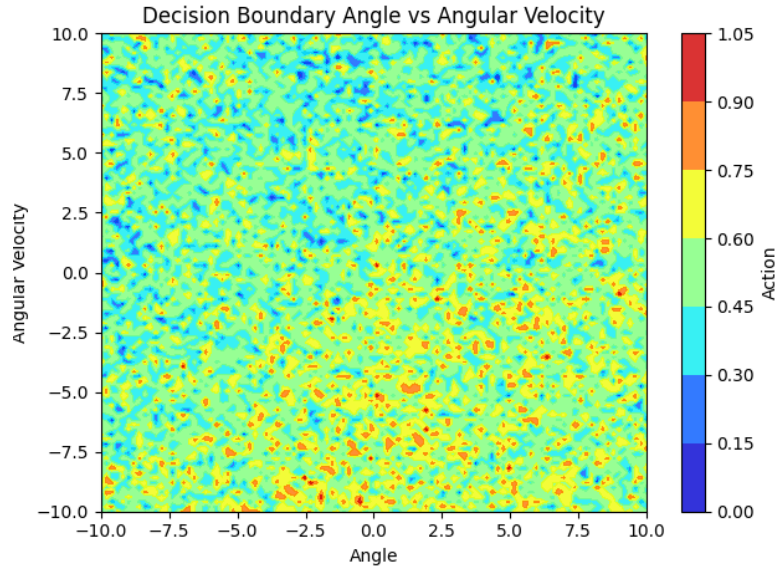


Figure 7.31: Initial decision boundary of the PPO agent prior to training.

After training, both agents show clearly defined decision boundaries for the action they are likely to take based on the input of the pole angle and angular velocity, as seen in Figures 7.32 and 7.33. Interestingly, the DDQN agent’s decision boundary is not as clean and simple as the PPO agent’s, and the boundaries indicate opposite actions. This raises the question of how two completely different decision mappings can still result in successful performance.

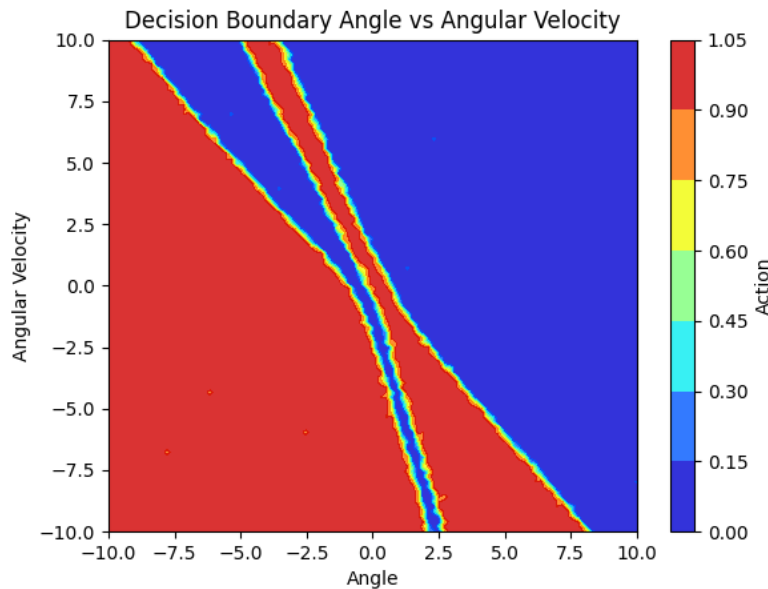


Figure 7.32: Post-training decision boundary of the DDQN agent.

Evaluation performance highlights this difference: the DDQN agent underperforms, while the PPO agent achieves a perfect record over 100 episodes (Figures 7.34 and 7.35)³. This

³The loss curves for these agents have no meaningful interpretation, as they represent only the evaluation

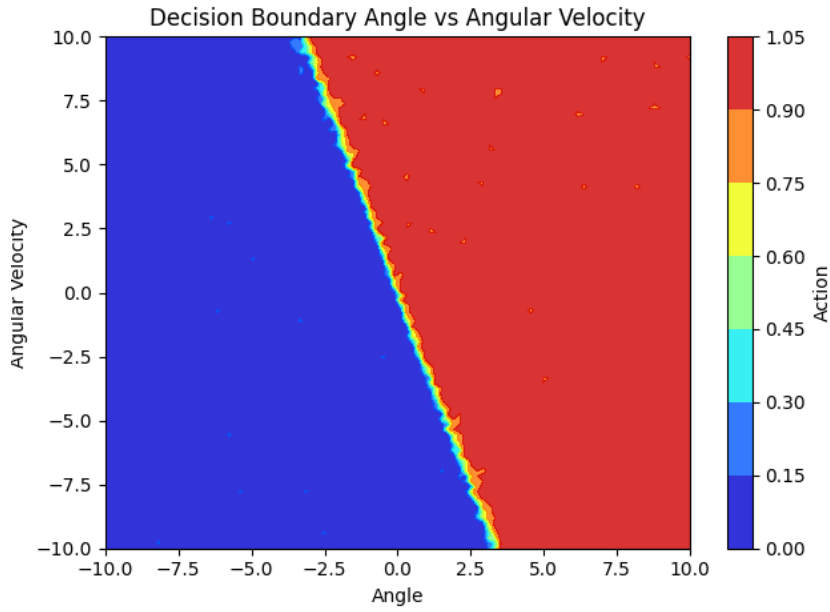


Figure 7.33: Post-training decision boundary of the PPO agent.

explains why the DDQN decision boundary appears less organized and why its opposing actions still result in some success: multiple valid policies exist for solving CartPole, but DDQN is simply performing worse in this case.

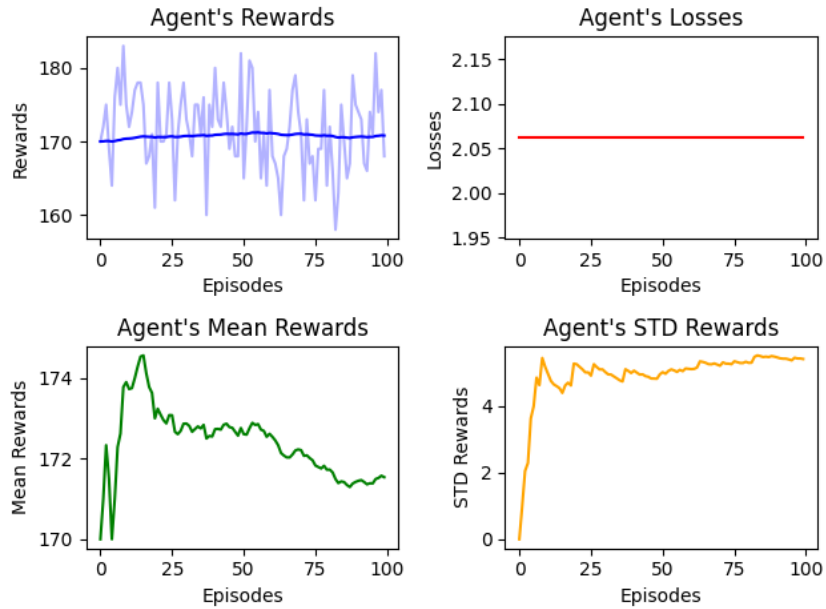


Figure 7.34: Evaluation performance of the DDQN agent over 100 episodes.

Further analysis of decision boundaries between position and the other features (Figures 7.36, 7.37, 7.38) shows that the position feature minimally alters the PPO agent's decisions. The agent is more responsive to other features, indicating that position con-

of the agents. The evaluation system merely records the final loss value obtained during training and repeats it; therefore, these curves do not convey any substantive information.



Figure 7.35: Evaluation performance of the PPO agent over 100 episodes.

tributes less to the decision-making process. For Q-Learning agents, removing the position feature significantly improves performance, as it reduces the number of possible state-action mappings the agent must learn (Figures 7.39, 7.40).

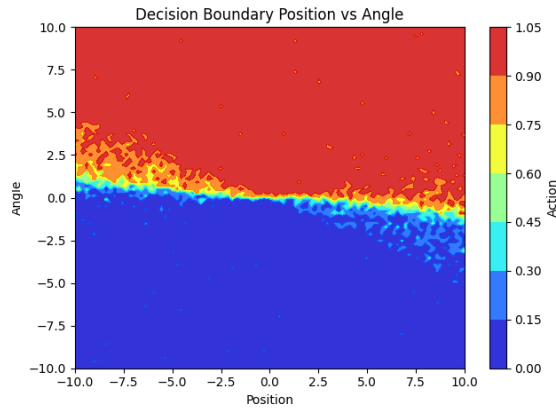


Figure 7.36: Decision boundary of the PPO agent for position vs. pole angle.

Finally, a simplified policy was constructed using only six linear boundaries extracted from the PPO agent's decision function. Evaluating this simplified agent over 100 episodes demonstrates that even with only a few linear decision rules, the agent achieves performance comparable to the full PPO neural network (Figure 7.41). This opens up an entirely new field of research and application known as TinyML, where simplified versions or smaller adaptations of advanced RL agents are used.

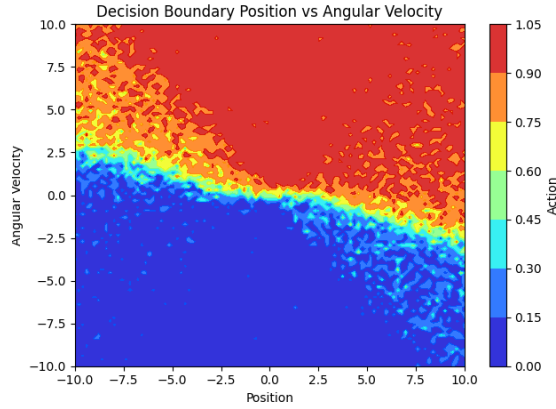


Figure 7.37: Decision boundary of the PPO agent for position vs. angular velocity.

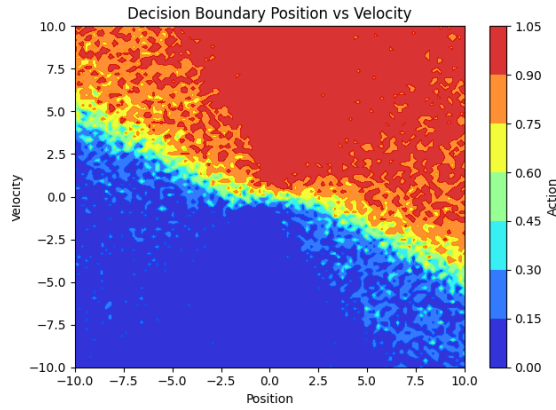


Figure 7.38: Decision boundary of the PPO agent for position vs. velocity.

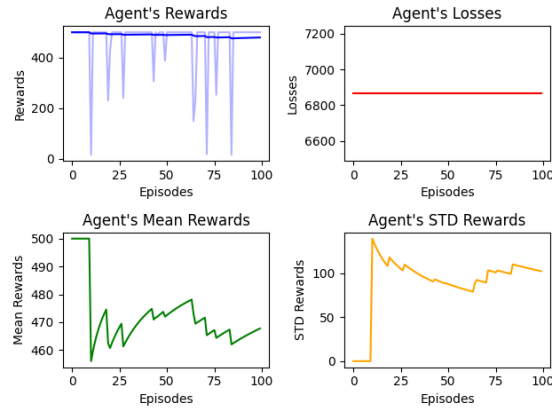


Figure 7.39: Evaluation performance of the Q-Learning agent trained without the position feature over 100 episodes, demonstrating improved control due to reduced state space complexity.

7.6.5 Conclusion

This experiment demonstrates that more features are not always better; the relevance of features depends on the agent and task. Proper feature selection allows reinforcement



Figure 7.40: Evaluation performance of the Q-Learning agent trained with all features over 100 episodes, showing lower performance compared to the feature-reduced agent due to larger state space complexity.



Figure 7.41: Performance of a simplified PPO agent using only six linear decision boundaries.

learning agents to achieve high performance with fewer input features. Q-Learning benefits significantly from removing less informative features, whereas PPO can produce effective policies even with all features, and its decision boundaries can be approximated using simple linear rules. This highlights the importance of careful feature engineering and decision boundary analysis in reinforcement learning.

Furthermore, the method highlights how the environment can be interpreted, how the agent's understanding of the feature profile can be visualized and internally enhanced, enhancing the agent's interpretability, and allowing for such deep understanding that the agent can effectively be cloned using a completely different set of decision-making mechanisms but generally following the same policy, greatly enhancing the agent's interpretability.

7.7 Example

As a final example, interpretability methods can also be used to debug agents during implementation and training. For instance, the REINFORCE agent, which struggled during the implementation validation experiment, only reaches reward values around 0.2, compared to other agents that achieve values close to 1. This issue can be investigated and addressed using the same interpretability methods.

Starting by examining the state distribution of the REINFORCE agent in the Frozen Lake environment, Figure 7.42 shows that over 2000 episodes, one state is visited on average 3 times per episode, while some other states are visited only once, and the remaining states are visited in less than half of all training episodes. This reveals a major issue: the agent does not fully explore the environment and fails to experience many states.

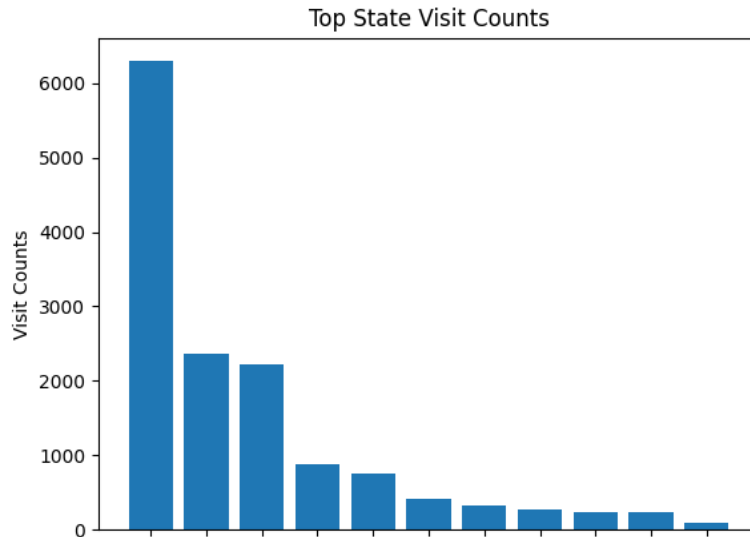


Figure 7.42: State visitation distribution of the REINFORCE agent over 2000 episodes. The agent rarely visits many states, indicating insufficient exploration.

Another issue arises from the Monte Carlo nature of the REINFORCE algorithm, which updates the policy at the end of each episode. If all rewards in an episode are the same, the agent does not learn effectively. In the FrozenLake environment, for example, rewards are only given for successfully reaching the goal, making learning difficult.

A first intuitive solution is to alter the environment by assigning negative rewards for falling into holes. However, this can have the opposite effect: early failures discourage learning, and the agent may avoid exploring altogether. Instead, a more gradual approach is effective: allow the agent to experience some uncertainty by visiting a few states multiple times, then progressively introduce negative rewards for unsafe actions, providing additional training signals. This approach was applied, and the results are shown in Figure 7.43.

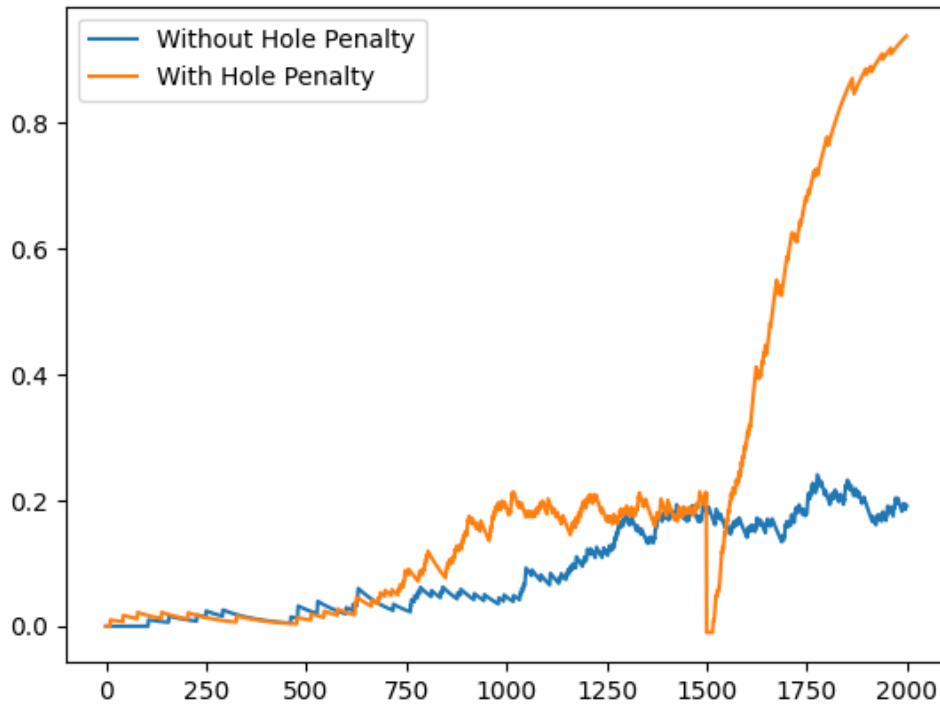


Figure 7.43: Training results after modifying the environment to include negative rewards for falling into holes. The REINFORCE agent eventually achieves optimal performance after initial poor results.

In Figure 7.43, it is evident that the agent achieves a perfect score after the initial 2000 episodes. A small alteration to the environment, introducing negative rewards for falling into holes after episode 1500, dramatically increases the number of learning opportunities, as each failure now provides informative feedback. This allows the agent to improve its policy significantly and achieve optimal performance.

In conclusion, interpretability methods are a powerful tool to analyze and debug agents during training. In this case, they demonstrate that the REINFORCE agent is implemented correctly and provide insight into how environment modifications and reward shaping can improve learning outcomes. By understanding how REINFORCE works at a fundamental level and applying interpretability analysis was this improvement possible.

7.8 Conclusion

This chapter contained six experiments, each using agents in different environments and applying the analysis methods outlined in Chapter 5. These experiments illustrate how such analyses can be used to generate useful insights and improve understanding of agent behavior. The results demonstrate improved understanding of agent action policies, high-

lighting how the explanation and interpretability of an agent are enhanced when using the analysis methods presented.

Although many experiments were executed only once, this does not present a significant hindrance. Since the interpretability techniques are applied to an agent regardless of its performance, they can still provide meaningful results. The aim was not to measure the agent's performance but to analyse its behaviour, independent of outcomes. In cases where multiple runs were necessary, they have been performed.

Chapter 8

Conclusion

This chapter presents the concluding discussion based on the research implementation, experimentation, and results obtained from Chapters 2 to 7. The chapter begins with a summary of each preceding chapter, followed by a final conclusion that addresses the main research question. Furthermore, it outlines possible areas for future research and suggests how this study can be continued and expanded.

8.1 Summary

Chapter 1 provides the background and motivation for addressing current challenges in reinforcement learning (RL). It highlights the problem statement and establishes the rationale for focusing on transparency and interpretability rather than performance alone. The chapter defines the scope and limitations of the study and introduces the main research question: *How can first-principles implementations of RL agents enhance our understanding of their behaviour and improve intelligibility in controlled environments?* This question is divided into six sub-questions, each addressed in Chapters 2 through 7. The chapter concludes with an overview of the overall report structure.

Chapter 2 introduces the theoretical and mathematical foundations of RL, explaining how RL agents function. It discusses concepts such as Markov Decision Processes (MDPs), policies, value functions, the Bellman equations, dynamic programming, Monte Carlo methods, temporal-difference learning, policy gradient methods, the policy gradient theorem, the advantage function, entropy regularization, and the exploration–exploitation trade-off. This chapter provides the theoretical framework necessary to answer the first part of the initial sub-question: *How do reinforcement learning agents work?*

The second part of this sub-question is addressed in Chapter 3, which presents a taxonomy of RL agents and categorizes them according to their underlying mechanisms. It offers conceptual explanations of how these agent types operate and finalizes the answer to the first sub-question. The chapter then identifies four agents that are selected for

experimentation in Chapter 7.

Chapter 4 addresses the second sub-question: *In what environments will the agents be trained?* It describes the different environments used during experimentation and explains their structure and relevance. The chapter concludes with an overview of the experimental methodology.

Chapter 5 addresses the third and fourth sub-questions: *What is intelligibility?* and *Which techniques are available to interpret the behaviour of an agent?* It begins by defining and formalizing the concept of intelligibility, distinguishing between its primary categories. It then explores existing and novel methods for improving intelligibility in RL agents, thereby addressing both sub-questions comprehensively.

Chapter 6 focuses on the fifth sub-question: *How can implementation and evaluation improve intelligibility?* It demonstrates how the theoretical and conceptual principles discussed in earlier chapters are integrated into practical implementations. This bridges the gap between theoretical understanding and practical application, making agent operations more transparent and comprehensible.

Finally, Chapter 7 and the current chapter address the final sub-question: *How does this process help improve the intelligibility of RL agents?* Chapter 7 illustrates how agents and environments can be investigated to visualize and interpret learning processes, showing how various training algorithms and intelligibility techniques enhance interpretability. This allows for improved evaluation and visualization of agent decision-making, ultimately increasing understanding and explainability.

8.2 Main question

How can first-principles implementations of RL agents enhance the understanding of their behaviour and improve intelligibility in controlled environments?

Chapter 7 presented several ways in which the application of intelligibility methods contributes to a deeper understanding of reinforcement learning (RL) agents. It demonstrated that a simple performance test can be used to verify whether agents are implemented correctly and, if not, how the principles introduced in Chapter 2 and the overarching ideas of Chapter 3 can be combined with intelligibility methods to debug or improve the REINFORCE agent so that it performs optimally. Additionally, it was shown that an agent's policy can be examined through various visualisation techniques. This revealed, for example, that value-based systems such as DDQN exhibit sharper and larger parameter updates compared with the smoother, more gradual updates of methods like REINFORCE and PPO.

Another key observation concerned the agents' sensitivity to learning rates. It was highlighted that DDQN agents are less sensitive to changes in learning rate than PPO agents,

due to exploration in PPO being closely tied to the learning rate. The analysis further showed how entropy influences REINFORCE and PPO agents: while increased entropy can improve consistency, excessively high entropy has the opposite effect. Moreover, it was demonstrated that an agent’s policy or general strategy can be visualised through multiple analytical techniques even when direct strategy visualisation is not feasible. Finally, decision boundaries were shown to be a powerful tool for determining whether an agent has converged to a high-quality solution, as well as for compressing an agent into a smaller “Tiny RL” model suitable for TinyML applications.

These methods revealed how the agent’s internal properties affect their training, which can all be extended to investigate other properties and new properties to learn and understand how they influence the agent’s training, thereby enhancing explainability of the agent. Furthermore, these methods revealed how an agent in training or a cloned trained agent can be examined and its decision-making understood through feature analysis, policy analysis, etc., thereby enhancing our understanding of the agent’s interpretability into its decision-making. These methods also provided a lot of insight into the agents; however, by adding the transparency of the `fund_rl` library aligned with the basic principles upon which these agents are built, one can thereby get an even greater understanding of these agents and proven facts, opening up the possibility of finding new facts and features of these agents.

Overall, these methods illustrated that first-principles implementations enable a thorough conceptual understanding of an agent alongside its implementation. By employing intelligibility techniques, several theoretical and practical characteristics of these agents were highlighted, showing how first-principles implementations enhance and improve intelligibility in controlled environments.

8.3 Future research directions

Although this research successfully addresses the primary research objective, there remain numerous avenues for further investigation. Many of the experiments conducted in this study were executed only once. Performing additional repetitions would provide deeper and more statistically robust results, though time constraints limited this in the current study—for example, the exploration consistency experiment required approximately 11 hours to complete.

Furthermore, the use of feature heatmaps presents additional opportunities. These visualizations could be leveraged to guide partially trained or structured agents, and similar approaches could be applied to decision boundary analysis for more comprehensive research.

Finally, alternative agent architectures and hybrid approaches could be explored. Methods such as Soft Actor-Critic or other policy/value hybrid algorithms may allow for increased

versatility and improved performance, expanding upon the work presented here. These directions could yield more generalizable insights into agent behavior, learning dynamics, and exploration strategies.

8.4 Final conclusion

In conclusion, this research demonstrates that first-principles implementations of RL agents significantly enhance the understanding of their inner workings and improve interpretability within controlled environments. By combining the theoretical foundations of Chapter 2 and the conceptual taxonomy of Chapter 3 with the transparent implementations in Chapter 6, the study effectively bridges theory and practice. Moreover, the structured environments discussed in Chapter 4 and the intelligibility methods introduced in Chapter 5 provide a systematic way to evaluate and explain agent behaviour. The results presented in Chapter 7 confirm that these approaches meaningfully improve the intelligibility of RL agents, offering valuable insights for both academic research and practical applications.

References

- [1] S. Hajkowicz, C. Sanderson, S. Karimi, A. Bratanova, and C. Naughtin, “Artificial intelligence adoption in the physical sciences, natural sciences, life sciences, social sciences and the arts and humanities: A bibliometric analysis of research publications from 1960–2021,” *Technology in Society*, vol. 74, p. 102260, 2023, published as arXiv:2306.09145 [cs.CY]. [Online]. Available: <https://arxiv.org/abs/2306.09145>
- [2] A. Gilson, C. W. Safranek, T. Huang, V. Socrates, L. Chi, R. A. Taylor, and D. Chartash, “Performance of chatgpt across different versions in medical licensing examinations worldwide: Systematic review and meta-analysis,” *Journal of Medical Internet Research*, vol. 26, no. 1, p. e60807, 2024. [Online]. Available: https://www.jmir.org/2024/1/e60807/?utm_source=chatgpt.com
- [3] OpenAI, “Openai five defeats dota 2 world champions,” 2019, accessed: 2025-07-14. [Online]. Available: <https://openai.com/index/openai-five-defeats-dota-2-world-champions/>
- [4] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, “Mastering the game of go without human knowledge,” *Nature*, vol. 550, no. 7676, pp. 354–359, 2017. [Online]. Available: https://augmentingcognition.com/assets/Silver2017a.pdf?utm_source=chatgpt.com
- [5] K. Souchleris, G. K. Sidiropoulos, and G. A. Papakostas, “Reinforcement learning in game industry—review, prospects and challenges,” *Applied Sciences*, vol. 13, no. 4, p. 2443, 2023, published: 14 February 2023. [Online]. Available: <https://www.mdpi.com/2076-3417/13/4/2443>
- [6] Y. Bhadauria, G. Yadav, and N. K. Bharti, “Openai gym in machine learning,” *International Journal of Research Publication and Reviews*, vol. 6, no. 6, pp. 2739–2748, 2025, galgotias University. [Online]. Available: https://ijrpr.com/uploads/V6ISSUE6/IJRPR47900.pdf?utm_source=chatgpt.com
- [7] C. Glanois, D. Li, P. Weng, M. Zimmer, T. Yang, J. Hao, and W. Liu, “A survey on interpretable reinforcement learning,” *Machine Learning*, vol. 113, no. 8, pp. 5847–5890, 2024. [Online]. Available: <https://doi.org/10.1007/s10994-024-06543-w>
- [8] R. Guidotti, A. Monreale, S. Ruggieri, F. Turini, D. Pedreschi, and F. Giannotti, “A survey of methods for explaining black box models,” *arXiv preprint arXiv:1802.01933*, 2018. [Online]. Available: <https://arxiv.org/abs/1802.01933>

- [9] T. Zahavy, N. Ben-Zrihem, and S. Mannor, “Graying the black box: Understanding dqns,” in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 1899–1908. [Online]. Available: <https://proceedings.mlr.press/v48/zahavy16.html>
- [10] Y. Qing, S. Liu, J. Song, H. Wang, and M. Song, “A survey on explainable reinforcement learning: Concepts, algorithms, challenges,” *arXiv preprint*, vol. arXiv:2211.06665, 2022. [Online]. Available: <https://arxiv.org/abs/2211.06665>
- [11] M. Steyvers and A. Kumar, “Three challenges for ai-assisted decision-making,” *Perspectives on Psychological Science*, vol. 19, no. 5, pp. 722–734, 2024, epub 2023 Jul 13.
- [12] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, Massachusetts and London, England: The MIT Press, 2018.
- [13] K. Siegrist, *Introduction to Markov Processes*, 2022, university of Alabama in Huntsville via Random Services, Last updated April 24, 2022, CCBY licensed. [Online]. Available: [https://stats.libretexts.org/Bookshelves/Probability_Theory/Probability_Mathematical_Statistics_and_Stochastic_Processes_\(Siegrist\)/16%3A_Markov_Processes/16.01%3A_Introduction_to_Markov_Processes](https://stats.libretexts.org/Bookshelves/Probability_Theory/Probability_Mathematical_Statistics_and_Stochastic_Processes_(Siegrist)/16%3A_Markov_Processes/16.01%3A_Introduction_to_Markov_Processes)
- [14] Oxford University Press, “Policy,” 2025, accessed: 2025-02-01. [Online]. Available: <https://www.oed.com/search/dictionary/?scope=Entries&q=policy&tl=true>
- [15] Baeldung, “Epoch vs episode in reinforcement learning,” <https://www.baeldung.com/cs/epoch-vs-episode-reinforcement-learning>, 2024, accessed: 2025-09-29.
- [16] OpenAI. (2018) Part 1: Key concepts in rl. OpenAI Spinning Up. Accessed: April 17, 2025. [Online]. Available: https://spinningup.openai.com/en/latest/spinningup/rl_intro.html
- [17] GeeksforGeeks, “Dynamic programming in reinforcement learning,” 2025, last accessed: May 5, 2025. [Online]. Available: <https://www.geeksforgeeks.org/dynamic-programming-in-reinforcement-learning/>
- [18] A. Nanda, “Policy gradient theorem explained: A hands-on introduction,” <https://www.datacamp.com/tutorial/policy-gradient-theorem>, DataCamp, 2025, accessed: 2025-07-24; Updated: February 13, 2025.
- [19] OpenAI Spinning Up, “Vanilla policy gradient (vpg),” <https://spinningup.openai.com/en/latest/algorithms/vpg.html>, OpenAI, 2018–, accessed: 2025-07-24.
- [20] Alooba, “Entropy: A fundamental concept in data science,” <https://www.alooba.com/skills/concepts/data-science/entropy>, 2025, accessed: 2025-10-03.
- [21] F. AlMahamid and K. Grolinger, “Reinforcement learning algorithms: An overview and classification,” in *2021 IEEE Canadian Conference on Electrical and*

Computer Engineering (CCECE). IEEE, Sep. 2021, p. 1–7. [Online]. Available: <http://dx.doi.org/10.1109/CCECE53047.2021.9569056>

- [22] H. Singh, “What is reinforcement learning and how does it work (updated 2025),” *Analytics Vidhya*, January 2025, last accessed: 3 February 2025. [Online]. Available: <https://www.analyticsvidhya.com/blog/2021/02/introduction-to-reinforcement-learning-for-beginners/>
- [23] OpenAI, “Part 2: Kinds of rl algorithms,” 2018, accessed: 2025-02-03. [Online]. Available: <https://spinningup.openai.com/en/latest/spinningup/rl.intro2.html#id19>
- [24] I. A. Lab, “Selecting an algorithm - intel coach,” 2019, accessed: 2025-02-03. [Online]. Available: https://intellabs.github.io/coach/selecting_an_algorithm.html
- [25] B. Li, H. Tang, Y. Zheng, J. Hao, P. Li, Z. Wang, Z. Meng, and L. Wang, “Hyar: Addressing discrete-continuous action reinforcement learning via hybrid action representation,” 2022. [Online]. Available: <https://arxiv.org/abs/2109.05490>
- [26] Scopus, “Scopus: Access and Use,” 2025, accessed: Feb. 13, 2025. [Online]. Available: <https://www.scopus.com>
- [27] R. Frigg and S. Hartmann, “Models in science,” *The Stanford Encyclopedia of Philosophy*, 2020. [Online]. Available: <https://plato.stanford.edu/entries/models-science/>
- [28] Unknown. (2023, December) On-policy vs off-policy methods in reinforcement learning. Accessed: 2025-02-03. [Online]. Available: <https://www.geeksforgeeks.org/on-policy-vs-off-policy-methods-reinforcement-learning/>
- [29] GeeksforGeeks, “Sarsa (state-action-reward-state-action) in reinforcement learning,” <https://www.geeksforgeeks.org/machine-learning/sarsa-reinforcement-learning/>, 2025, last updated: 17 Jul 2025, Accessed: 2025-10-06.
- [30] MathWorks, “Actor-critic (ac) agent,” 2025, accessed: 2025-02-26. [Online]. Available: <https://www.mathworks.com/help/reinforcement-learning/ug/actor-critic-agents.html>
- [31] GeeksforGeeks, “Actor-critic algorithm in reinforcement learning,” 2025, accessed: 2025-02-26. [Online]. Available: <https://www.geeksforgeeks.org/actor-critic-algorithm-in-reinforcement-learning/>
- [32] R. Williams, “Simple statistical gradient-following algorithms for connectionist reinforcement learning,” *Machine Learning*, vol. 8, pp. 229–256, May 1992. [Online]. Available: <https://doi.org/10.1007/BF00992696>
- [33] GeeksforGeeks, “Proximal Policy Optimization (PPO) in Reinforcement Learning,” 2025, accessed: Feb. 26 2025. [Online]. Available: <https://www.geeksforgeeks.org/a-brief-introduction-to-proximal-policy-optimization/>

- [34] OpenAI, “Trust region policy optimization,” 2025, accessed: 2025-02-26. [Online]. Available: <https://spinningup.openai.com/en/latest/algorithms/trpo.html>
- [35] —, “Proximal policy optimization,” Available online, 2018, accessed: 2025-02-26. [Online]. Available: <https://spinningup.openai.com/en/latest/algorithms/ppo.html>
- [36] OpenAI, “Proximal policy optimization (ppo),” July 20 2017, accessed: 2025-10-07. [Online]. Available: <https://openai.com/index/openai-baselines-ppo>
- [37] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” *arXiv preprint arXiv:1707.06347*, 2017.
- [38] GeeksforGeeks, “Q-learning in reinforcement learning,” <https://www.geeksforgeeks.org/machine-learning/q-learning-in-python/>, 2025, last updated: 11 Jul 2025, Accessed: 2025-10-06.
- [39] MathWorks, “Deep q-network (dqn) agent,” 2025, accessed: 2025-03-03. [Online]. Available: <https://www.mathworks.com/help/reinforcement-learning/ug/dqn-agents.html>
- [40] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” *DeepMind Technologies*, 2013. [Online]. Available: <https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf>
- [41] Y. Chen, “Comparison of deep q-learning network and double deep q-learning network for trading strategy,” in *Proceedings of the 5th EAI International Conference on Computer Science and Engineering*, Leeds, United Kingdom, 2023. [Online]. Available: <https://eudl.eu/pdf/10.4108/eai.27-10-2023.2341940>
- [42] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, “Dueling network architectures for deep reinforcement learning,” 2016. [Online]. Available: <https://arxiv.org/abs/1511.06581>
- [43] MathWorks, “Deep deterministic policy gradient (ddpg) agent,” 2024, last accessed: 4 March 2025. [Online]. Available: <https://www.mathworks.com/help/reinforcement-learning/ug/ddpg-agents.html>
- [44] M. G. Bellemare, W. Dabney, and R. Munos, “A distributional perspective on reinforcement learning,” 2017. [Online]. Available: <https://arxiv.org/abs/1707.06887>
- [45] S.-B. Contributors, “Quantile regression dqn (qr-dqn),” 2024, accessed: 2025-03-04. [Online]. Available: <https://sb3-contrib.readthedocs.io/en/master/modules/qrdqn.html>
- [46] W. Dabney, M. Rowland, M. G. Bellemare, and R. Munos, “Distributional reinforcement learning with quantile regression,” 2017. [Online]. Available: <https://arxiv.org/abs/1710.10044>

- [47] MathWorks, “Td3 (twin delayed deep deterministic policy gradient) agent,” 2024, accessed: 2025-03-04. [Online]. Available: <https://www.mathworks.com/help/reinforcement-learning/ug/td3-agents.html>
- [48] —, “Soft actor-critic (sac) agents,” 2024, accessed: 2025-03-04. [Online]. Available: <https://www.mathworks.com/help/reinforcement-learning/ug/soft-actor-critic-agents.html>
- [49] APXML, “Comparing value-based and policy-based methods,” <https://apxml.com/courses/intro-to-reinforcement-learning/chapter-8-introduction-policy-gradient-methods/comparing-value-policy-based-methods>, 2025, accessed: 2025-10-06.
- [50] GeeksforGeeks, “Types of reinforcement learning,” <https://www.geeksforgeeks.org/machine-learning/types-of-reinforcement-learning/>, 2025, last updated: 23 Jul 2025, Accessed: 2025-10-06.
- [51] —, “Reinforce algorithm,” 2025, last updated: 05 Jun 2025, Accessed: 2025-10-06. [Online]. Available: <https://www.geeksforgeeks.org/machine-learning/reinforce-algorithm/>
- [52] C. J. Watkins and P. Dayan, “Q-learning,” *Machine Learning*, vol. 8, pp. 279–292, 1992, copyright © 2020 by Annual Reviews. All rights reserved.
- [53] J. Clifton and E. Laber, “Q-learning: Theory and applications,” *Annual Review of Statistics and Its Application*, vol. 7, pp. 279–301, 2020, copyright © 2020 by Annual Reviews. All rights reserved. [Online]. Available: <https://doi.org/10.1146/annurev-statistics-031219-041220>
- [54] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, pp. 529–533, February 2015. [Online]. Available: <https://doi.org/10.1038/nature14236>
- [55] D. Silver, “Deep reinforcement learning,” <https://deepmind.google/discover/blog/deep-reinforcement-learning>, 2016, accessed: 2025-10-06.
- [56] H. van Hasselt, A. Guez, and D. Silver, “Deep reinforcement learning with double q-learning,” 2015. [Online]. Available: <https://arxiv.org/abs/1509.06461>
- [57] F. Foundation, “Taxi environment — gymnasium,” 2025, https://gymnasium.farama.org/environments/toy_text/taxi.
- [58] —, “Frozenlake environment — gymnasium,” 2025, https://gymnasium.farama.org/environments/toy_text/frozen_lake.
- [59] —, “Cartpole environment — gymnasium,” 2025, https://gymnasium.farama.org/environments/classic_control/cart_pole.

- [60] B. VanHooker. (2017) An oral history of ‘snake’ on nokia. Mel Magazine, May 25, 2017. [Online]. Available: <https://melmagazine.com/en-us/story/snake-nokia-6110-oral-history-taneli-armanto>
- [61] D. Kosbie, “15-110 lecture topics: Snake,” <https://www.kosbie.net/cmu/fall-10/15-110/handouts/snake/snake.html>, 2010, week 9, Oct 18–22, Course: 15-110.
- [62] C. Glanois, P. Weng, M. Zimmer, D. Li, T. Yang, J. Hao, and W. Liu, “A survey on interpretable reinforcement learning,” *Mach. Learn.*, vol. 113, no. 8, pp. 5847–5890, 2024. [Online]. Available: <https://doi.org/10.1007/s10994-024-06543-w>
- [63] S. Huang, N. Papernot, I. J. Goodfellow, Y. Duan, and P. Abbeel, “Adversarial attacks on neural network policies,” in *ICLR 2017 Workshop Track*, 2017, workshop version. [Online]. Available: <https://openreview.net/pdf?id=ryvIRyBKl>
- [64] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, “Deep reinforcement learning that matters,” in *Proceedings of the 32nd AAAI Conference on Artificial Intelligence*, 2018, pp. 3207–3214. [Online]. Available: <https://ojs.aaai.org/index.php/AAAI/article/view/11694>
- [65] A. Alharin, T.-N. Doan, and M. Sartipi, “Reinforcement learning interpretation methods: A survey,” *IEEE Access*, vol. 8, pp. 171 058–171 077, 2020, received August 20, 2020; accepted August 31, 2020; published September 11, 2020; current version September 29, 2020.
- [66] T. Huber, K. Weitz, E. André, and O. Amir, “Local and global explanations of agent behavior: Integrating strategy summaries with saliency maps (extended abstract),” in *Proceedings of the 31st International Joint Conference on Artificial Intelligence, IJCAI 2022*, ser. IJCAI International Joint Conference on Artificial Intelligence, L. {De Raedt} and L. {De Raedt}, Eds. International Joint Conferences on Artificial Intelligence, Jan. 2022, pp. 5747–5751, publisher Copyright: © 2022 International Joint Conferences on Artificial Intelligence. All rights reserved.; 31st International Joint Conference on Artificial Intelligence, IJCAI 2022 ; Conference date: 23-07-2022 Through 29-07-2022.
- [67] I. Farrance and R. Frenkel, “Uncertainty of measurement: A review of the rules for calculating uncertainty components through functional relationships,” *Clinical Biochemist Reviews*, vol. 33, no. 2, pp. 49–75, 2012. [Online]. Available: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3387884/>
- [68] P. Gaines, Paul, “Accuracy, precision, mean and standard deviation,” ICP Operations Guide: Part 14, Inorganic Ventures, Inc., n.d., accessed: 2025-10-20. [Online]. Available: <https://www.inorganicventures.com/icp-guide/accuracy-precision-mean-and-standard-deviation>
- [69] T. Spooner, R. Savani, J. Fearnley, and A. Koukorinis, “Market making via reinforcement learning,” 07 2018.

- [70] S. Milani, N. Topin, M. Veloso, and F. Fang, “Explainable reinforcement learning: A survey and comparative review,” *ACM Computing Surveys*, vol. 56, 08 2023.
- [71] J. Martin, S. N. Sasikumar, T. Everitt, and M. Hutter, “Count-based exploration in feature space for reinforcement learning,” 2017. [Online]. Available: <https://arxiv.org/abs/1706.08090>
- [72] Deepchecks, “Decision boundary,” <https://www.deepchecks.com/glossary/decision-boundary/>, 2025, accessed: 2025-10-21.
- [73] F. Foundation, “Lunarlander environment — gymnasium,” 2025, https://gymnasium.farama.org/environments/box2d/lunar_lander.
- [74] R. Tangri. (2021) Generalized advantage estimate: Maths and code. [Online]. Available: <https://towardsdatascience.com/generalized-advantage-estimate-maths-and-code-b5d5bd3ce737/>

Appendix A

Code

The code consists of the `fund_rl` library, including the experimental code and examples demonstrating its usage. The library and all related resources are publicly available on GitHub, along with instructions for setup and downloading. The repository can be accessed at: https://github.com/Aizohiith/fund_rl.