Master's thesis

INTRINSICALLY MOTIVATED REINFORCEMENT LEARNING FOR EFFICIENT INTERFACE NAVIGATION BY AIVA

Bc. Pavel Chudomel

Faculty of Information Technology
Department of Applied Mathematics
Supervisor: Mgr. Petr Šimánek
May 9, 2025

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Intrinsically motivated reinforcement learning for efficient interface navigation by AIVA
Student:	Bc. Pavel Chudomel
Supervisor:	Mgr. Petr Šimánek
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2025/2026

Instructions

As autonomous systems become more integrated into daily life, the need for robots that can effectively interact with electronic devices like smartphones is increasing. AIVA, a robot designed to operate mobile interfaces through actions such as clicking and swiping, offers an opportunity to develop methods that enhance its efficiency in navigating complex user interfaces. The goal is to train AIVA to quickly and comprehensively explore all possible scenarios within a device's interface.
This project will focus on creating a method that enables AIVA to apply reinforcement learning (RL) strategies, and incorporate intrinsic motivation techniques to optimize the navigation process. Intrinsic motivation will drive AIVA to explore without explicit rewards, potentially leading to more efficient learning and faster task completion.

Steps:
1/ Literature Review: Review existing methods in reinforcement learning, intrinsic motivation, and robotic interface interaction, with a focus on applications in autonomous navigation and software testing.
2/ Explore Button Detection: Develop a method for detecting clickable buttons on a mobile interface, utilizing computer vision techniques or machine learning models, assess usability of such approach in AIVA.
3/ Reinforcement Learning Setup: Implement a basic RL (e.g. PPO - Proximal Policy Optimization) framework where AIVA learns to interact with the interface. Define state space (e.g., current screen), action space (e.g., clicks, swipes), and rewards (e.g., new

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 30 January 2025 in Prague.

screen reached).
4/ Intrinsic Motivation Integration: Implement and compare some intrinsic motivation methods, e.g. curiosity-driven exploration, to enhance AIVA's learning process. Possibly, try to combine these methods.
5/ Training and Optimization: Train the RL model with integrated intrinsic motivation and fine-tuning parameters to balance exploration and exploitation for optimal learning.
6/ Performance Evaluation: Assess the system's performance using metrics like time to complete navigation paths, e.g. the number of unique screens visited.
7/ Analysis and Comparison: Compare the impact of different methods on AIVA's navigation performance, analyzing contributions to speed and thoroughness.
8/ Conclusion and Future Work: Summarize findings, discuss strengths and limitations, and propose future research directions, such as refining motivation techniques or expanding the approach to other devices.
This project aims to advance autonomous robotic interaction with complex interfaces, offering insights for broader applications in human-robot interaction, software testing, and assistive technologies.

Citation of this thesis: Chudomel Pavel. *Intrinsically Motivated Reinforcement Learning for Efficient Interface Navigation by AIVA*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2025.

I would like to express my gratitude to my supervisor, Mgr. Petr Šimánek, for his valuable guidance, to my parents for their support, to the Y Soft team for their prompt technical assistance, and to the people from DataLab for their patience whenever the robot decided it was time for a spontaneous ringtone performance.

Declaration

Abstract

Efficient navigation in a complex user interface presents a significant challenge. This work focuses on the automatic exploration of such environments using reinforcement learning, leveraging intrinsic motivation methods as a driving force in the learning process. This work first establishes the theoretical foundations, starting from the fundamentals of reinforcement learning and gradually advances to more complex topics, which are implemented in the practical part on a real-world system AIVA, a robot designed for interaction with touchscreen devices. Initially, the possibility of using a deep learning model for detecting interactable elements on the screen is explored, but because of its high data requirements and low portability, this approach proves impractical. The reinforcement learning methods attempt to learn to identify and discover new screens using visual features derived from advanced feature extractors. This work demonstrates that a naïve reward implementation fails, and therefore two alternative reward mechanisms based on intrinsic motivation are proposed. The first of them uses Intrinsic Curiosity Module (ICM) and leads to a noticeable improvement in the exploration policy. The other method, Random Network Distillation (RND), because of its non-episodic approach, results in the robot being trapped in states it cannot easily escape from for an extended period. Despite that, the use of RND leads to an improvement in the robot's ability to explore the environment. The findings thus suggest that intrinsic motivation can improve the robot's exploration capability, but a number of aspects must be considered during its implementation, as they have a significant impact on the robot's overall performance.

Keywords reinforcement learning, intrinsic motivation, automated exploration, user interface navigation, robotic interaction

Abstrakt

Efektivní navigace v komplexním uživatelském rozhraní představuje zásadní výzvu. Tato práce se zaměřuje na automatické prozkoumávání takových prostředí pomocí zpětnovazebního učení s využitím metod vnitřní motivace, sloužících jako hnací síla procesu učení. Práce poskytuje nezbytné teoretické pozadí od základů zpětnovazebního učení a postupně se dostává ke složitějším tématům, které jsou v praktické části implementovány na skutečném systému AIVA, robotu navrženém pro interakci se zařízeními s dotykovou obrazovkou. Nejprve je prozkoumána možnost použití modelu hlubokého učení pro detekci interagovatelných prvků na obrazovce, která se však pro svou datovou náročnost a nízkou přenositelnost ukazuje jako nepraktická. Metody zpětnovazebního učení se z obrazových příznaků získaných pomocí pokročilých extraktorů snaží naučit poznávat a objevovat nové obrazovky. Práce ukazuje, že naivní implementace odměny selhává, a proto jsou navrženy dva alternativní mechanismy odměňování založené na vnitřní motivaci. První z nich používá Intrinsic Curiosity Module (ICM) a vede k citelnému zlepšení strategie prozkoumávání. Druhá metoda, Random Network Distillation (RND), kvůli svému neepizodickému přístupu dostává robota do stavů, z nichž se po dlouhou dobu nedokáže dostat. I přes to vede použití RND ke zlepšení schopnosti robota prozkoumávat prostředí. Výsledky tedy naznačují, že vnitřní motivace může zlepšit schopnost prozkoumávání robota, ale je nutné při její implementaci zvážit řadu aspektů, které mají významný vliv na celkové fungování robota.

Klíčová slova zpětnovazební učení, vnitřní motivace, autonomní prozkoumávání, navigace v uživatelském rozhraní, robotická interakce

Contents

List of Figures

List of Tables

List of abbreviations

RL	Reinforcement Learning
MDP	Markov Decision Process
MRP	Markov Reward Process
DP	Dynamic Programming
TD	Temporal Difference
GLIE	Greedy In The Limit with Infinite Exploration
SGD	Stochastic Gradient Descent
MSE	Mean Squared Error
RMSE	Root Mean Squared Error
DQN	Deep Q-Network
TRPO	Trust Region Policy Optimization
MM	Minorize-Maximize
KL	Kullback-Leibler
GAE	Generalized Advantage Estimation
CPI	Conservative Policy Iteration
PPO	Proximal Policy Optimization
ICM	Intrinsic Curiosity Module
RND	Random Network Distillation
YOLO	You Only Look Once
MIM	Masked Image Modelling
MLM	Masked Language Modelling
WPA	Word-Patch Alignment
GPU	Graphics Processing Unit
OCR	Optical Character Recognition
OS	Operating System
DDPG	Deep Deterministic Policy Gradient
TD3	Twin Delayed DDPG
SAC	Soft Actor-Critic
REDQ	Randomized Ensembled Double Q-Learning

Introduction

The automation of user interface testing, while aiming for efficiency and consistency, paradoxically remains dependent on complicated, manually defined workflows. User interfaces may be complex, and the only information that is always obtainable is a brief observation of the current state, e.g. screenshot. Navigating complex environments and processing images have been among the most challenging problems in computing, but recent breakthroughs in artificial intelligence changed the way these tasks are approached.

This work explores the potential of reinforcement learning methods to improve navigation efficiency in complex environments. Traditional navigation approaches often rely on predefined rules or heuristics, which can perform well in simple scenarios, but may struggle in more difficult settings. Using reinforcement learning, the exploring robot will be rewarded for exploring and discovering new states. The focus of this work is specifically on exploring applications that operate on electronic devices with a touch screen, such as smartphones, using the AIVA system of Y Soft [1].

As a main driver for exploration, this work investigates the use of intrinsic motivation. The robot is not rewarded extrinsically by the environment, but rewards itself for satisfying its intrinsic motivation. This motivation can be expressed through curiosity, where the robot is rewarded for performing actions with surprising outcomes, or novelty, where it is rewarded for exploring new, unseen states.

The ultimate end goal is to build a system that, after a period of time, learns to efficiently explore the complex system of a touch-screen device. Such a solution would be of great benefit not only in the test automation area but also in user experience optimization or automated assistance. It could create graphs describing the user interface and may help discover its weaknesses, bugs and potential issues. Since the system would adapt to various touch screen interfaces, the solution could assist in enhancing applications across different platforms.

The first chapter offers a comprehensive overview of standard methods and approaches in reinforcement learning, providing the theoretical background necessary for understanding how agents learn from interactions with the environment. Second chapter builds upon this foundation and explores more advanced topics in reinforcement learning — algorithms allowing scalable learning and the concept of intrinsic motivation, a technique that ensures continual learning of robots, even in environments with sparse or absent rewards. Chapter three presents other methods necessary for the robotic interface interaction, such as image processing and methods for extracting useful features from the screenshot. Fourth chapter discusses the setups and highlights practical considerations and pitfalls encountered during implementation and testing. Chapter five concludes the work with a presentation and discussion of the results.

Chapter 1

Foundations of Reinforcement Learning

This chapter describes the main ideas that have shaped the field of reinforcement learning. Starting with the general reinforcement learning problem definition, it gradually develops the ideas and explores methods of finding the optimal policy of a reinforcement learning agent. The goal of this chapter is to lay the foundation for the chapters that follow.

Machine learning was traditionally divided into three broad categories. Supervised learning algorithms use labeled data during training to try and learn patterns that can be utilized with new unseen examples. Unsupervised learning, in contrast, deals with unlabeled data and focuses on discovering and leveraging hidden structures or relationships within it. Reinforcement learning introduces a third paradigm, *learning through experience*.

Reinforcement learning algorithms closely mirror how humans learn from interacting with their environment. Humans affect the environment around them, learn from it and use that experience to guide future actions. The main distinction lies in the learning process.

A standard concept in machine learning is the backpropagation of error. After making a prediction (in supervised learning) or taking an action (in reinforcement learning), the result is evaluated using a problem-specific loss function (in supervised learning) or a reward function (in reinforcement learning). This error (or reward) signal is then used to update the parameters of the model, enabling it to improve performance over time. This concept will play central role later in this chapter and in Chapter 2.

Although recent theories suggest that it may be possible for the human brain to approximate the process of backpropagation [2], the mechanisms underlying human learning appear to differ significantly. An influential perspective is the Hebbian theory, often summarized as: "*neurons that fire together wire together*" [3] — when one neuron repeatedly activates another, the synap-

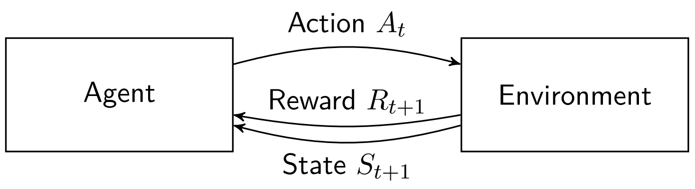


Figure 1.1 A diagram of the reinforcement learning loop.

tic connection between them becomes stronger over time [4]. Although the biological learning theories are interesting and relevant in the broader context of learning, a detailed exploration is beyond the scope of this thesis.

1.1 Definition of Reinforcement Learning

The foundational text *Reinforcement Learning: An Introduction* [5] by Richard S. Sutton and Andrew G. Barto serves as one of the main references for this chapter, as it provides the most comprehensive and widely accepted treatment of the field's core concepts, definitions, and algorithms. The book sets reinforcement learning apart as the problem, the class of solution methods, and the field that studies both. The problem can be intuitively understood as a continuous interaction between an *agent* and an *environment*. The agent performs an *action* in the environment and the environment responds with a new *state* and a *reward* that the agent is trying to maximize over time — this is how the objective of an agent is defined. A scheme of this is depicted in Figure 1.1. A method designed to solve a reinforcement learning problem can be considered a reinforcement learning method or algorithm [5].

The following sections aim to build a mathematical framework of reinforcement learning problem within which theorems can be stated and analyzed.

1.2 Markov Decision Process

A discrete-time stochastic process is defined as a system of dependent random variables

$$\mathbf{X} = \{X_n \mid n \in T\}$$

over a probability space $(\Omega, \mathcal{F}, \mathrm{P})$ and an index set $T \subseteq \mathbb{N}_0$ [6]. A stochastic process will be used to describe a system in which random state changes occur.

A specific subset of stochastic processes is *Markov processes*, introduced in 1907 by Andrey Andreyevich Markov [7]. A stochastic process $\{S_n \mid n \in \mathbb{N}_0\}$ with at most a countable set $\mathcal{S}$ is a discrete-time Markov process, if it fulfills the *Markovian property* $\forall n \in \mathbb{N}, \forall s, s_0, \ldots, s_{n-1} \in \mathcal{S}$:

$$\mathrm{P}(S_n = s \mid S_{n-1} = s_{n-1}, \ldots, S_0 = s_0) = \mathrm{P}(S_n = s \mid S_{n-1} = s_{n-1}). \text{ [6]}$$

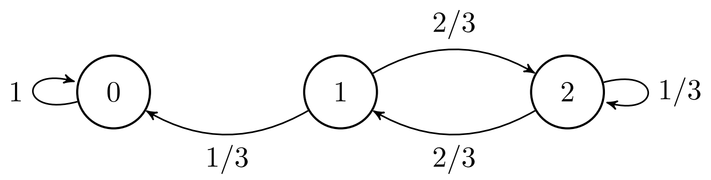


Figure 1.2 An example diagram of Markov process.

The key defining property is “forgetfulness” — at any given time t, it is possible to determine the probability of S_{t+1} only using the knowledge of S_t [6].

If $\mathcal{S}$ is the at most countable set of states in which the system can exist and $s_n, s_{n-1} \in \mathcal{S}$ are describing a control variable that defines the system's state, $\mathrm{P}(S_t = s_n \mid S_{t-1} = s_{n-1})$ can be thought of as the probability of state transition of the system from s_{n-1} to s_n at time step t. In Markov process, this transition probability is independent of t. Thanks to this property, a Markov process can be graphically illustrated as in Figure 1.2. Note that the sum of probabilities of transition from given state is one (transition occur in each time step) and the transitions with zero probability are not depicted in the figure.

The notion of states, understood as realizations of the random variables, and transitions, understood as changes between states, can formulate a more convenient definition of Markov process. This definition is suitable for finite state spaces: Markov process is a tuple $(\mathcal{S}, \mathcal{P})$, where $\mathcal{S} = s_1, s_2, \dots, s_n$ is a finite set of $n \in \mathbb{N}$ states and $\mathcal{P} \in \mathcal{S}^{n,n}$ is a *state transition probability matrix*, where $\mathcal{P}_{ss'} = \mathrm{P}(S_{t+1} = s' \mid S_t = s)$ [8].

Richard Bellman in *Dynamic Programming* [9], first published in 1957, describes Markovian Decision Process (later established as Markov Decision Process, MDP), as a part of broader framework of dynamic programming. MDPs later became crucial in the field of reinforcement learning, because they model the environment in the exact same way reinforcement learning does. In other words, had the MDPs not already existed, the field of reinforcement learning may have defined it independently by itself.

The goal of an agent can be formally expressed in terms of reward. A reward is a time-dependent value $R_{t+1} \in \mathbb{R}$ that the agent obtains in each time step. The total sum of rewards starting from time step t is denoted as *(discounted) return* G_t:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{i=1}^{\infty} \gamma^{i-1} R_{t+i}.$$

Parameter $\gamma \in \mathbb{R}, 0 \leq \gamma \leq 1$ is called the *discount factor* and adjusts the preference of short-term/long-term reward [5].

There are various reasons to use discount factor apart from mathematical convenience or to ensure convergence of returns in cyclic Markov processes. Immediate rewards may be preferred in a dynamic environments with high degree of uncertainty of the future, or short-term rewards may be preferred in finances, because it may yield more interest [8].

Consider Markov process $(\mathcal{S}, \mathcal{P})$ with a finite set of states $\mathcal{S}$ and state transition probability matrix $\mathcal{P}$. This can be expanded to *Markov reward process*, a tuple $(\mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma)$, where $\mathcal{R}$ is a *reward function* $\mathcal{R} : \mathcal{S} \to \mathbb{R}$,

$$\mathcal{R}_s = \mathbb{E}[R_{t+1} \mid S_t = s],$$

and γ is a discount factor [8]. Markov processes are used to define the environment and Markov reward processes allow the definition of an objective.

The final component necessary is the ability of the agent to make decisions. MRP $(\mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma)$ can be extended by adding actions to define the Markov decision process. Thus, *finite discounted Markov decision process* is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions, $\mathcal{P}$ is a state transition matrix, now also dependent on action a:

$$\mathcal{P}^a_{ss'} = \mathrm{P}(S_{t+1} = s' \mid S_t = s, A_t = a),$$

$\mathcal{R}$ is a reward function dependent on action a as well, $\mathcal{R} : (\mathcal{S}, \mathcal{A}) \to \mathbb{R}$,

$$\mathcal{R}^a_s = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$$

and γ is the discount factor [8].

This mathematical framework describes the interaction between the agent and the environment depicted in Figure 1.1. An agent carries out an action and the state of the environment changes. Note that this state transition is stochastic, meaning the outcome may vary. The environment also produces reward signal and the objective of the agent will be to maximize the total expected return. Whether the agent is willing to accept smaller short-term rewards in order to maximize long-term return or prioritizes higher immediate gains depends on discount factor γ.

▶ **Example 1.1.** Throughout this thesis, a very simple examples will be used to help obtain a better intuition of the presented methods.

Consider a very simple touchscreen interface consisting of three screens — red, green and blue. Each screen has two buttons, white and black. Interacting with the white button changes the screen, while interacting with the black button stays on the current screen. See Figure 1.3.

This environment can be modelled very simply using an MDP. Let $\mathcal{S} = \{\text{red, green, blue}\}$ be the set of states and $\mathcal{A} = \{\text{white}, \text{black}\}$ a set of actions. The action-state transitions are deterministic, so the $\mathcal{P}$ matrix for pressing the white button will be

$$\mathcal{P}^{\text{white}} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

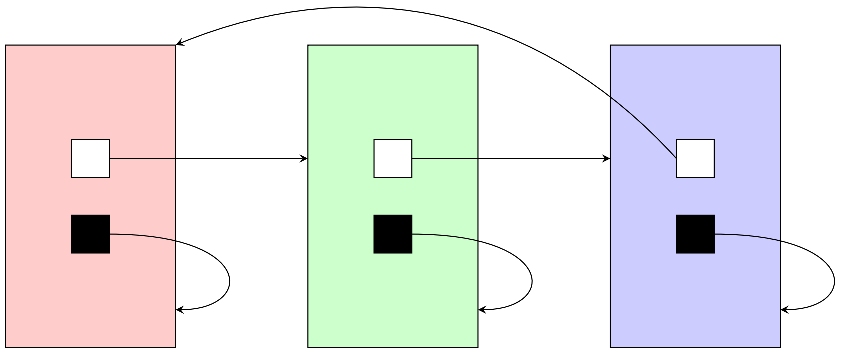


Figure 1.3 A simple example of a real-life environment.

and for pressing the black button,

$$\mathcal{P}^{\text{black}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Note that until a reward is defined, there is no objective. The $\mathcal{R}$ function can reward actions that transition between states (i.e. pressing the white button), staying in the same state (i.e. pressing the black button) or any other behaviour. It is also important to note that the reward function is not assumed to be deterministic.

This minimal, yet illustrative example will be used in the following sections to showcase the discussed methods.

1.3 Value Function, Bellman Equations

Let $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ be a discounted finite MDP, where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions, $\mathcal{P}$ is a state transition matrix, $\mathcal{R}$ is a reward function and γ is the discount factor. To distinguish different agents, a policy mapping π will be used. The policy can be defined deterministically, as function $\pi : \mathcal{S} \to \mathcal{A}$, but more general approach is to define it stochastically. π describes the probability of taking action $a \in \mathcal{A}$ when in state s:

$$\pi(a \mid s) = \mathrm{P}(A_t = a \mid S_t = s) \text{ [5], [8].}$$

Each agent has given policy that is used to determine the next action.

Since policy is independent of state transition probabilities, it is possible to determine the probability of agent following policy π in state s moving to state s':

$$\mathcal{P}^{\pi}_{ss'} = \sum_{a \in \mathcal{A}} \pi(a \mid s) \mathcal{P}^{a}_{ss'}. \text{ [8]}$$

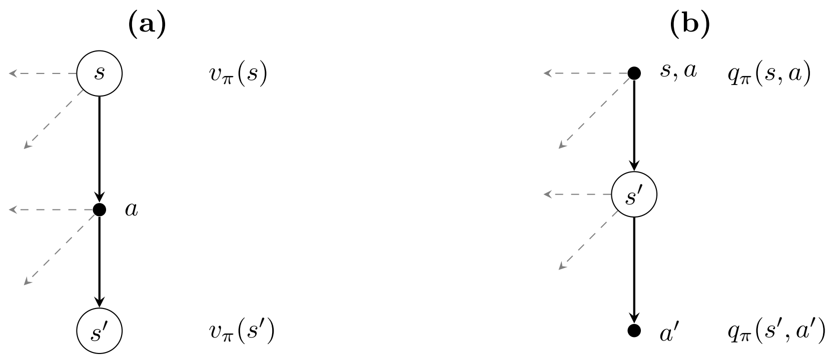


Figure 1.4 Diagram illustrating the recursive nature of (a) $v_\pi(s)$ and (b) $q_\pi(s,a)$. Based on [5].

Similarly, the expected reward of an agent following policy π in state s can be determined as:

$$\mathcal{R}_s^\pi = \sum_{a\in\mathcal{A}} \pi(a \mid s)\mathcal{R}_s^a \text{ [8].}$$

The reward R_t at time step t is indeterministically given by the previous state s and the action a the agent performed. This makes it better for some states to be in than others — an agent should prefer the states that promise greater discounted return G_t in the future. A state-value function is used to determine this quantity:

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s],$$

where $\mathbb{E}_\pi$ is the expected value if an agent is on policy π [5].

Similarly, an action-value function is defined. Intuitively, if state-value function $v_\pi(s)$ for an agent following policy π describes the expected value of discounted return G_t for an agent in state s, action-value function goes one step forward and describes expected discounted return G_t after taking an action a from step s. Formally:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] \text{ [5].}$$

These two functions are the cornerstone of reinforcement learning. As Figure 1.4 suggests, it is possible to express state-value function recursively:

$$v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a \mid s) \left[\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s')\right].$$

Similarly, an action-value function may be expressed recursively as follows:

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a'\in\mathcal{A}} \pi(a' \mid s') q_\pi(s',a').$$

These are the Bellman expectation equations [8].

A definition of a preorder on policies (here, sources [8] and [5] say partial order, but antisymmetry does not hold) allow us not only to compare two agents on different policies, but, more importantly, to define an optimal policy. Let

$$\pi \geq \pi' \iff \forall s \in \mathcal{S} : v_\pi(s) \geq v_{\pi'}(s).$$

An optimal policy π_* fulfills $\forall \pi : \pi_* \geq \pi$. Solving a reinforcement learning task means finding this optimal policy π_* [8]. An optimal policy also defines the optimal state-value function

$$v_*(s) = \max_\pi v_\pi(s)$$

and the optimal action-value function

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] = \max_\pi q_\pi(s,a) \text{ [5]}.$$

The optimal action-value function immediately gives the optimal policy. Intuitively, the agent always selects the action that maximizes $q_*(s,a)$, formally

$$\pi_*(a \mid s) = \begin{cases} 1, & \text{if } a = \arg\max\limits_{a \in \mathcal{A}} q_*(s,a) \\ 0, & \text{otherwise.} \end{cases} \quad \text{[8]} \tag{1.1}$$

It can be shown that for MDP as it is defined here, a deterministic optimal policy exists. Proof of this can be found in [10].

The optimal state-value function must fulfill

$$v_*(s) = \max_{a \in \mathcal{A}} q_*(s,a)$$

and considering $S_{t+1} = s'$, the optimal action-value function can be expressed similarly as

$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s').$$

By substituting $q_*(s,a)$ in the first equation, the following recursive equation is obtained:

$$v_*(s) = \max_{a \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$

Similarly, $v_*(s')$ can be substituted in the second equation to obtain

$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a \in \mathcal{A}} q_*(s',a).$$

These two equations are called *Bellman optimality equations*, sometimes referred to as just Bellman equations. Solving these equations allows us to construct the optimal policy, as shown in equation (1.1), which, in the context of reinforcement learning, means finding an optimal agent. While the equations do not have closed form solution in general, many iterative solution methods exist [8]. For an idea of how a policy of an MDP can be optimized, the next section will introduce two of the most common methods.

▶ **Example 1.2.** Using the environment introduced in Example 1.1, consider the following reward function:

$$\mathcal{R}_s^a = \begin{cases} 1, & \text{if screen changed} \\ 0, & \text{otherwise.} \end{cases}$$

It is clear that the optimal policy π_* will be to always interact with the white button. If $\gamma = 0.66$, the state-value function of each state (because the environment is symmetric) using this optimal policy will be

$$\forall s \in \mathcal{S} : v_{\pi_*}(s) = \sum_{k=0}^{\infty} \gamma^k \cdot 1 = 1 + 0.66 + 0.66^2 + \cdots \approx 2.9412,$$

because the return is deterministic. If a different policy was used, its state-value function would not be greater than that of π_*. Consider a policy π' that starts interacting with white button and then alternates between the two possible actions. The state-value will be

$$\forall s \in \mathcal{S} : v_{\pi'}(s) = \sum_{k=0}^{\infty} \gamma^{2k} \cdot 1 = 1 + 0 + 0.66^2 + \cdots \approx 1.7717,$$

so the policy π' is not optimal.

1.4 Solving MDPs with Dynamic Programming

Dynamic programming (DP) is an algorithmic paradigm that breaks down a problem into overlapping subproblems and stores results of these subproblems into a lookup table for future reference. The key step is to use a recursive or iterative formulation of the problem. Dynamic programming is typically applied to optimization problems. This section briefly highlights the use of dynamic programming methods to find the MDP's optimal policy π_* [11].

A policy's performance can be evaluated by finding its state-value function. This is necessary to compare two policies. Recall that $\pi \geq \pi' \iff \forall s \in \mathcal{S} : v_\pi(s) \geq v_{\pi'}(s)$, where $\geq$ is a preorder relation.

Let $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ be a discounted finite MDP, where $\mathcal{S}$ is a finite set of states with $n \in \mathbb{N}_0^+$ states denoted $s_1, \ldots, s_n$. Furthermore, $\mathcal{A}$ is a finite set of actions, $\mathcal{P}$ is a state transition matrix, $\mathcal{R}$ is a reward function and γ is the discount factor. It can be shown that the state-value function can be expressed using the Bellman expectation equation for any policy π:

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s) \left[\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right].$$

This equation can be expressed in matrix form. Let $\mathbf{v}_\pi \in \mathbb{R}^n$ be a vector where $(\mathbf{v}_\pi)_i = v_\pi(s_i)$. Let

$$(\mathbf{R}_\pi)_i = \sum_{a \in \mathcal{A}} \pi(a \mid s_i) \mathcal{R}_{s_i}^a$$

and

$$(\mathbf{P}_\pi)_{ij} = \sum_{a \in \mathcal{A}} \pi(a \mid s_i) \mathcal{P}^a_{s_i s_j},$$

then $\mathbf{v}_\pi$ can be expressed as

$$\mathbf{v}_\pi = \mathbf{R}_\pi + \gamma \mathbf{P}_\pi \mathbf{v}_\pi.$$

This equation has a closed form solution, provided $(\mathbf{I} - \gamma \mathbf{P}_\pi)$ is invertible:

$$\mathbf{v}_\pi = (\mathbf{I} - \gamma \mathbf{P}_\pi)^{-1} \mathbf{R}_\pi \text{ [8]}.$$

The first approach to solving the MDPs that will be highlighted here is referred to as *policy iteration* and was invented in 1960 by Ronald A. Howard [12]. It is an iterative algorithm, where each iteration consists of two steps:

1. Policy evaluation
2. Policy improvement

One way to evaluate a policy was shown, but using the closed form solution is usually not very efficient. Policy evaluation is preferably performed through iterative application of Bellman expectation equation. In each time step k, state-value function estimate is improved using

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s) \left[\mathcal{R}^a_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} v_k(s') \right], \tag{1.2}$$

or in matrix form

$$\mathbf{v}^{k+1}_\pi = \mathbf{R}_\pi + \gamma \mathbf{P}_\pi \mathbf{v}^k_\pi.$$

It can be shown that $\lim_{k \to \infty} v_k = v_\pi$, i.e. the estimate converges to the real state-value function [8].

Let π_k be an arbitrary policy. The approach described above can be used to compute the value function v_{π_k}. The policy improvement step is simple and intuitive: a new policy is obtained by acting greedily with respect to state-value function v_{π_k}, which can also be expressed in terms of the action-value function:

$$\pi_{k+1} = \arg\max_{a \in \mathcal{A}} q_{\pi_k}(s, a).$$

This greedy update guarantees that the new policy is at least as good as the previous one, i.e. $\pi_{k+1} \geq \pi_k$. Recall that a deterministic optimal policy exists. It can also be shown that by repeatedly applying policy evaluation followed by policy improvement, the sequence of policies converges to the optimal policy π_* [8]. This result is remarkable, particularly because the method is limited to deterministic policies and relies on straightforward greedy improvement step.

The policy evaluation step, in each of its iterations, updates the intermediate state-value function for all states $s \in \mathcal{S}$. This operation is called *full* *backup* — a backup corresponds to a single update of the intermediate state-value function [5].

Now instead of evaluating the full policy π, after just this one backup, an algorithm can skip directly to the greedy action selection, which is the policy improvement step. So in each step, the algorithm updates the value function by using the following rule:

$$v_{k+1}(s) = \max_{a \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s').$$

This equation is in fact the Bellman optimality equation and this new algorithm is called the *value iteration*, introduced by Richard Bellman in 1957 [9], and just like the policy iteration, the algorithm converges to an optimal policy for discounted finite MDPs [5].

▶ **Example 1.3.** Consider the environment introduced in Example 1.1, the alternating policy π' from Example 1.2 and $\gamma = 0.66$. Setting the arbitrary policy $v_0(s) = 1$ for all states, the policy for each state can be evaluated using the equation (1.2):

$$\begin{aligned} v_0(s) &= 1 \\ v_1(s) &= 0 + \gamma v_0(s) = 0.66 \\ v_2(s) &= 1 + \gamma v_1(s) = 1.436 \\ v_3(s) &= 0 + \gamma v_2(s) = 0.948 \\ v_4(s) &= 1 + \gamma v_3(s) = 1.625 \\ &\dots \\ v_{100}(s) &= 1.772 \approx v_{\pi'}(s). \end{aligned}$$

The value corresponds with the result from Example 1.2. To use the calculated policy in the policy improvement step, the new policy will act greedily with respect to it, i.e. will select the action that maximizes the action-value function

$$q_{\pi'}(s, a) = \mathcal{R}_s^a + \gamma v_{\pi'}(s')$$

where $\mathcal{R}_s^a$ is the reward for taking action a in state s and s' is the next state. For each state,

$$\begin{aligned} q_{\pi'}(s, \text{white}) &= 1 + \gamma v_{\pi'}(s') = 2.170 \\ q_{\pi'}(s, \text{black}) &= 0 + \gamma v_{\pi'}(s) = 1.170 \end{aligned}$$

So the new greedy policy will select white button on each screen, which is the optimal policy.

Policy iteration and value iteration assume full knowledge of the MDP of the system. In reinforcement learning problems, however, this is not usually the case. In the following sections, algorithms that do not require full knowledge of the MDP, the *model-free* methods, will be discussed.

1.5 Model-Free Value-Based Methods

This thesis focuses on navigating a complex user interface. If the corresponding MDP of the interface were known, solving the task would be straightforward. However, since the MDP is unknown, the methods of finding optimal policy discussed in the previous section cannot be used. In such situations, model-free methods offer a viable solution.

1.5.1 Monte-Carlo Methods

First, consider a system with an unknown underlying MDP. An *episode* will be a sequence of sampled states, actions and rewards from actual or simulated interaction with the system under policy π, $(S_0, A_0, R_1, S_1, A_1, \dots, S_k)$. The state-value function is the expected value of discounted return

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s].$$

To find an estimate of the state-value function, sample mean of discounted returns

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$$

over multiple episodes can be used. This approach is an instance of the *Monte-Carlo method* [13].

First of the two variants of the Monte-Carlo approach is called the *every-visit Monte-Carlo*. Its idea is to iterate through the episode and for every encountered state, compute their actual discounted return until the end of the episode and the mean of all these values will be the estimate of the state-value function for this state. If rolling mean formula is used

$$\overline{x}_k = \overline{x}_{k-1} + \frac{1}{k}(x_k - \overline{x}_{k-1}),$$

the Monte-Carlo estimate can be expressed as an incremental update at time step t to the state-value function estimate V as

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t)), \tag{1.3}$$

where α is a small positive constant (learning rate) [13]. This can be used in a dynamic system, where value function is not constant as older episodes will have less importance.

The other variant is called *first-visit Monte-Carlo*. It is different than every-visit in that once a state was visited in an episode, its estimate is not further updated until the end of the episode. The practical results and speculations in [14] suggest that this version is superior to the every-visit Monte-Carlo in terms of convergence rate, but there is no theoretical evidence of this claim.

1.5.2 Temporal Difference (TD) Methods

Another approach to the state-value function estimate, considered one of the most significant ideas in the field of reinforcement learning [13], is *temporal difference learning*, introduced by Richard S. Sutton first in 1984 in his dissertation [15] and later in 1988 in an article [16]. Building on the Monte-Carlo update rule in equation (1.3), the idea of temporal difference (TD) is to use a Bellman expectancy equation to estimate the discounted return:

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

This can be viewed as a shift of the estimate $V(S_t)$ towards the *TD target*

$$R_{t+1} + \gamma V(S_{t+1}),$$

and the magnitude of the shift is determined by the *TD error*

$$\delta_t = (R_{t+1} + \gamma V(S_{t+1})) - V(S_t). \text{ [13]}$$

The DP, Monte-Carlo and TD methods can be compared. The obvious shortcoming of DP methods is that they require the knowledge of full MDP to work. It was mentioned that for a given policy π, DP methods converge to the actual state-value function v_π. It can also be shown that both first-visit and every-visit Monte-Carlo converge [14]. TD also converges — almost surely if α is sufficiently small, and with probability one if α decreases over time [5]. The TD and DP methods, unlike Monte-Carlo, use *bootstrapping*, which in this context means that the intermediate estimates are used to update newer estimates. This allows TD to update the estimates continuously, even if the episode hasn't finished yet, which for some applications may be a large advantage over the Monte-Carlo. The speed of convergence depends heavily on the task. Temporal difference exhibits lower variance and usually converges faster in practice than constant-α Mote-Carlo. But there are also examples where Monte-Carlo can converge faster [5].

1.5.3 Policy Improvement in Model-Free Methods

In model-free systems, it is not possible to construct the optimal policy by simply greedily selecting the action that maximizes the estimated value-function. This gives rise to the *exploration–exploitation dilemma*. If the MDP is known, there is essentially no need for exploration, as all the necessary information about the environment is available and pure exploitation can be performed, i.e. the action with the highest expected return can be selected greedily. However, because the underlying MDP is usually unknown, it becomes necessary to spend time exploring the environment. The time spent exploring must be carefully balanced, as excessive exploration may lead to problems, such as slower convergence. A common strategy is to explore more in the initial phases of training and gradually start exploiting the knowledge in the later phases.

The simplest idea is probably using the ε-greedy policy: with probability $1-\varepsilon$, pick the greedy action that maximizes the estimated value function; with probability ε, pick an action uniformly at random. The usual method is using the ε-greedy improvement after a policy evaluation step (Monte-Carlo or TD). The sequence of ε-greedy policies can then converge to the optimal policy, if ε is chosen correctly [8].

A policy is said to be *greedy in the limit with infinite exploration* (GLIE), if it satisfies following two conditions:

1. Every state-action pair is visited infinitely many times.
2. The learning policy converges to a greedy policy in the limit.

An example of such policy is an ε-greedy policy with $\varepsilon_k = \frac{1}{k}$, where k is the iterator of policy updates [8]. GLIE policies will play important role in finding the optimal policy.

1.5.4 SARSA and Q-learning

Recall that the temporal difference approach evaluates the value-function of a policy using the update rule,

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

In model-free methods, the model of environment's dynamics is unknown and the transitions from a state are unknown as well. Instead of learning the value-function of a state, an estimate of action-value Q may be learned for every state-action pair:

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

By using an ε-greedy policy maximizing the estimated action-value, the policy can be continually improved through a process similar to policy iteration as described in Figure 1.5. This algorithm was originally introduced as Modified Connectionist Q-Learning in [17] in 1994. A footnote in the article mentions: *"Though Rich Sutton suggests SARSA, as you need to know State-Action-Reward-State-Action before performing an update"* [17]. Since then, the name SARSA has become standard in the literature.

If step-sizes α_t form a Robbins-Monro sequence, i.e.

$$\sum_{t=0}^{\infty} \alpha_t = \infty$$
$$\sum_{t=0}^{\infty} \alpha_t^2 < \infty$$

and a GLIE policy is used, then SARSA algorithm converges with probability one to the optimal action-value function, and thus yields an optimal policy [18].

Algorithm 1 SARSA

```
1: Initialize $Q(S, A)$ arbitrarily for all $S \in \mathcal{S}, A \in \mathcal{A}$
2: for each episode do
3:     Initialize state $S$ and choose action $A$ using policy derived from $Q$
4:     while $S$ is not terminal do
5:         Take $A$, observe $R, S'$
6:         Choose $A'$ from $S'$ using policy derived from $Q$
7:         $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
8:         $S \leftarrow S'$, $A \leftarrow A'$
9:     end while
10: end for
```

Figure 1.5 Pseudocode of the SARSA algorithm, paraphrased from [5].

SARSA is an *on-policy* learning algorithm, which means that it needs to sample episodes using the policy that is being optimized to estimate its action-value function. There exists an *off-policy* TD alternative, called *Q-learning*, which can learn on episodes sampled using a different policy. Q-learning was introduced already in 1989 by Christopher J. C. H. Watkins in his dissertation [19] and the proof to its convergence theorem was presented in 1992 in [20].

The update rule of its simplest form, the *one-step* Q-learning, is derived from the Bellman optimality equation:

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

Since Bellman optimality equation is used, $Q(S_t, A_t)$ is now updated toward the action-value function of an optimal policy [5]. In each state-action pair, the update is done greedily using the action a that maximizes the estimated action-value, even though in the next time step, the algorithm continues with the state-action pair that was experienced, which is not necessarily the one with the greedy action a.

As mentioned, the algorithm operates off-policy. The policy used to generate the state-action transitions is referred to as *behaviour policy* and the policy being optimized is *target policy*. The two processes of generating samples and updating value function estimate are separate. The Q-learning algorithm makes it possible to use an arbitrary behaviour policy, given that all state-action pairs are updated infinitely often in the limit. Under this assumption and the standard stochastic approximation conditions on the step size, it has been shown that Q converges to q_* [5].

In a closed-loop situation, where the agent makes decisions based on observed feedback, an ϵ-greedy policy is still commonly used as the behaviour

Algorithm 2 Q-learning

```
1: Initialize $Q(S, A)$ arbitrarily for all $S \in \mathcal{S}, A \in \mathcal{A}$
2: for each episode do
3:     Initialize state $S$
4:     while $S$ is not terminal do
5:         Choose action $A$ from $S$ using behaviour policy
6:         Take $A$, observe $R, S'$
7:         $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$
8:         $S \leftarrow S'$
9:     end while
10: end for
```

Figure 1.6 Pseudocode of the Q-learning algorithm, paraphrased from [5].

policy [13]. Pseudocode for Q-learning can be seen in Figure 1.6. The algorithm is sometimes referred to as SARSAMAX [8].

1.5.5 Eligibility Traces

This section describes an important concept in temporal difference methods that will be briefly introduced here. It was introduced in Richard S. Sutton's 1984 PhD thesis [15], and a thorough description can be found in his and Andrew G. Barto's book [5].

First, the algorithm will be allowed to use information from multiple steps ahead. In the standard TD update rule, the estimate is updated towards TD target

$$R_{t+1} + \gamma V(S_{t+1}).$$

The update uses the immediate reward R_{t+1} and the estimate of future returns represented by the estimated state-value function $V(S_{t+1})$, i.e. bootstrapping from S_{t+1}. By allowing TD to look further ahead, it could use multiple immediate rewards and bootstrap from a later state. For example, with a two-step look ahead, the TD target becomes

$$R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2}).$$

With infinite steps look-ahead, the TD target becomes the actual discounted return value, so the temporal difference method becomes equivalent to Monte-Carlo [5].

The TD target can thus be expressed as *n-step return*, defined as

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{n-1} R_h + \gamma^n V(S_{t+n}),$$ [8]

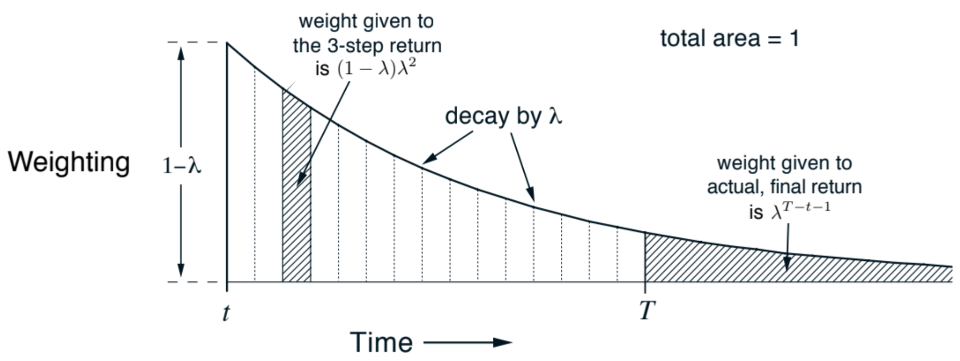


Figure 1.7 The behaviour of weighting in each n-step return in the λ-return. Reproduced from [5].

where $h = t + n$ is called the *horizon* of the n-step return. This extension of TD will then be called *n-step temporal difference learning* and the update step will be

$$V(S_t) \leftarrow V(S_t) + \alpha \left[G_t^{(n)} - V(S_t) \right].$$

If the episode concludes before the horizon is reached, the sequence is truncated at the end of an episode and the n-step return effectively contains the actual return [5].

Although n-steps TD methods are conceptually helpful, in practice, they are not very convenient to implement [5]. Instead, it is common to use the average of n-step returns for all n. Let $\lambda \in \mathbb{R}, 0 \leq \lambda \leq 1$. The *$\lambda$-return* can be expressed as

$$G_t^{\lambda} = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}.$$

Using the G_t^{λ} as the TD target, the *forward-view TD*(λ) algorithm is obtained [8]. As illustrated in Figure 1.7, lower value of parameter λ increases the rate of decay, so more weight is given to the lower n.

The forward-view approach to TD(λ) is conceptually straightforward, but it cannot be directly implemented in an online setting, i.e. when updates are made during an episode [5]. An offline implementation would be possible, but it would sacrifice one of the primary advantages of TD methods over Monte-Carlo.

In the TD(λ) algorithm, an additional variable called *eligibility trace* will be associated with each state. Eligibility trace $E_t(s) \in \mathbb{R}^+$ of state s decays at each time step by parameter λ introduced above and the discount factor γ

$$E_t(s) = \gamma \lambda E_{t-1}(s),$$

and whenever the state s is visited, its eligibility trace is incremented by one.

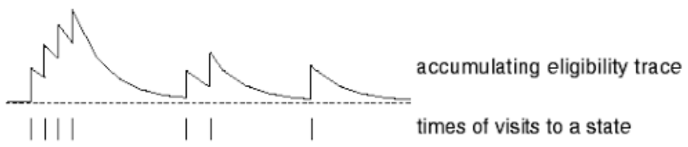


Figure 1.8 An illustration of accumulating eligibility trace behaviour. Reproduced from [5].

Algorithm 3 Backward-view TD(λ)

```
1:  Initialize $V(S)$ arbitrarily for all $S \in \mathcal{S}$
2:  for each episode do
3:      Initialize $E(s) = 0$ for all $s \in \mathcal{S}$
4:      Initialize $S$
5:      while $S$ is not terminal do
6:          Choose action $A$ using the evaluated policy
7:          Take $A$, observe $R, S'$
8:          $\delta \leftarrow (R + \gamma V(S')) - V(S)$
9:          $E(S) \leftarrow E(S) + 1$                    ▷ accumulating trace
10:         for all $s \in \mathcal{S}$ do
11:             $V(s) \leftarrow V(s) + \alpha\delta E(s)$
12:             $E(s) \leftarrow \gamma\lambda E(s)$
13:         end for
14:         $S \leftarrow S'$
15:     end while
16: end for
```

Figure 1.9 Pseudocode of the online TD(λ) algorithm, paraphrased from [5].

Generally,

$$E_0(s) = 0, \qquad E_t(s) = \gamma\lambda E_{t-1}(s) + \mathbf{1}_{\{S_t=s\}}$$

where S_t is the state visited at time step t and $\mathbf{1}_{\{S_t=s\}}$ is an indicator function, which equals one if $S_t = s$, or zero otherwise [8]. Illustration of how accumulating eligibility trace evolves over time can be seen in Figure 1.8. There are other strategies for updating eligibility traces, such as the *replacing trace*, which sets the eligibility trace value to one instead of incrementing it, or *dutch trace*, which is sort of intermediate between replacing trace and accumulating trace [5].

The *backward-view TD*(λ) algorithm calculates the TD error in each step,

$$\delta_t = (R_{t+1} + \gamma V(S_{t+1})) - V(S_t),$$

and updates the state-value function estimate for each state s,

$$V(s) \leftarrow V(s) + \alpha\delta_t E_t(s). \text{ [8]} \tag{1.4}$$

The update of states in equation (1.4) in fact happens only in recently visited states, as for other states the value of eligibility trace will be zero. A possible optimization is to keep track of recently visited states that are to be updated. The value of parameter λ determines what "recently visited" means. The backward view allow us to construct an online algorithm. A pseudocode for this is illustrated in Figure 1.9.

It can be seen that TD(0) is the algorithm introduced before as TD (only $n = 1$ step is used), while offline TD(1) is equivalent to the constant-α Monte-Carlo [8].

Since SARSA is a straightforward extension of TD, a *SARSA*(λ) can be constructed in the same way as TD(λ). The intuition behind forward-view SARSA(λ) is very similar. The *n-step Q-return* is defined as

$$q_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{n-1} R_h + \gamma^n Q(S_{t+n}, A_{t+n}),$$

where $h = t + n$ is again the horizon of the *n-step Q-return*. By using the exponentially weighted average over all n with decay parameter λ, a q^λ return is obtained,

$$q_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} q_t^{(n)}.$$

Finally, using these the forward-view SARSA(λ) is obtained,

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(q_t^\lambda - Q(S_t, A_t) \right). \text{ [8]}$$

In its backward-view variant, eligibility trace for each state-action pair,

$$E_0(s, a) = 0, \qquad E_t(s, a) = \gamma \lambda E_{t-1}(s, a) + \mathbf{1}_{\{S_t = s, A_t = a\}},$$

is used to construct the online algorithm. TD error is calculated in the same way as in TD(λ), except that instead of state-value function, an action-value function is being shifted,

$$\delta_t = (R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})) - Q(S_t, A_t),$$

and an action-value function for each (recently visited) state-action pair s, a is updated,

$$Q(s, a) \leftarrow Q(s, a) + \alpha \delta_t E_t(s, a). \text{ [8]}$$

It is possible to combine Q-learning with eligibility traces in a straightforward way, which was shown already in 1989 by Christopher J. C. H. Watkins when Q-learning was introduced [19]. The algorithm is known as *Watkins' Q*(λ). In the backward view, it operates in the same way as SARSA(λ), but whenever a non-greedy action is taken, all eligibility traces are cut off and set to zero. This can reduce the benefits of eligibility traces, especially in the initial phases of training when exploratory actions are taken more often [5].

1.6 Function Approximation in RL

All the methods introduced so far are known as *tabular methods*, which require maintaining an explicit table of estimates of value function for every state (or state-action pair). For control of an agent in model-free environment, an action-value function estimate must be stored for every possible state-action pair.

For example, the number of possible legal positions in chess is estimated to be approximately $4.8 \cdot 10^{44}$ [21], making it infeasible to maintain a separate estimate for each state. The problem is not just a large memory requirement to store the table, but also the time necessary to accurately fill it [5]. In this work, state may represent a screen (e.g., a display screenshot) in possibly a complex environment consisting of large amount of screens. An alternative approach might be more suitable.

The straightforward way is to use a function that approximates the real state-value function,

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s),$$

or action-value function,

$$\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a),$$

for given policy π [5].

Using function approximation, it is no longer necessary for each state or state-action pair to be visited. The concept of *generalization* plays a crucial role here. It is necessary to approximate the value function so that the approximation works well even for states that were never visited [5].

There exists wide variety of methods for function approximation, e.g. linear regression, neural networks, proximity-based methods and many more. Recent works (e.g. [22], [23]) have shown great promise in differentiable methods and neural networks.

There are also requirements on the training method, namely that it must handle *non-stationary* targets, as the value estimates may change during training, and that it should be suitable for *non-iid* data, i.e. it should be able to handle steps that are dependent on each other [8].

First, suppose that $v_\pi(s)$ is a known function and $\hat{v}(s, \mathbf{w})$ will be its approximation. The theoretical objective can be expressed as minimization of *root-mean-squared error (RMSE)*

$$\mathrm{RMSE}(\mathbf{w}) = \sqrt{\sum_{s \in \mathcal{S}} d(s) \left[v_\pi(s) - \hat{v}(s, \mathbf{w})\right]^2},$$

where $d(s) \in [0, 1], \sum_s d(s) = 1$ is a distribution over the states that specifies relative importance of error in each state. In order to gain better approximation at some states, approximation in other states must become less important.

For example, states that are visited more often may have higher importance. In fact, the distribution d may match the distribution of sampled data, which will be assumed in the rest of this section [5].

1.6.1 Gradient Descent

A *gradient* of a function is a vector that contains all of its first-order partial derivatives with respect to its input variables. Let $f(\mathbf{w})$ be a differentiable function, where $\mathbf{w} = (w_1, w_2, \ldots, w_n)^T$. The gradient of f with respect to $\mathbf{w}$ is given by

$$\nabla_{\mathbf{w}} f(\mathbf{w}) = \left(\frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2}, \ldots, \frac{\partial f}{\partial w_n} \right)^T.$$

The gradient points in the direction of the steepest ascent, so *stochastic gradient descent* (SGD) iteratively updates the parameters $\mathbf{w}$ in the direction opposite to the gradient of the function.

In order to minimize the RMSE, parameters must be slightly adjusted in the opposite direction of its gradient. This leads to the following update rule:

$$\begin{aligned} \mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{1}{2}\alpha \nabla_{\mathbf{w}_t} [v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)]^2 \\ &= \mathbf{w}_t + \alpha [v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)] \nabla_{\mathbf{w}_t} \hat{v}(S_t, \mathbf{w}_t). \text{ [5]} \end{aligned}$$

The value $\frac{1}{2}$ is used as a part of loss function to simplify the gradient. Note that as long as $\hat{v}(S_t, \mathbf{w}_t)$ is differentiable, gradient-based optimization algorithms can be used to perform the update.

Since the assumption that $v_\pi(s)$ is known does not make much sense (otherwise the function itself would be a perfect approximation), consider using an estimate V_t of $v_\pi(S_t)$ as the target,

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha [V_t - \hat{v}(S_t, \mathbf{w}_t)] \nabla_{\mathbf{w}_t} \hat{v}(S_t, \mathbf{w}_t). \text{ [5]}$$

The methods discussed in previous sections can be used to form a state-value function approximation:

- Monte-Carlo: $V_t = G_t$
- TD(0): V_t is the TD target $R_{t+1} + \hat{v}(S_{t+1}, \mathbf{w}_t)$
- Forward-view TD(λ): V_t is the λ-return G_t^λ
- Backward-view TD(λ): the update is done using eligibility traces,

$$\begin{aligned} \mathbf{e}_0 = \mathbf{0}, \qquad \mathbf{e}_t &= \gamma\lambda \mathbf{e}_{t-1} + \nabla_{\mathbf{w}_t} \hat{v}(S_t, \mathbf{w}_t), \\ \delta_t &= R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t), \\ \mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha \delta_t \mathbf{e}_t. \text{ [5]} \end{aligned}$$

It can be shown that if V_t is an unbiased estimate and the standard stochastic approximation conditions on the step size are met, then $\mathbf{w}_t$ converges to a to a stationary point of the mean-squared error. This is true for Monte-Carlo, but with $\lambda < 1$, TD is not an unbiased estimate. However, bootstrapping methods are effective and offer other performance guarantees, making them still useful [5].

Similarly to model-free tabular methods, an action-value function must be evaluated to be able to make decisions between actions. So instead of state-value, the objective will shift to approximate action-value function $\hat{q}(s, a, \mathbf{w})$. Luckily, as with the tabular methods, the extension for on-policy methods is straightforward,

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha[Q_t - \hat{q}(S_t, A_t, \mathbf{w}_t)]\nabla_{\mathbf{w}_t}\hat{q}(S_t, A_t, \mathbf{w}_t), \text{ [5]}$$

and by using previously introduced methods for action-value function estimation, an approximation can be done in te following way:

- Monte-Carlo: Q_t is the return G_t
- TD(0): Q_t is the TD target $R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t)$
- Forward-view TD(λ): Q_t is the q^λ return q_t^λ
- Backward-view TD(λ): the update is done using eligibility traces,

$$\begin{aligned}\mathbf{e}_0 &= \mathbf{0}, \qquad \mathbf{e}_t = \gamma\lambda\mathbf{e}_{t-1} + \nabla_{\mathbf{w}_t}\hat{q}(S_t, A_t, \mathbf{w}_t),\\ \delta_t &= R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t),\\ \mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha\delta_t\mathbf{e}_t. \text{ [5]}\end{aligned}$$

Backward-view TD(λ) with a control mechanism, e.g. ε-greedy policy, is called *gradient-descent SARSA*(λ) [5].

Because of bias, temporal difference methods do not guarantee convergence for a non-linear function approximation. It might seem that this goes against the classical SGD results, but in fact, the TD methods are not instances of true gradient descent [24], which is why they are referred to as *semi-gradient methods.*

The extension of function approximation to the off-policy methods is even more concerning, as there is no convergence guarantee even for the linear function approximation. A famous example of divergence is the *Baird's counterexample* [25], introduced in 1995 by Leemon Baird. Richard S. Sutton and Andrew G. Barto discuss the *deadly triad* [5], a combination of three elements that may lead to a danger of instability and divergence:

- Function approximation
- Bootstrapping

- Off-policy training

According to them, when only two elements of the triad are in use, divergence and instability can be avoided. Note however, that there are highly successful algorithms, such as Deep Q-Network (DQN) [23], that use all three elements of the deadly triad. There are recent works that suggest possibility to mitigate the dangers of deadly triad [26].

1.7 Policy Gradient Methods

So far, finding a reinforcement learning agent was discussed as a task of estimating a value function and acting with respect to it, e.g. greedily. These are known as *value-based* methods. This section will focus on the other class, *policy-based* methods, that aim to parametrize the policy,

$$\pi(a \mid s, \boldsymbol{\theta}) = \mathrm{P}(A_t = a \mid S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}), \ [5]$$

where $\boldsymbol{\theta}$ parametrizes the distribution.

This section will focus on policies where it is possible to estimate a gradient of their performance measure. The objective will be to maximize this measure, so a *gradient ascent* method will be used. Its update rule looks as follows:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \widehat{\nabla_{\boldsymbol{\theta}_t} J(\boldsymbol{\theta}_t)}, \tag{1.5}$$

where $J(\boldsymbol{\theta}_t)$ is the performance measure and $\widehat{\nabla_{\boldsymbol{\theta}_t} J(\boldsymbol{\theta}_t)}$ is the estimate of its gradient [5].

It was in 1992 when Ronald J. Williams introduced the *REINFORCE* algorithm [27]. It is a Monte-Carlo algorithm, so in each iteration, it samples a complete *trajectory* (episode) $\tau = (S_0, A_0, R_1, S_1, A_1, \dots, A_{n-1}, R_n)$ from a space of all possible trajectories $\mathcal{T}$ using policy $\pi(a \mid s, \boldsymbol{\theta})$ and then, in each step of the episode, an update rule in the form of equation (1.5) will be applied.

Williams' original episodic REINFORCE update is

$$\Delta\boldsymbol{\theta}_t = \alpha(G_t - b(s))\nabla_{\boldsymbol{\theta}_t} \log \pi(A_t \mid S_t, \boldsymbol{\theta}_t),$$

where α is *learning rate* (unlike in the original, constant is assumed in this thesis), b is *reinforcement baseline*, which will be discussed in more detail later, for now, consider $b(s)$ to be uniformly zero, and $\nabla_{\boldsymbol{\theta}_t} \log \pi(A_t \mid S_t, \boldsymbol{\theta}_t)$ is called the *characteristic eligibility* [27]. Since the update rule is not straightforward, the following paragraph will explain the idea of how it is derived.

Consider use of the expected return from policy π as the performance measure,

$$J(\boldsymbol{\theta}) = \mathbb{E}_\pi[R(\tau)],$$

where $R(\tau)$ is the discounted return value of trajectory τ. This value can be expressed as an integral over all trajectories:

$$J(\boldsymbol{\theta}) = \int_{\mathcal{T}} R(\tau)\pi_{\boldsymbol{\theta}}(\tau)\ d\tau,$$

Algorithm 4 Episodic REINFORCE with baseline

```
1: Initialize $\boldsymbol{\theta}$ and $\mathbf{w}$ arbitrarily
2: for each episode do
3:     Generate an episode $(S_0, A_0, R_1, S_1, A_1, \ldots, A_{T-1}, R_T)$ using policy $\pi(a \mid s, \boldsymbol{\theta})$
4:     for each step $t$ in the episode do
5:         $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
6:         $\delta \leftarrow G - \hat{v}(S_t, \mathbf{w})$
7:         $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t \delta \nabla_{\boldsymbol{\theta}} \log \pi(A_t \mid S_t, \boldsymbol{\theta})$
8:         $\mathbf{w} \leftarrow \mathbf{w} + \beta \delta \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w})$
9:     end for
10: end for
```

Figure 1.10 Pseudocode of the REINFORCE algorithm with a learned state-value function as a baseline, adapted from [5].

where $\pi_{\boldsymbol{\theta}}(\tau)$ is a probability of the trajectory under policy $\pi_{\boldsymbol{\theta}}$. This value could also be expressed using separate steps of the trajectory, i.e. if $\rho_0(s)$ is the probability distribution of the initial state (a probability of $S_0 = s$),

$$S_0 \sim \rho_0(S_0), \quad A_t \sim \pi_{\boldsymbol{\theta}}(A_t \mid S_t, \boldsymbol{\theta}), \quad S_{t+1} \sim \mathrm{P}(S_{t+1} \mid S_t, A_t).$$

The simpler notation of probability of the whole trajectory, $\pi_{\boldsymbol{\theta}}(\tau)$, will be used. The goal is to find the gradient of the J function and since the trajectory space $\mathcal{T}$ is independent of $\boldsymbol{\theta}$, Leibniz rule applies and the gradient can be moved inside the integral. Applying the product rule for differentiation,

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \int_{\mathcal{T}} R(\tau) \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(\tau)\ d\tau.$$

Now using the “log-derivative trick”,

$$\nabla_{\boldsymbol{\theta}} p(x \mid \boldsymbol{\theta}) = p(x \mid \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} \log p(x \mid \boldsymbol{\theta}),$$

the gradient can be expressed as

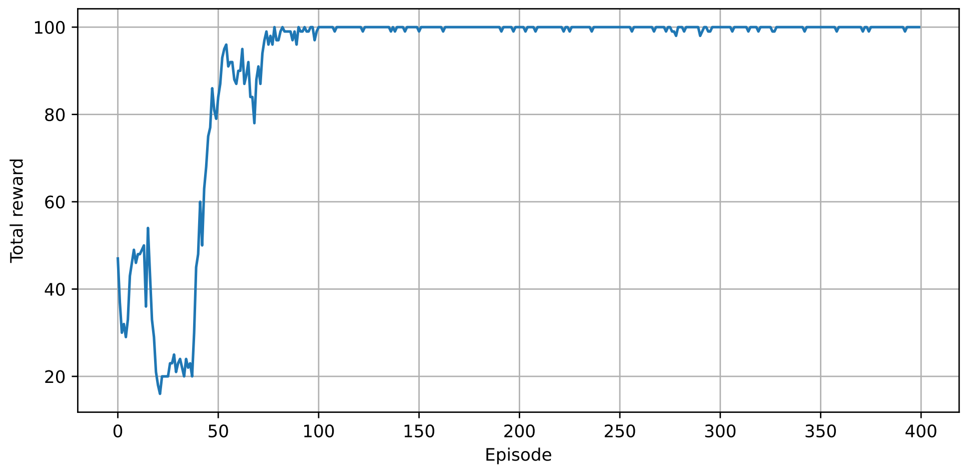
$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \int_{\mathcal{T}} R(\tau) \pi_{\boldsymbol{\theta}}(\tau) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\tau)\ d\tau,$$

which is an expression for the expected value

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi}[R(\tau) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\tau)]. \tag{1.6}$$

Finally, to obtain the Monte-Carlo estimate of this value, a sampled trajectory $\tau = (S_0, A_0, R_1, S_1, A_1, \ldots, A_{n-1}, R_n)$ and a discount factor γ will be used. The gradient estimate can then be computed for each time step $t \in 0, \ldots, n-1$:

$$\widehat{\nabla_{\boldsymbol{\theta}_t} J(\boldsymbol{\theta}_t)} = \gamma^t G_t \nabla_{\boldsymbol{\theta}_t} \log \pi_{\boldsymbol{\theta}_t}(A_t \mid S_t, \boldsymbol{\theta}_t),$$



■ **Figure 1.11** Example run of the REINFORCE algorithm without baseline.

where $G_t = \sum_{k=t+1}^{n} \gamma^{k-t-1} R_k$ [27]. This is the value that is used in the update rule of REINFORCE algorithm if baseline $b(s)$ is zero for all $s \in \mathcal{S}$.

The baseline function $b(s)$ is used to mitigate high variance. An example of such baseline is an approximation of the state-value function $\hat{v}(s, \mathbf{w})$ introduced in Section 1.6. The parameter vector $\mathbf{w}$ may be learned using gradient descent with Monte-Carlo estimate inside the REINFORCE algorithm's loop. This way, the action taken is measured against the average outcome from that state. A pseudocode of this can be seen in Figure 1.10. Using this baseline can make REINFORCE learn much faster [5].

▶ **Example 1.4.** REINFORCE algorithm can be used to find an optimal policy of the simple environment introduced in Example 1.1. An extremely simple approximation using parameters $\boldsymbol{\theta} = \big(\theta_0, \theta_1, \theta_2\big)$ will be used.

$$\pi(a \mid s, \boldsymbol{\theta}) = \sigma\left(\boldsymbol{\theta}^T \mathbf{s}\right),$$

where σ is the sigmoid function and $\mathbf{s}$ is a one-hot state encoding. Parameters will be initialized to zero, so that the initial action probabilities are uniform.

When using binary actions, it is possible to rewrite the probability model as Bernoulli distribution,

$$\pi(a \mid s, \boldsymbol{\theta}) = \sigma\big(\boldsymbol{\theta}^T \mathbf{s}\big)^a \big(1 - \sigma\big(\boldsymbol{\theta}^T \mathbf{s}\big)\big)^{1-a}.$$

Now taking the logarithm,

$$\log \pi(a \mid s, \boldsymbol{\theta}) = a \log \sigma\big(\boldsymbol{\theta}^T \mathbf{s}\big) + (1-a) \log\big(1 - \sigma\big(\boldsymbol{\theta}^T \mathbf{s}\big)\big),$$

by first taking the partial derivative of the approximation with respect to $x = \boldsymbol{\theta}^T \mathbf{s}$,

$$\frac{\partial \log \pi(a \mid s, \boldsymbol{\theta})}{\partial x} = a(1 - \sigma(x)) + (1-a)(-\sigma(x)) = a - \sigma(x),$$

because
$$\frac{\partial \log \sigma(x)}{\partial x} = 1 - \sigma(x) \text{ and } \frac{\partial \log(1-\sigma(x))}{\partial x} = -\sigma(x).$$
Then, after computing partial derivative of x with respect to the parameters $\boldsymbol{\theta}$,
$$\frac{\partial x}{\partial \boldsymbol{\theta}} = \frac{\partial \boldsymbol{\theta}^T \mathbf{s}}{\partial \boldsymbol{\theta}} = \mathbf{s},$$
a chain rule can be applied to obtain
$$\nabla_{\boldsymbol{\theta}} \log \pi(a \mid s, \boldsymbol{\theta}) = (a - \sigma(\boldsymbol{\theta}^T \mathbf{s}))\mathbf{s}.$$

And the update rule of REINFORCE without baseline for each timestep t will be
$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G_t (A_t - \sigma(\boldsymbol{\theta}^T \mathbf{s}_t))\mathbf{s}_t,$$

where $\mathbf{s}_t$ is the one-hot encoding of state S_t, and $\gamma = 0.66, \alpha = 0.1$ are used.

How the value of obtained reward per episode changes on the example can be seen in Figure 1.11. A small value of learning rate $\alpha = 0.003$ was used so that the learning process is more explicit in the illustration. Episodes of length 100 were sampled. The probability of selecting action white in each state approaches one.

1.7.1 Actor-Critic Methods

A framework of what is known today as *actor-critic* was introduced already in 1977 by Ian H. Witten [28]. A majority of early reinforcement learning algorithms fell into one of two categories:

1. *Actor-only* methods that use parametrized policy. An example would be the policy gradient methods, such as the REINFORCE algorithm without baseline. These alone tend to have a large variance.

2. *Critic-only* methods that try to find the approximation value function and the approximate solution to the Bellman equations. An example would be the SARSA algorithm introduced in Section 1.5.4.

Actor-critic methods aim to combine these approaches [29].

REINFORCE algorithm with baseline uses policy gradient to update its policy and value function approximation to stabilize the training, but the value function approximation is done before an action is taken so it is not used to “criticize” the policy gradient method. Actor-critic methods aim to do just that [5].

Algorithm 5 Episodic one-step actor-critic

```
1: Initialize $\boldsymbol{\theta}$ and $\mathbf{w}$ arbitrarily
2: for each episode do
3:     Initialize $S$
4:     $I \leftarrow 1$
5:     while $S$ is not terminal do
6:         Sample $A$ from $\pi(A \mid S, \boldsymbol{\theta})$
7:         Take $A$, observe $R, S'$
8:         $\delta \leftarrow R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$
9:         $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha I \delta \nabla_{\boldsymbol{\theta}} \log \pi(A \mid S, \boldsymbol{\theta})$
10:        $\mathbf{w} \leftarrow \mathbf{w} + \beta \delta \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w})$
11:        $S \leftarrow S'$
12:        $I \leftarrow \gamma I$
13:    end while
14: end for
```

Figure 1.12 Pseudocode of the one-step actor-critic algorithm, adapted from [5].

The transition from Monte-Carlo Policy Gradient (REINFORCE) to the simplest actor-critic can be done analogically to the transition from Monte-Carlo to TD in the value-based methods. Instead of using the discounted return G_t, its bootstrapped estimate will be used,

$$\delta = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}). \text{ [5]}$$

By comparing this to the δ of REINFORCE with baseline,

$$\delta = G_t - \hat{v}(S_t, \mathbf{w}),$$

it can be seen that the actor-critic's δ actually uses it's estimate to assess the action taken by the actor. Intuitively, if $R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$ yields lower value than the critic estimates, the parameters will be shifted in the negative direction of the gradient. On the other hand, if it is better than the estimate of the critic, parameters are shifted in the direction of the gradient. Using this approach, the algorithm becomes fully online. The pseudocode is in Figure 1.12. Note that the extension to actor-critic with action-value function approximation is straightforward.

An obvious extension to this is adding eligibility traces. Recall that by using them, the algorithm is allowed to look n steps ahead and use a weighted average of approximations for $n = 1, 2, \ldots, \infty$, where the parameter $\lambda \in [0, 1]$ is used — higher λ values result in a more uniform weighting across n, while lower λ values emphasize shorter-term returns (i.e. lower n). The implementation is straightforward and follows the implementation of backward-view approach

Algorithm 6 Episodic actor-critic with eligibility traces

```
1: Initialize $\boldsymbol{\theta}$ and $\mathbf{w}$ arbitrarily
2: for each episode do
3:     Initialize $S$
4:     $\mathbf{e}^{\boldsymbol{\theta}} \leftarrow \mathbf{0}, \mathbf{e}^{\mathbf{w}} \leftarrow \mathbf{0}$
5:     $I \leftarrow 1$
6:     while $S$ is not terminal do
7:         Sample $A$ from $\pi(A \mid S, \boldsymbol{\theta})$
8:         Take $A$, observe $R, S'$
9:         $\delta \leftarrow R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$
10:        $\mathbf{e}^{\boldsymbol{\theta}} \leftarrow \gamma \lambda^{\boldsymbol{\theta}} \mathbf{e}^{\boldsymbol{\theta}} + I \nabla_{\boldsymbol{\theta}} \log \pi(A \mid S, \boldsymbol{\theta})$
11:        $\mathbf{e}^{\mathbf{w}} \leftarrow \gamma \lambda^{\mathbf{w}} \mathbf{e}^{\mathbf{w}} + \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w})$
12:        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \delta \mathbf{e}^{\boldsymbol{\theta}}$
13:        $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \mathbf{e}^{\mathbf{w}}$
14:        $S \leftarrow S'$, $I \leftarrow \gamma I$
15:    end while
16: end for
```

Figure 1.13 Pseudocode of the actor-critic algorithm with eligibility traces, adapted from [5].

discussed in Section 1.5.5, its pseudocode is in Figure 1.13. $\lambda^{\boldsymbol{\theta}}, \lambda^{\mathbf{w}} \in [0,1]$ are trace-decay parameters, $\alpha^{\boldsymbol{\theta}}, \alpha^{\mathbf{w}}$ are learning rates and $\mathbf{e}^{\boldsymbol{\theta}}, \mathbf{e}^{\mathbf{w}}$ are eligibility trace vectors [5].

1.7.2 Policy Gradient Theorem

This section briefly presents the *policy gradient theorem* [30]. It was derived in Equation (1.6) that using the expected return of an episode as the performance measure, its gradient is

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi}[R(\tau) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\tau)].$$

Policy gradient theorem describes a general case, which states that for any differentiable policy and for usual objective functions, such as the average reward or the expected value function, the gradient is

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_\theta}[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s, \boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(s, a)] \text{ [8].} \tag{1.7}$$

When this was derived in Equation (1.6) for REINFORCE, a Monte-Carlo estimate of G_t was used as an unbiased estimate of the $q_{\pi_{\boldsymbol{\theta}}}(s, a)$. Actor-critic methods use a bootstrapped approximation of policy gradient, which may introduce bias and a biased policy gradient may not find the right solution.

By carefully choosing the approximation according to the *compatible function approximation theorem*, the method may work without introducing a bias. That is, the gradient is exactly as in Equation (1.7), if the following two conditions are satisfied:

1. Compatibility of value function aproximator and the policy, i.e.

$$\nabla_{\mathbf{w}} \hat{q}(s, a, \mathbf{w}) = \nabla_{\boldsymbol{\theta}} \log \pi(a \mid s, \boldsymbol{\theta})$$

2. Parameters $\mathbf{w}$ minimize MSE, i.e.

$$\varepsilon = \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left[\left(q_{\pi_{\boldsymbol{\theta}}}(s, a) - \hat{q}(s, a, \mathbf{w}) \right)^2 \right] \quad [8]$$

Readers further interested are referred to the original paper by Sutton et al. [30]. Note that even methods without formal guarantees may perform well in practice, as previously discussed.

Chapter 2

Advanced Methods in Reinforcement Learning

This chapter builds upon the foundations laid in Chapter One and introduces two important ideas in modern reinforcement learning: scalable learning and intrinsic motivation. The discussion on scalable learning gradually leads to Proximal Policy Optimization, which will be employed in the practical part of this thesis. The chapter then explores intrinsic motivation as a fundamental driver for the agent's natural exploration of the environment.

2.1 Natural Policy Gradient

The policy gradient discussed in Section 1.7 is known as *vanilla policy gradient*. As previously mentioned, it uses the stochastic gradient ascent algorithm to optimize the policy and the rule is

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}).$$

Parameter α controls the step size. This rule, however, is non-covariant — not only may not all θ_i have the same unit size, but also the left hand side of the update rule has different units than the right hand side. To be exact, each parameter on the left hand side has unit size of $[\theta_i]$, while the partial derivative of this parameter has unit size

$$\left[\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i}\right] = \frac{[J(\boldsymbol{\theta})]}{[\theta_i]} = \frac{1}{[\theta_i]}.$$

$J(\boldsymbol{\theta})$ is an expected return, which is unit-less [31].

Natural policy gradient uses underlying structure of the policy to update the parameters. To simplify the notation, suppose that $\pi_{\boldsymbol{\theta}}(a \mid s, \boldsymbol{\theta}) = \pi_{\boldsymbol{\theta}}(a \mid s)$, i.e. the parametrization will only be denoted in the subscript. The policy gradient theorem introduced in Section 1.7.2 says that the (vanilla) policy gradient is

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s) q_{\pi_{\boldsymbol{\theta}}}(s,a)],$$

which, given that policy π is *ergodic*, i.e. has a defined stationary distribution over states $\rho_{\pi_{\boldsymbol{\theta}}}$, can be rewritten as

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_{s\in\mathcal{S}} \rho_{\pi_{\boldsymbol{\theta}}}(s) \sum_{a\in\mathcal{A}} \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \mid s) q_{\pi_{\boldsymbol{\theta}}}(s,a).$$

Instead of the task of finding the direction of steepest ascent, Natural Policy Gradient method solves the task of finding max $J(\boldsymbol{\theta}+\Delta\boldsymbol{\theta})$, under the constraint that $||\Delta\boldsymbol{\theta}||^2_{\mathbf{G}} = \Delta\boldsymbol{\theta}^T \mathbf{G}(\boldsymbol{\theta})\Delta\boldsymbol{\theta} = \varepsilon$, where ε is small positive constant and $\mathbf{G}(\boldsymbol{\theta})$ is a matrix defining the metric $||\cdot||_{\mathbf{G}}$ (thus is positive-definite). The direction of steepest ascent is $\Delta\boldsymbol{\theta} \propto \mathbf{G}(\boldsymbol{\theta})^{-1}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ [31].

Using vanilla policy gradient, the steepest ascent that is followed is $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$, thus $\mathbf{G}(\boldsymbol{\theta}) = \mathbf{I}$ where $\mathbf{I}$ is identity matrix [31].

Natural policy gradients reparametrize the gradient using a different $\mathbf{G}$ matrix, so

$$\nabla^{\text{nat}}_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbf{G}(\boldsymbol{\theta})^{-1}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}).$$

A matrix used as $\mathbf{G}$ is the *Fisher information matrix*, defined per-state as

$$F_s(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s)^T]$$

and it *"is an invariant metric on the space of the parameters of probability distributions"* [31]. Suppose $\mathbb{E}_{s\sim\rho}[\cdot] = \mathbb{E}[\cdot \mid s \sim \rho]$, natural gradients use the metric

$$F(\boldsymbol{\theta}) = \mathbb{E}_{s\sim\rho_{\pi_{\boldsymbol{\theta}}}}[F_s(\boldsymbol{\theta})],$$

and the steepest ascent direction is

$$\nabla^{\text{nat}}_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = F(\boldsymbol{\theta})^{-1}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \text{ [31]}.$$

2.1.1 Natural Actor-Critic

An example of use of Natural Policy Gradient is *Natural Actor-Critic*, introduced in 2005 in [32]. An example of compatible value function approximation (more in Section 1.7.2), i.e. $\nabla_{\mathbf{w}} \hat{q}(s,a,\mathbf{w}) = \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s)$, is

$$\hat{q}(s,a,\mathbf{w}) = (\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s))^T \mathbf{w}.$$

The policy gradient, according to policy gradient theorem, will be

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s) q_{\pi_{\boldsymbol{\theta}}}(s,a)].$$

Using the compatible value function above as an approximation of $q_{\pi_{\boldsymbol{\theta}}}(s,a)$,

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s)(\nabla_{\boldsymbol{\theta}} \log \pi(a \mid s))^T \mathbf{w}] = \mathbf{G}(\boldsymbol{\theta})\mathbf{w},$$

so, because the critic is compatible and the expectation equals the Fisher matrix, the natural policy gradient is

$$\nabla_{\boldsymbol{\theta}}^{\text{nat}} J(\boldsymbol{\theta}) = \mathbf{w}.$$

And finally, the update step of natural actor-critic is

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \mathbf{w} \text{ [32]}.$$

The beauty of this approach is that the Fisher information matrix does not need to be explicitly computed or inverted to use the advantages of natural policy gradients.

2.2 Trust Region Policy Optimization (TRPO)

Trust region policy optimization (TRPO) [33] is a relatively recent approach to policy optimization. It borrows the minorize-maximise (MM) [34] intuition and works in a similar way. The objective is to find parametrization $\boldsymbol{\theta}$ that maximizes the expected return, i.e.

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[G_t],$$

Suppose now that $\boldsymbol{\theta}_{\text{old}}$ is the previous policy parameter that the update is relative to. MM algorithm uses a *surrogate function* $g(\boldsymbol{\theta} \mid \boldsymbol{\theta}_{\text{old}})$, which is said to *minorize* function J if

$$\forall \boldsymbol{\theta} : g(\boldsymbol{\theta} \mid \boldsymbol{\theta}_{\text{old}}) \leq J(\boldsymbol{\theta}) \qquad \text{and} \qquad g(\boldsymbol{\theta}_{\text{old}} \mid \boldsymbol{\theta}_{\text{old}}) = J(\boldsymbol{\theta}_{\text{old}}).$$

Finding a minorization of a function is the first step of the MM algorithm. The second and last step is to find parameter $\boldsymbol{\theta}$ that maximizes the surrogate function. In the next iteration, this newly found value is used as $\boldsymbol{\theta}_{\text{old}}$ [34].

Recall, that the state-value and the action-value functions following policy π are defined as

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] \qquad \text{and} \qquad q_\pi(s,a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a],$$

where G_t is the discounted reward. We define the *advantage function* a_π following policy π as

$$a_\pi(s,a) = q_\pi(s,a) - v_\pi(s) \text{ [33]}.$$

Intuitively, if taking action a in state s yields a higher expected return than the average action under policy π, the advantage is positive; otherwise, it is negative.

Advantage function will be of great importance. Suppose that $\boldsymbol{\theta}$ and $\boldsymbol{\theta}_{\text{old}}$ are two arbitrary parametrizations of policy $\pi_{\boldsymbol{\theta}}$. It can be shown (proof in [35]) that if $\gamma \in (0,1)$ and $\pi_{\boldsymbol{\theta}}$ is ergodic, these two can be related using the advantage function,

$$J(\boldsymbol{\theta}) = J(\boldsymbol{\theta}_{\text{old}}) + \mathbb{E}_{\pi_{\boldsymbol{\theta}}}\left[\sum_{t=0}^{\infty} \gamma^t a_{\pi_{\boldsymbol{\theta}_{\text{old}}}}(s_t, a_t)\right].$$

This can be rewritten as a sum over states and actions, given that $\pi_{\boldsymbol{\theta}}$ is ergodic and $\rho_{\pi_{\boldsymbol{\theta}}}(s)$ is a unnormalized discounted visitation frequency of state s under policy $\pi_{\boldsymbol{\theta}}$,

$$J(\boldsymbol{\theta}) = J(\boldsymbol{\theta}_{\text{old}}) + \sum_{s\in\mathcal{S}} \rho_{\pi_{\boldsymbol{\theta}}}(s) \sum_{a\in\mathcal{A}} \pi_{\boldsymbol{\theta}}(a \mid s) a_{\pi_{\boldsymbol{\theta}_{\text{old}}}}(s,a). \tag{2.1}$$

Recall that policy iteration (Section 1.4) indirectly uses this — if there is a positive advantage value in a state-action pair with non-zero probability, then the policy improves [33].

Looking at the Equation (2.1), because of dependency of $\rho_{\pi_{\boldsymbol{\theta}}}(s)$ on $\pi_{\boldsymbol{\theta}}$ it would be difficult to optimize it directly. Using the following approximation is one of the key ideas in the derivation of TRPO:

$$g(\boldsymbol{\theta} \mid \boldsymbol{\theta}_{\text{old}}) = J(\boldsymbol{\theta}_{\text{old}}) + \sum_{s\in\mathcal{S}} \rho_{\pi_{\boldsymbol{\theta}_{\text{old}}}}(s) \sum_{a\in\mathcal{A}} \pi_{\boldsymbol{\theta}}(a \mid s) a_{\pi_{\boldsymbol{\theta}_{\text{old}}}}(s,a). \tag{2.2}$$

It can be shown that a sufficiently small step that improves $g(\boldsymbol{\theta} \mid \boldsymbol{\theta}_{\text{old}})$ also improves $J(\boldsymbol{\theta})$ [33].

So far, this mostly proceeds from Kakade and Langford's paper [35]. They proposed a mixture policy update rule,

$$\pi_{\text{new}}(a \mid s) = (1-\alpha)\pi_{\text{old}}(a \mid s) + \alpha\pi'(a \mid s),$$

where $\pi'(a \mid s) = \mathbf{1}\{a = \arg\max_{a'} q_{\pi_{\text{old}}}(s,a')\}$. This mixture policy update rule is however not very practical [33].

2.2.1 Defining the Trust Region

The key idea in TRPO is to constrain how much the new policy parameters deviate from the old parameters. The parameter α is replaced by a distance measure. A natural choice is the *Kullback-Leibler (KL) divergence* [36],

$$D_{\text{KL}}(\pi_1(\cdot \mid s) || \pi_2(\cdot \mid s)),$$

which measures the average divergence between two policies. TRPO uses a heuristic approximation of this,

$$\overline{D}_{\text{KL}}^{\rho}(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2) = \mathbb{E}_{s\sim\rho}\left[D_{\text{KL}}(\pi_{\boldsymbol{\theta}_1}(\cdot \mid s) || \pi_{\boldsymbol{\theta}_2}(\cdot \mid s))\right].$$

This is a straightforward way to approximate the size of the policy update from policy parametrized using $\boldsymbol{\theta}_1$ to one parametrized with $\boldsymbol{\theta}_2$ across states sampled from ρ. So the TRPO update rule is

$$\text{maximize}_{\boldsymbol{\theta}} \; g(\boldsymbol{\theta} \mid \boldsymbol{\theta}_{\text{old}})$$
$$\text{subject to } \overline{D}_{\text{KL}}^{\rho_{\pi_{\boldsymbol{\theta}_{\text{old}}}}}(\boldsymbol{\theta}, \boldsymbol{\theta}_{\text{old}}) \leq \delta \text{ [33].}$$

2.2.2 Importance Sampling for Practical Use

In practice, the trajectories are collected using the current policy, $\pi_{\boldsymbol{\theta}_{\text{old}}}$, and the expectations must be estimated under the new policy $\pi_{\boldsymbol{\theta}}$. This is done by replacing the sum over the actions in Equation (2.2) using the *importance sampling*.

$$g(\boldsymbol{\theta} \mid \boldsymbol{\theta}_{\text{old}}) = J(\boldsymbol{\theta}_{\text{old}}) + \mathbb{E}_{s \sim \rho_{\pi_{\boldsymbol{\theta}_{\text{old}}}}} \left[\frac{\pi_{\boldsymbol{\theta}}(a \mid s)}{\pi_{\boldsymbol{\theta}_{\text{old}}}(a \mid s)} A_{\pi_{\boldsymbol{\theta}_{\text{old}}}}(s, a) \right],$$

where $A_{\pi_{\boldsymbol{\theta}_{\text{old}}}}(s, a)$ is an estimate of the advantage function. The TRPO update becomes

$$\begin{aligned} &\text{maximize}_{\boldsymbol{\theta}} \ \mathbb{E}_{s \sim \rho_{\pi_{\boldsymbol{\theta}_{\text{old}}}}} \left[\frac{\pi_{\boldsymbol{\theta}}(a \mid s)}{\pi_{\boldsymbol{\theta}_{\text{old}}}(a \mid s)} A_{\pi_{\boldsymbol{\theta}_{\text{old}}}}(s, a) \right] \\ &\text{subject to } \mathbb{E}_{s \sim \rho_{\pi_{\boldsymbol{\theta}_{\text{old}}}}} \left[D_{\text{KL}}(\pi_{\boldsymbol{\theta}_{\text{old}}}(\cdot \mid s) || \pi_{\boldsymbol{\theta}}(\cdot \mid s)) \right] \leq \delta, \end{aligned}$$

where $\delta > 0$ is a hyperparameter, determining the magnitude of the constraint [33].

The original paper also suggests using a penalty instead of a constraint:

$$\text{maximize}_{\boldsymbol{\theta}} \ \mathbb{E}_{s \sim \rho_{\pi_{\boldsymbol{\theta}_{\text{old}}}}} \left[\frac{\pi_{\boldsymbol{\theta}}(a \mid s)}{\pi_{\boldsymbol{\theta}_{\text{old}}}(a \mid s)} A_{\pi_{\boldsymbol{\theta}_{\text{old}}}}(s, a) \right] - \beta \overline{D}_{\text{KL}}^{\rho_{\pi_{\boldsymbol{\theta}_{\text{old}}}}}(\boldsymbol{\theta}, \boldsymbol{\theta}_{\text{old}}),$$

where $\beta > 0$ is a hyperparameter [33].

A note in the original paper describes how TRPO is closely related to a natural policy gradient. Specifically, it can be derived by applying a linear approximation to the surrogate objective and a quadratic approximation to the KL-divergence constraint — leading to an update in the natural gradient direction. Connection of this algorithm to policy iteration was shown [33].

2.3 Generalized Advantage Estimation (GAE)

An estimate of advantage function can be obtained by replacing value functions by their estimate in the advantage function definition:

$$A_{\pi}(s_t, a_t) = Q_{\pi}(s_t, a_t) - V_{\pi}(s_t),$$

where $Q_{\pi}(s, a)$ and $V_{\pi}(s)$ are the estimates. Because

$$Q_{\pi}(s_t, a_t) = \mathbb{E}[R_{t+1} + \gamma V_{\pi}(s_{t+1})],$$

where γ is the discount factor, this can be estimated as

$$A_{\pi}(s_t, a_t) = R_{t+1} + \gamma V_{\pi}(s_{t+1}) - V_{\pi}(s_t).$$

Recall that this is the TD error introduced in Section 1.5.2. Similarly to TD, this estimator has low variance, but unfortunately is biased (unless V estimator is unbiased) [37].

The derivation of *Generalized Advantage Estimation (GAE)* [37] is analogical to how the eligibility traces introduced in Section 1.5.5 are derived. This simple advantage estimator can be extended to multi-step returns, yielding an n-step advantage estimate:

$$\hat{A}_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_\pi(s_{t+n}) - V_\pi(s_t).$$

If the episode concludes before reaching n, the sequence is truncated. Next, using a decay parameter $\lambda \in \mathbb{R}, 0 \leq \lambda \leq 1$, the GAE estimator is defined as the exponentially weighted average of the n-step advantage estimate values,

$$A_\pi(s_t, a_t) = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} \hat{A}_t^{(n)}.$$

This is equivalent to a more compact form,

$$A_\pi(s_t, a_t) = \sum_{n=0}^{\infty} (\gamma\lambda)^n \delta_{t+n},$$

where $\delta_t = R_{t+1} + \gamma V_\pi(s_{t+1}) - V_\pi(s_t)$ [37].

Parameter λ controls the *bias-variance trade-off.* With $\lambda = 0$ the estimate is reduced to a one-step TD error, which has the lowest variance but retains the high bias of the imperfect value function estimator. With $\lambda = 1$, GAE uses the Monte-Carlo return (unbiased) minus the estimate $V_\pi(s_t)$. The $V_\pi(s_t)$ is cancelled out in the associated policy gradient, so it remains unbiased. Similarly to Monte-Carlo methods, however, this advantage function estimate would typically have a high variance [37].

State-value function v could be estimated by using an approximator function introduced in Section 1.6. A neural network used as a value-function approximator would typically be called a *value-function network.* Its output — the value function estimate — would then be used by GAE to obtain the advantage estimate.

2.4 Proximal Policy Optimization (PPO)

Natural policy gradient and trust region methods exhibit strong theoretical properties and practical performance, but they tend to be relatively complex. Proximal Policy Optimization, proposed in 2017 as a project of OpenAI, is simpler, and on many benchmarks shows better robustness, scalability and data efficiency [38].

Recall that the objective in each step of TRPO is to maximize the surrogate function,

$$\mathbb{E}_{s \sim \rho_{\pi_{\boldsymbol{\theta}_{\text{old}}}}} \left[\frac{\pi_{\boldsymbol{\theta}}(a \mid s)}{\pi_{\boldsymbol{\theta}_{\text{old}}}(a \mid s)} A_{\pi_{\boldsymbol{\theta}_{\text{old}}}}(s, a) \right],$$

which was derived from how two policies relate using the advantage function (Equation (2.1)), originally from [35].

The probability ratio between two policies, $\frac{\pi_{\boldsymbol{\theta}}(a|s)}{\pi_{\boldsymbol{\theta}_{\text{old}}}(a|s)}$ will be denoted as $r(\boldsymbol{\theta})$. Denoting the advantage estimate $\hat{A} = A_{\pi_{\boldsymbol{\theta}_{\text{old}}}}(s,a)$ and $\mathbb{E}_{s\sim\rho_{\pi_{\boldsymbol{\theta}_{\text{old}}}}}[\cdot] = \hat{\mathbb{E}}[\cdot]$ as in the original PPO paper for clarity, TRPO maximizes the following objective function:

$$L^{\text{CPI}}(\boldsymbol{\theta}) = \hat{\mathbb{E}}[r(\boldsymbol{\theta})\hat{A}].$$

CPI stands for conservative policy iteration. Because the update would otherwise be too large, TRPO introduced a Kullback-Leibler divergence measure to constrain it [38].

PPO uses a different strategy, it constrains the update by *clipping the surrogate function*,

$$L^{\text{CLIP}}(\boldsymbol{\theta}) = \hat{\mathbb{E}}[\min(r(\boldsymbol{\theta})\hat{A}, \text{clip}(r(\boldsymbol{\theta}), 1-\varepsilon, 1+\varepsilon)\hat{A})].$$

Clipping $r(\boldsymbol{\theta})$ ensures the ratio between the policies does not deviate too much from 1 (at most by $|\varepsilon|$). Then, PPO uses the more conservative update, which is either L^{CPI} or the clipped surrogate [38].

The PPO implementation is designed to reuse the samples multiple times, to be more effective. The agent performs actions for n steps to obtain trajectory $(S_0, A_0, R_1, S_1, A_1, \ldots, S_n)$. This trajectory is treated as a dataset of size n — the steps are shuffled and the data is separated into minibatches. These are then used to update the policy by optimizing the clipped surrogate objective function for a fixed number of epochs [38].

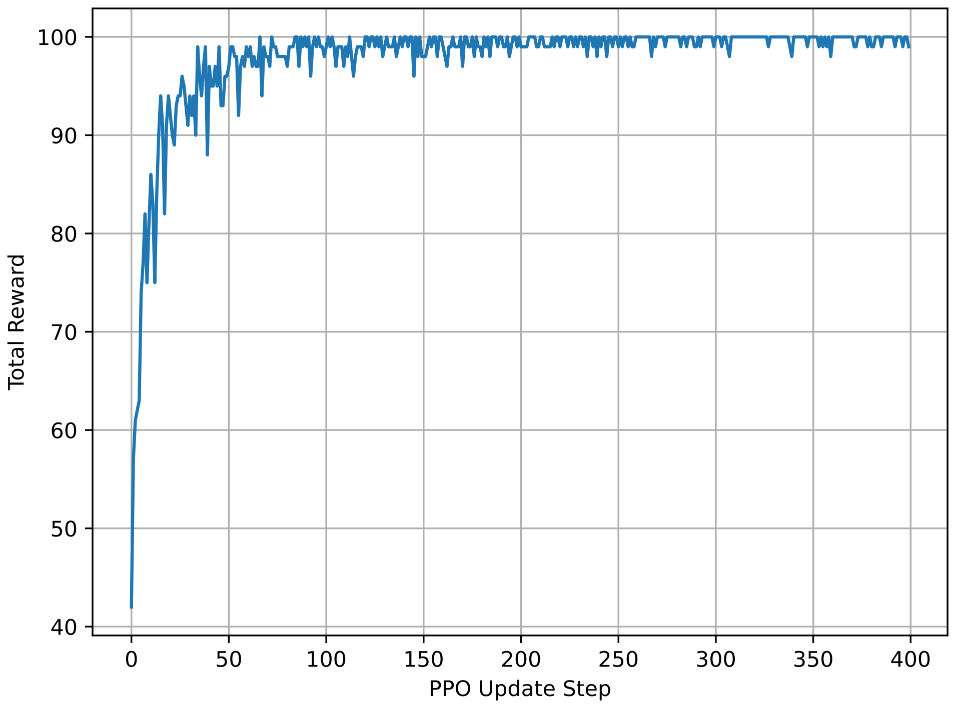
Another approach presented in the PPO paper is using the penalty coefficient,

$$L^{\text{KLPEN}}(\boldsymbol{\theta}) = \hat{\mathbb{E}}\left[\frac{\pi_{\boldsymbol{\theta}}(a \mid s)}{\pi_{\boldsymbol{\theta}_{\text{old}}}(a \mid s)}\hat{A} - \beta D_{\text{KL}}(\pi_{\boldsymbol{\theta}_{\text{old}}}(\cdot \mid s)||\pi_{\boldsymbol{\theta}}(\cdot \mid s))\right]. \text{ [38]}$$

The original TRPO paper experimented with this approach, but found the coefficient β very difficult to establish robustly enough — β too high leads to slow optimization, while β too low lead to instability [33]. In PPO, the penalty coefficient β is adjusted dynamically after each policy update based on the observed KL divergence:

- Optimize $L^{\text{KLPEN}}(\boldsymbol{\theta})$ for several epochs using minibatch SGD.
- With $d = \hat{\mathbb{E}}\left[D_{\text{KL}}\left(\pi_{\boldsymbol{\theta}_{\text{old}}}(\cdot \mid s) \parallel \pi_{\boldsymbol{\theta}}(\cdot \mid s)\right)\right]$, adjust β:
  - If $d < d_{\text{targ}}/1.5$, halve β.
  - If $d > d_{\text{targ}} \times 1.5$, double β.

d_{targ} is used as the target divergence term. They, however, found that this method performed worse than the clipped surrogate function objective [38].



■ **Figure 2.1** PPO used on the simple screen environment. Note that the initial fluctuation seen in Figure 1.11 are not present, as the variance in policy gradient estimates is lower.

To use this in practice, all that needs to be done is replacing the policy gradient objective function with L^{CLIP} or L^{KLPEN}. Commonly used frameworks like PyTorch [39] contain automatic differentiation, so this process becomes very straightforward [38].

▶ **Example 2.1.** Example 1.4 showed how to parametrize the touchscreen environment introduced in Example 1.1 so that policy gradients could be obtained and optimized using the REINFORCE algorithm. The approximation there was very simple: a plain sigmoid function over a linear combination of two parameters. In practice, the approximations usually need to be more complex — the most common approach is to use a neural network, which is then called *policy network*. Obtaining gradients of the objective function with respect to the policy network's parameters is done automatically, using the automatic differentiation.

Recall that in Example 1.4, the chain rule was used to calculate the gradients. Neural networks and their frameworks are designed around this. A neural network can be decomposed into a sequence of functions, whose partial derivatives are simple. These partial derivatives are then "chained" together, yielding the gradient of the whole network. This allows for massive scalability.

By configuring the PPO algorithm on the simple screen environment example with the policy used in the REINFORCE example, where the clipped objective $L^{\text{CLIP}}(\boldsymbol{\theta})$ is optimized for a single epoch using minibatch of size one after each step with the clipping parameter $\varepsilon = 0.2$, Figure 2.1 demonstrates strong performance. Small value-function estimation learning rate of 0.003 was used. It is important to note, however, that the environment considered here is extremely simple.

2.5 Intrinsic Motivation

The methods and approaches discussed so far did not address the design of a reward function. While this process might seem relatively straightforward, the natural reward signals can be very sparse or missing altogether in many environments. In such scenarios, it is desirable to encourage the model to explore, and the field of intrinsically motivated exploration is focused on finding ways to achieve this.

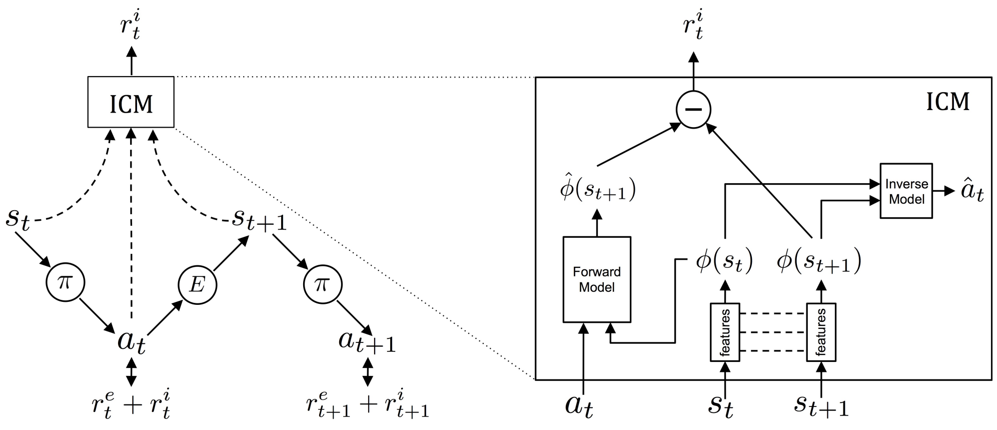
The reward r_t is in each time step decomposed into *extrinsic reward* r_t^e awarded from the environment (e.g. scoring in a pong game) and *intrinsic reward* r_t^i, awarded from within the agent as an exploration bonus. The approaches that will be discussed here are curiosity-driven learning approaches, which treat prediction error as a reward. In other words, not being able to predict the outcome of an action would reward taking the action with higher reward.

A study from 2018 [40] found that reinforcement learning agents were able to learn useful behaviour purely from the intrinsic reward. For example, they show that in Super Mario Bros., the agent was capable of *"discovering 11 different levels of the game, finding secret rooms and defeating bosses"* [40].

2.5.1 Intrinsic Curiosity Module (ICM)

In the original work from 2017 where ICM was introduced [41], the authors separate the RL agent into two subsystems: *Intrinsic Curiosity Module (ICM)* and a policy estimator (e.g. PPO). Policy estimator decides on which action the agent should take and is being optimized by using the extrinsic reward from an environment combined with the reward generated by ICM, $r_t = r_t^e + r_t^i$. In this case, the extrinsic reward r_t^e is expected to be mostly or always zero (this is not necessary to use the ICM) [41].

An illustration of the ICM architecture is shown in Figure 2.2. Its input consists of current state s_t, next state s_{t+1} and the action a_t that the policy selected to move between the two states. The ICM produces the intrinsic reward r_t^i. It uses feature extractor ϕ to extract features from states, as it is undesirable to work with raw sensory data, such as individual pixels. The ICM then operates only with the embedded states $\phi(s_t)$ and $\phi(s_{t+1})$ [41].



■ **Figure 2.2** A diagram of the ICM module, reproduced from [41]. Note that policy π operates independently of ICM, it uses only the rewards for learning.

The two main parts of the module are *inverse model* and *forward model.* The forward model predicts the next state embedding $\hat{\phi}(s_{t+1})$ from previous state $\phi(s_t)$ and the action a_t. Its parameters $\boldsymbol{\theta}_F$ are optimized to minimize the squared distance between the predicted $\hat{\phi}(s_{t+1})$ and the actual $\phi(s_{t+1})$, its loss function is

$$L_F(\phi(s_{t+1}), \hat{\phi}(s_{t+1}) \mid \boldsymbol{\theta}_F) = \frac{1}{2}\|\hat{\phi}(s_{t+1}) - \phi(s_{t+1})\|^2.$$

This distance is also used as the intrinsic reward, i.e.

$$r^i_t = \frac{\eta}{2}\|\hat{\phi}(s_{t+1}) - \phi(s_{t+1})\|^2,$$

where η is a scaling factor. The intuition behind the forward model is simple — the model rewards unpredictable/unexpected response from the environment [41].

The other component, the inverse model is utilized to model the latent dynamics of the system and it is used mainly to learn good features in the feature extractor. Its input would be the embeddings of the two states consecutively visited, $\phi(s_t)$ and $\phi(s_{t+1})$, and its output $\hat{a}_t$ is the prediction of action that the policy selected, using parameters $\boldsymbol{\theta}_I$. Its loss function $L_I(a_t, \hat{a}_t \mid \boldsymbol{\theta}_I)$ depends on how the actions are defined. In the case of discrete actions, the softmax function is used at the output of the inverse model and categorical cross-entropy loss function may be used to optimize its parameters [41].

The overall objective of the RL agent, where $J(\boldsymbol{\theta})$ is the policy's performance measure (in previous sections, expected discounted return was used), is

$$\text{minimize}_{\boldsymbol{\theta}, \boldsymbol{\theta}_I, \boldsymbol{\theta}_F} \left[-\lambda J(\boldsymbol{\theta}) + (1-\beta) L_I(\boldsymbol{\theta}_I) + \beta L_F(\boldsymbol{\theta}_F)\right],$$

where $\beta \in [0, 1]$ weighs the importance of the forward vs. inverse model and $\lambda > 0$ weighs the importance of policy gradient optimization [41]. Again, in practice, this becomes simpler with the use of automatic differentiation.

2.5.2 Random Network Distillation (RND)

A promising method was introduced in 2018 as *Random Network Distillation (RND)* [42]. Its idea lies in awarding higher intrinsic reward r_t^i for novel or less frequently visited states. Similarly to ICM, the RND module is separated from the policy estimator [42].

Uncertainty of a model, resulting in error in its prediction objective, can generally be decomposed into *aleatoric* and *epistemic* components [43]. While aleatoric uncertainty is caused by the stochasticity of the target function, epistemic uncertainty results from incomplete knowledge of the model. In other words, epistemic uncertainty will be higher for less frequently visited states and can be used as an intrinsic reward for exploration [42].

The RND approach is conceptually straightforward. Consider a general observation space $\mathcal{O}$. It is a space in which any observation can be expressed. In a typical environment, that would be the space of image observations, sensory inputs, etc. RND uses two neural networks,

- A randomly initialized neural network $f : \mathcal{O} \to \mathbb{R}^k$ with randomly initialized, fixed parameters.
- A predictor neural network $\hat{f} : \mathcal{O} \to \mathbb{R}^k$ with trainable parameters $\mathbf{w}$, also initialized randomly (but with different parameters than f).

During the training process of the RL agent, for each observation $o \in \mathcal{O}$, the predictor neural network is updated via SGD to minimize the expected mean square error

$$\mathrm{MSE}(\mathbf{w}) = \|\hat{f}(o \mid \mathbf{w}) - f(o)\|^2.$$

The prediction error is used as the intrinsic reward, as it is expected to be higher for novel or less frequently visited states [42].

Since the prediction error becomes lower over time, the paper suggests keeping a running estimate of the standard deviation of prediction error and dividing the reward by it [42].

As a crucial step, observation normalization is introduced to stabilize the learning. Since the random network's scale is fixed, it cannot adjust for the scale of the inputs. For $o \in \mathcal{O}$,

$$\overline{o} = \mathrm{clip}\left(\frac{o - \mu}{\sigma}, -5, 5\right),$$

where μ is the running mean and σ is the running standard deviation. The agent initializes μ and σ by taking a few sample steps in the environment. The observation normalization is done only for the RND, not for the policy estimator [42].

The original paper demonstrates that the predictor network trained via SGD does not start to mimic the random network perfectly. This is crucial, as otherwise the intrinsic reward would collapse to zero [42].

An additional contribution is the exploration of combining extrinsic and intrinsic rewards. The proposed method is fitting two value heads $\hat{V}_e$ and $\hat{V}_i$ separately for each reward and using their sum to obtain $\hat{V}$ [42].

Chapter 3

Robotic Interface Interaction

This chapter discusses methods by which a RL agent-controlled robot can interact with a physical environment. First, two distinct approaches to the task are presented and analyzed. Subsequently, methods for realization of each are proposed and explored in detail.

Robots use sensors to perceive their environment and actuators to supply the motive power needed to interact with it. A robot is not defined by its use of the sensory data and decision making; in fact, *"in actual practice, in devices considered to be robotic, the amount of sensory and decision making capability may vary from a great deal to none"* [44]. The systems discussed in this thesis are designed for robots that possess a high level of decision-making capability.

The aim of this thesis is to develop a system for efficient robotic exploration of touchscreen interfaces. The robot's only means of perceiving the environment is a camera aimed at the display of the device. Using its actuators, the robot can perform actions directly on the touchscreen.

The robot's screenshot provides valuable information; however, raw pixel data lacks structural context. Extracting meaningful information, *features*, manually from such input is a challenging and often impractical task. Recent advancements in machine learning have introduced models that work with images with unprecedented effectiveness.

In this work, the task of screenshot processing is approached from two distinct directions:

1. *Detection of Clickable Elements*: The system first identifies all the clickable elements on the screen. These elements are provided to the RL agent, which selects the appropriate element to interact with based on its policy.

2. *Feature Extraction for Decision-Making*: The system extracts a set of features from the screenshot that characterizes the state of the environment. These features are provided to the RL agent, which determines the exact point on the screen to click based on its policy.

The first approach is simple and reduces the burden on the RL agent. However, its main drawbacks are a lack of robustness and a requirement for a large annotated dataset. Second approach is more flexible and can work without predefined annotations, but requires the RL agent to select from much larger action space (it can click anywhere on screen) using only the extracted features as the information about the environment.

These approaches will be further discussed in the following sections.

3.1 Detection of Clickable Elements

The first approach explored in this work is a guided strategy, where a separate machine learning model identifies interactive elements in a screenshot. An interactive element is one that after interaction causes a state change in the environment. By isolating such elements, the RL agent can concentrate on more effective exploration as it doesn't need to detect these elements by itself.

However, this method requires a large set of annotated data, which limits the robustness and scalability of the approach. In practice, each platform would require its own pre-trained clickable element detector. While a solution for one platform may be portable to some extent — for instance, icons across different mobile operating systems share visual similarities — any undetected interactive element risks cutting off the agent's access to whole branches of the state space. An example of such a failure would be missing the detection of a 'home' button on a mobile phone. Without the ability to return to the home screen, the robot would become stuck inside an application, leaving most of the state space unexplored.

A further limitation is that without appropriate data, this method is limited only to clicking on the elements. It cannot swipe, double tap or perform other, more complex, actions. Such datasets are typically not readily available and would require more manual effort to produce.

This thesis suggests *YOLO (You Only Look Once)* architecture [45] as a possible method for element detection.

3.1.1 Element Detection with YOLO

YOLO is a supervised learning model introduced in 2015 for object detection. It is illustrated in Figure 3.1. The input image is divided into a grid of size $S \times S$, where S is a hyperparameter. The key idea is that if the center of an object falls within a particular grid cell, that cell is responsible for detecting the object. Each grid cell predicts B bounding boxes and their confidence scores, and it also predicts probabilities of the grid cell belonging to any of the C predefined classes. Both C and B are hyperparameters. In this work, the classification part will not be necessary. However, it could become useful if additional interaction types, such as swipe or double-tap, were incorporated.

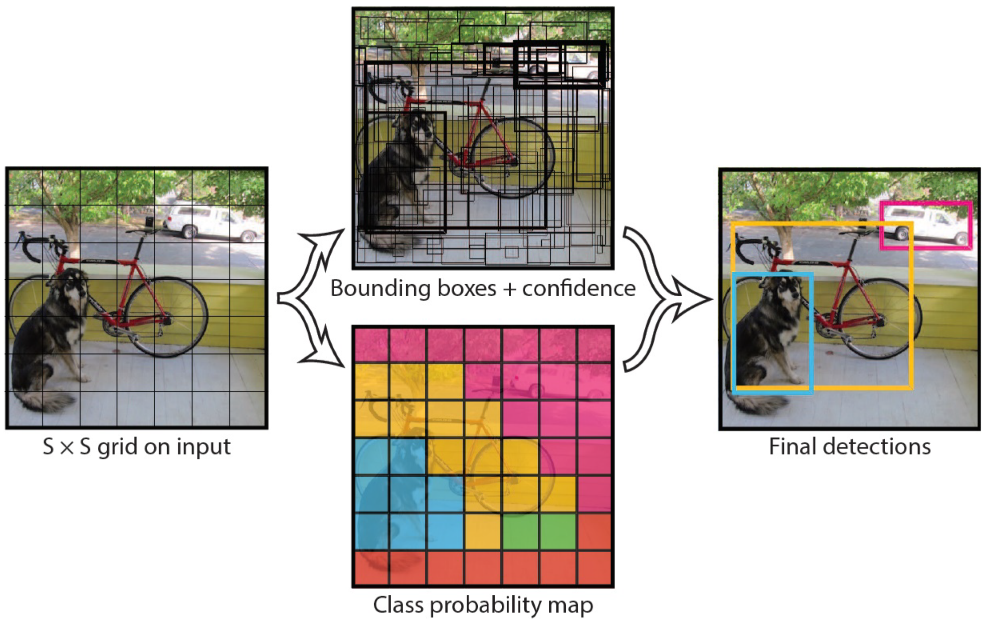


Figure 3.1 A YOLO model diagram, reproduced from [45].

The confidence of predictions is trained to predict the intersection-over-union (IoU) between the predicted and ground truth bounding boxes [45].

One of the main advantages of this model is its high performance, as YOLO can detect objects in real-time. It can be implemented using a single convolutional neural network (24 convolutional layers followed by 2 fully connected layers), hence its name, *You Only Look Once* [45].

The YOLO model evolved and new versions were developed. YOLOv5 was the first version of this object detection model to use the PyTorch [39] framework instead of a framework made by the YOLO author [46]. At the time of publishing this thesis, the newest version is YOLOv12, which is a first fully attention-centric version of YOLO that still achieves real-time FPS comparable to previous CNN-based versions [47].

3.1.2 Interaction with the Elements

Once the interactable elements are detected, the subsequent steps are relatively straightforward. The coordinates of the centers of their bounding boxes are mapped from image space to real-world coordinates. Based on this information, the RL agent selects an element to interact with.

While the reinforcement learning part of this thesis discusses mostly fixed action spaces, there are approaches to handle variable action spaces. One of the most interesting RL papers in recent years, [48], introduces an agent that is able to reach Grandmaster level in the videogame StarCraft II. It uses an auto-regressive policy and pointer networks to manage structured combinatorial action space (actions are composed of multiple interdependent choices, where earlier selections alter the set of available next ones) [48]. However, this approach extends beyond the scope of the current work. Another similar paper, [49], which presented an agent that could defeat a world champion team in the Dota 2 videogame, uses an approach similar to action masking — invalid actions are ignored [49].

Designing a policy network to produce probabilities over clickable elements would require incorporating additional information about the environment, making its structure inherently more complex. However, once the clickable elements are detected, even random uniform policy may produce relatively good results and no learning of the RL agent would be necessary.

3.2 Feature Extraction for Decision-Making

The other approach's idea is to use features extracted from a screenshot as an input to the RL agent's policy network and allowing the robot to perform any action on the screen. This approach is naturally more robust — the scenario where the robot doesn't detect a button cannot happen, because an explorative policy will always click it, eventually. For it to be efficient, however, the policy is required to learn to navigate in the feature space and connect properties of the encoded screen to certain actions. The action space naturally becomes much larger, even more so if special actions, such as swiping or double-tapping, are included.

3.2.1 Extracting Features from Screenshot

Traditional image processing relied on handcrafted feature extractors, from edge detectors to more advanced methods like SIFT [50] from 1999. The concept of convolutional neural networks (CNNs) and their training using backpropagation was introduced by LeCun et al. [51] in 1990; however, due to a lack of computing power and the absence of large training data, their practical adoption was delayed. The field started developing especially after 2006 [52]. Notably, in 2012, Krizhevsky et al. [53] showed state-of-the-art performance in image classification on the ImageNet dataset, significantly outperforming the SIFT method. CNNs were mentioned in Section 3.1.1 about YOLO as its main component. This demonstrates how effective CNNs are for image-related tasks.

In 2017, new deep learning architecture, called *Transformer* [55], was introduced. While the standard Transformer architecture worked with text, a *Vision Transformer (ViT)* [54] enabled to use images in a similar way to how Transformer works with text. Its principle is illustrated in Figure 3.2. An image is first divided into a grid of fixed-size patches, each patch is flattened into

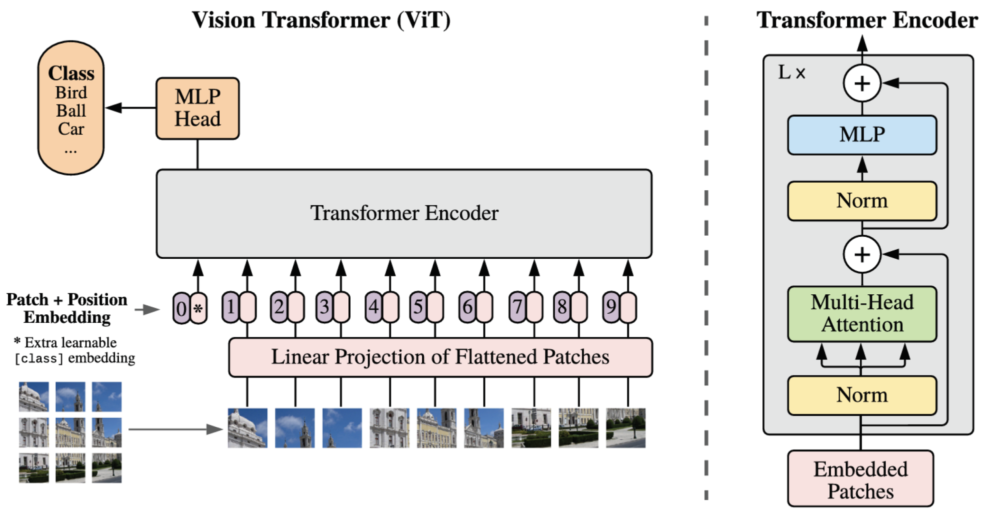


Figure 3.2 Illustration of the Vision Transformer, reproduced from [54]. Note that the Transformer Encoder remains unchanged from the original Transformer architecture.

a vector by linear projection, and the resulting sequence of patch embeddings is fed into the Transformer Encoder [54].

An interesting variant of ViT is the *Swin Transformer* [56]. Its authors discuss how the ViT approach overlooks certain properties of images that differ from text and propose alternative solutions. First, the high resolution of patches makes the encoding inaccurate and prediction at the dense pixel level impossible. Another issue is that, unlike word tokens, visual entities can vary in scale, and the image space is much larger than that of textual embeddings [56].

The paper proposes a hierarchical architecture with multiple layers. After each layer, adjacent patches are merged, so that deeper layers operate on fewer but larger patches with more feature channels, forming a hierarchical representation [56].

Unlike the Vision Transformer (ViT), which applies self-attention globally across the entire image, the Swin Transformer restricts self-attention to small, non-overlapping local windows within each layer — in other words, attention is limited to a few neighboring patches. In the following layer, the window configuration is shifted so that the new windows span across the boundaries of the previous ones. This shift enables information to flow across windows, which allows for connection of the attention between layers [56].

Visual feature extractors discussed so far perform well with images, but screenshot contains more information than that. In recent years, *multimodal architectures*, which can work with inputs in various formats, started gaining prominence. Such formats can be text, images, bounding boxes, or generally any information that can be somehow embedded. A *LayoutLMv3* [57] architecture has certain properties that may be useful for the objective of this

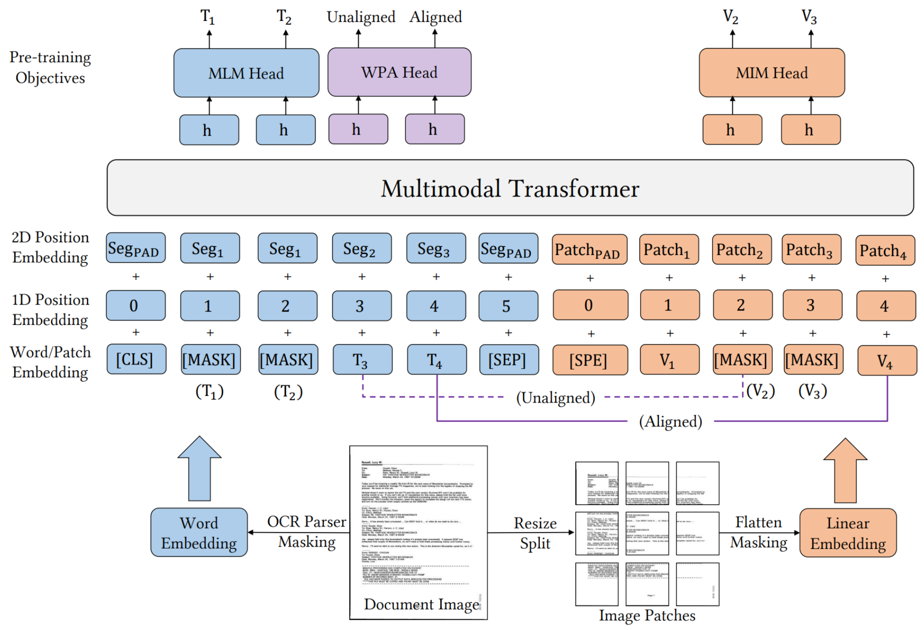


Figure 3.3 An illustration of LayoutLMv3 pre-training objectives, reproduced from [57]. The objectives are reconstructive Masked Language Modelling (MLM), Masked Image Modelling (MIM) and predictive Word-Patch Alignment (WPA).

thesis.

LayoutLMv3 was designed for Document AI tasks, which apply artificial intelligence methods to facilitate working with documents. It extends standard Transformer Encoder with a *multimodal embedding layer*, consisting of two parallel pipelines. First one performs image embedding using patches in a similar way to ViT. Second pipeline produces text embedding, which uses 1D positional encoding, indicating the position of a token in the input sequence (as in Transformer) and additionally a *2D layout position*, referring to the bounding box of the text in document. As a result, LayoutLMv3 embeddings contain visual information, textual information and information about the positions of the text in the document, which captures the layout of the document [57].

During pre-training, a portion of the word tokens and image patches are masked, and the network is trained with the objective of reconstructing them. In addition to this masked language and image modeling (MLM and MIM), LayoutLMv3 introduces a Word-Patch Alignment (WPA) objective, where the model learns to predict whether an image patch corresponds to a given text token. This task encourages the model to align visual and textual representations. These objectives are illustrated in Figure 3.3 [57].

The discussed networks might be designed for specialized tasks, such as classification. However, they can be repurposed as feature extractors simply by cutting off the last few layers. For example, networks designed for classification often end with a few multi-layer perceptron (MLP) layers followed by a softmax activation. By removing these MLP layers, the network outputs abstract intermediate features that encode high-level information about the input. This approach can even be applied to pre-trained networks. The result is a working feature extractor that requires minimal or no additional training. Finetuning the network, however, could make it more domain-specific.

3.2.2 Large Action Spaces

As already mentioned, by allowing the robot to perform any arbitrary action, the state space becomes very large. The simplest approach to mitigate this is using *discretization* into a fixed size state space. This approach will be used for the practical part of this thesis. However, there are other, more advanced approaches. The following two have interesting concepts and could be used to further improve the training speed.

First of the approaches, introduced in 2016, presents the *Wolpertinger* architecture [58] as an extension of the classic Actor-Critic architecture (Section 1.7.1), where a distance measure over actions is defined. The actor selects (more appropriate wording could be generates) an intermediate *"proto-action"* $\hat{a}$, that may not lie in the action space. Using k-nearest neighbors, n most similar actions from the action space are found and critic selects the action that yields the highest estimated action-value [58].

The other approach's idea is very intuitive. Instead of learning which actions to take, an *Action Elimination Network* [59] (AEN) may be used to eliminate actions and reduce the action space. In the view of standard reinforcement learning loop (agent-environment), environment produces new state, reward and an *elimination signal* used to train the Action Elimination Network [59]. For the task of touchscreen interface navigation, an elimination signal could be designed around actions that do not change the state of the environment.

Implementing these would require further design and testing. Since this thesis is mainly focused on intrinsic motivation integration, it will not be used in the practical part.

Chapter 4

Experimental Setup

This chapter outlines the experimental framework used for the practical part of this thesis. It introduces the hardware and software environment, details key technical decisions, and discusses practical considerations and challenges encountered during the implementation and testing phases.

4.1 The AIVA Testing Robot

AIVA [1] is a robot developed by Y Soft for automating the testing and interaction with touchscreen devices, reducing manual effort in quality assurance workflows. It features two main components: a robotic arm with a stylus for simulating touch, and a camera that captures the device's screen. The setup is shown in Figure 4.1.

The AIVA robot communicates via RESTful APIs. For the purposes of this work, three of these APIs are utilized. API names in this section follow Y Soft's internal naming conventions. *RobotControl* API enables control over the robot's motion and *ImageProcessing* API is used for camera operations, including taking a screenshot. AIVA additionally utilizes the *Peripherals* API. In case of failure, this API allows AIVA to be restarted to its initial state. OpenAPI Generator [60] was used to generate Python client libraries from the APIs for better integration into the scripts.

4.1.1 Error Handling

API calls can occasionally fail for various reasons such as network instability. To mitigate this, the implemented solution uses a retry mechanism — the API call is repeatedly attempted until a successful response is received. A short delay, within a matter of seconds, is introduced between consecutive calls to prevent large request frequency and allow the issues to resolve. If the system failed to perform the call after multiple attempts, the system restart procedure discussed previously is used as a fallback.

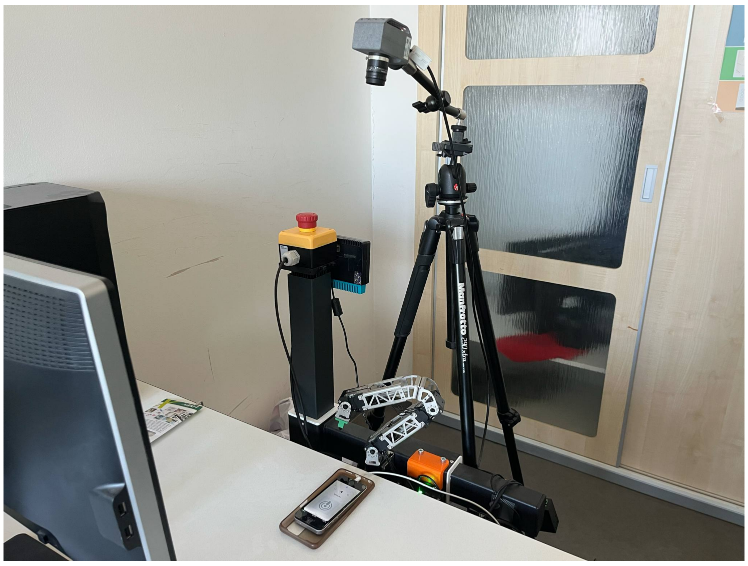


Figure 4.1 The AIVA robot setup. The two main components are the robotic arm and a camera.

This mechanism enabled the robot to recover from various runtime issues. Training took multiple days and was typically unsupervised. Training failures often went undetected for long periods, resulting in significant time loss.

The AIVA system requires active internet connection. This offers several advantages, including rapid support from the Y Soft development team during technical difficulties. However, there are shortcomings to the always online requirement. For instance, during the course of this project, a router experienced a malfunction and had to be replaced. This resulted in nearly a month of downtime, severely impacting the progress of the work.

4.2 Element Detection Approach Setup

The implementation of the first approach — interactive element detection, discussed in Chapter 3, uses YOLOv5 from Ultralytics [61] for clickable element detection, specifically a pretrained `yolov5m.pt` model (medium-sized variant), which, according to [46], performs slightly worse than larger models, but have lower computational cost. It is trained to detect a single class (clickable elements) for 50 epochs and the images used for training were resized to size 640×640. Using a batch size of 4 made it possible to train the model on GPU with 12 GB VRAM. This made the training process significantly faster. Each epoch (≈53k images were used for training) took approximately 16 minutes to run.

Finding an appropriate dataset for training the model is challenging. For the practical part, the YOLO model was trained on the *RICO* dataset [62].

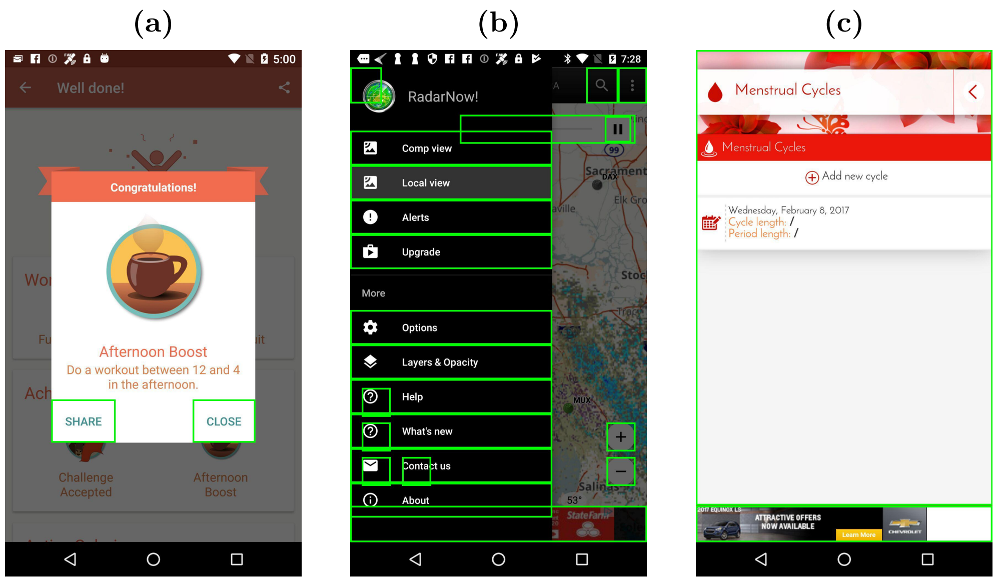


Figure 4.2 Examples from the RICO dataset 3 with highlighted bounding boxes for elements marked as clickable and visible. The dataset contains (a) well-annotated data, (b) data with minor problems, and (c) poorly annotated data.

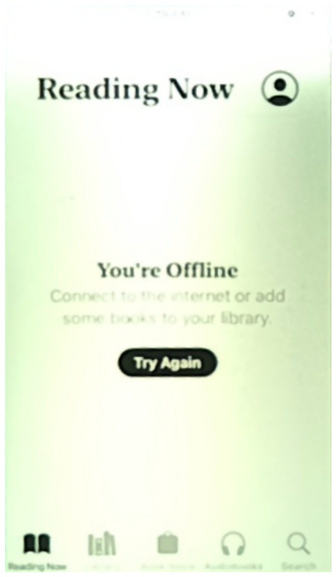
This dataset comprises ≈66k images of Android application UIs with their JSON annotations. These contain information about the UI and its components, including clickable elements. As depicted in Figure 4.2, a substantial portion of the dataset suffers from inaccuracies, which makes the dataset slightly ineffective — as mentioned, to enable full exploration, accurate detection of the elements is crucial. The inaccuracies include missing elements, annotated elements that are not directly visible and falsely annotated clickable elements. It is expected that the YOLO model will learn to generalize over these inaccuracies and provide reasonable results.

For the training, the dataset was split randomly into two subsets: 80% for training and 20% for evaluation. The annotations were converted into a format compatible with YOLO using a custom Python script.

The objective for practical experiments will be to evaluate how well the model has learned to detect clickable elements in the evaluation data, assess its ability to generalize to UIs from a different operating system and determine its usability when integrated with the AIVA robot.

4.3 Feature Extraction Approach Setup

For feature extraction, the LayoutLMv3 model described in Section 3.2.1 was used, specifically, the pretrained `microsoft/layoutlmv3-base` model available via the HuggingFace library. This model was pre-trained on 11 million document images [57] from IIT-CDIP dataset [63] (more on training process in


Reading Now

You're Offline

Connect to the internet or add some books to your library.

Figure 4.3 An example screenshot. Note that not all text is clearly visible.

Section 3.2.1). By using this pre-trained version, the model remains relatively general in scope while still effectively capturing the layout structure present in the input data.

While OCR can be incorporated relatively easily, it is not used in the practical part of this work due to issues with text readability in the screenshots. These were mainly caused by the combination of camera quality and the reflected backlight of the screen, which made the OCR output unreliable. An example of this issue is depicted in Figure 4.3. Since LayoutLMv3 is a multimodal network, even after omitting the textual input, the model should still be able to capture the layout of the image.

The Swin Transformer used to compare the LayoutLMv3 to a more general encoder is `microsoft/swin-tiny-patch4-window7-224` [56] also available via the HuggingFace library. This model was pre-trained on ImageNet dataset [64], a hierarchically organized large general image database.

Finally, for the evaluation of experiments, a proximity-based detector of new states will be employed. It checks whether the distance between the extracted features of a new state and those of a previously observed states exceeds a predefined threshold, in order to determine whether the state has been encountered before.

4.4 Reinforcement Learning Agent Setup

The reinforcement learning agent consists of two parts, the policy network and the module for intrinsic motivation. How they communicate is illustrated in Figure 4.4. They both can be implemented in various ways, the following

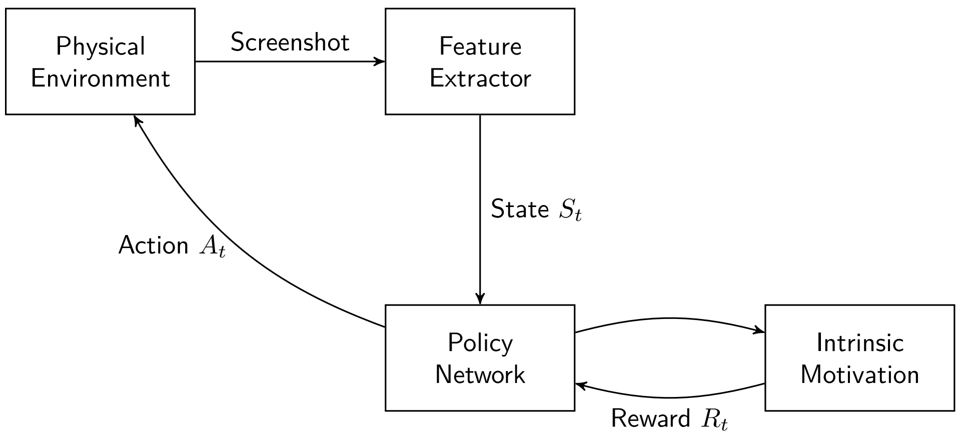


Figure 4.4 An overview of the system learning process. The information used by policy network for intrinsic motivation depends on the specific implementation. Action A_t is mapped to a real-world action.

sections describe the implementations used in this work.

4.4.1 Policy Network

This work uses PPO procedure described in Section 2.4. Specifically its implementation from Stable-Baselines3 [65]. This implementation uses slightly different approaches than the original to make the learning process more stable. The objective of the policy network is to model the probabilities of actions as a discrete, uniformly distributed set of points on the screen, where the robot can click. The actions of the robot are limited to click only; methods of how this can be extended will be discussed at the end of this work. The implementations use 15×25 distribution of these points, reflecting the dimensions of the device. Stable-Baselines3 allows for defining a custom policy network.

The primary approach explored in this thesis takes a screenshot as an input and utilizes a trainable convolutional network. Stable-Baselines3 includes an implementation of convolutional neural network from [66], which comprises three convolutional layers. Their output is mapped to a feature vector of size 512 (by default) using a linear layer. While the network expects images in a channel-first format, (C, H, W), Stable-Baselines3 provides a wrapper (VecTransposeImage) that automatically transposes the image.

An alternative approach is to use a fully connected neural network applied to features produced by a separate pre-trained feature extractor. Stable-Baselines3's default MLP policy uses two hidden layers of 64 units each for both policy network and the value function network. This approach is only used with ICM to test whether it is a viable alternative.

While the size of the MLP in the alternative approach may seem small given the large action space, it could be well-suited for the task for two main reasons.

First, smaller networks learn faster and are more stable. Note that the robot is very slow, executing around only 500 actions in an hour. Unlike typical reinforcement learning settings that rely on fast simulation and parallelism, the real-world setting limits data collection speed. Second, although the action space is large, it is highly structured. Many actions represent nearby points on a screen and have similar effects.

The rewards of the basic setup are defined using the proximity-based detector of new states. The agent is awarded 1, if it discovers a new state, and 0 otherwise. Note, however, that the detector using a distance of features to determine if a new state was reached only approximates state novelty. Some applications, where the robot can freely change the state (such as writing and sending SMS, or refocusing camera) may lead to robot recognizing new state was reached and awarding reward after every action, even though practically the robot remains on the same screen. On the other hand, small changes that should be perceived as a state change may not be caught.

4.4.2 Implementation of Intrinsic Motivation

The cornerstone of this thesis is intrinsic motivation, which is explored through two approaches: the Intrinsic Curiosity Module (ICM) and Random Network Distillation (RND), both discussed in Section 2.5. Both are implemented in PyTorch [39].

4.4.2.1 ICM Implementation

The forward model of ICM concatenates the extracted features of current state S_t (LayoutLMv3 uses size 768) with one-hot encoded action A_t. This tensor is passed to a MLP with a single hidden layer of size 256. The MLP is optimized to predict the state S_{t+1} representing the screen the UI ends up with, using MSE loss function $L_{\text{MSE}}(S_{t+1}, \hat{S}_{t+1})$. $\hat{S}_{t+1}$ is the predicted next state.

Because an external feature extractor is used, the ICM's inverse model is redundant. This is done not only to reduce the number of parameters that must be optimized during training to make the learning process more efficient, but also to eliminate the need to tune the weighting hyperparameter between the forward and inverse losses.

The loss function is then reduced only to

$$L_{\text{ICM}} = L_{\text{MSE}}(S_{t+1}, \hat{S}_{t+1}),$$

and this loss function is optimized using Adam optimizer with a standard learning rate of 1×10^{-3}. PyTorch's automatic differentiation makes this process very straightforward. ICM's parameters are updated after each step in the environment.

The value used as intrinsic reward is

$$R_t^i = \frac{1}{n} \sum_{j=1}^{n} \left(S_{t+1}^{(j)} - \hat{S}_{t+1}^{(j)} \right)^2$$

where n is a feature dimension of the state vectors. This value is essentialy the mean squared error multiplied by coefficient α, but the implementation doesn't use L_{MSE} and averages over dimension 1 (dimension 0 corresponds to the batch axis) so that it can be used for larger batch size. With the AIVA robot used in this work, however, batch size is always one, so the values are equal.

The training of the agent using ICM used episodes of length 100, after which the environment was reset.

4.4.2.2 RND Implementation

The RND implementation uses the same architecture for both the predictor and the random network. It comprises two fully connected layers, each projecting the input to a tensor of size 512. Another crucial component of RND is observation normalization. A Stable-Baselines3's [65] `RunningMeanStd` is used to track running mean and variance of the inputs. The same class is used to keep track of variance of intrinsic reward. The class actually tracks variance, but the standard deviation (root square of variance) is used in the normalization formula.

The input observation, which are the extracted features from the screenshot using feature extractor, is normalized exactly how it is described in Section 2.5.2. The standard normalization formula is used,

$$o' = \frac{o - \mu}{\sigma},$$

where o is observation, μ is running mean and σ is the running standard deviation. The result is then clipped in the $[-5, 5]$ range.

The normalized observation is then passed into both neural networks, the predictor $\hat{f}$ and the random f, and a mean-squared error is used as the loss function. The predictor network is optimized using the SGD.

Reward is calculated similarly to how it is done in the ICM. The mean-squared error of each sample in the batch, in case of this work, the one sample, is divided by the running mean of the reward. This value is multiplied by coefficient α, which would be used to configure the ratio between intrinsic and extrinsic reward. Since experiments in this work will use only the intrinsic reward, this parameter is unimportant.

The original paper states that in experiments that only used intrinsic reward, *"treating the problem as non-episodic resulted in better exploration"* [42]. Because of this, the RND is implemented to continuously explore the environment (non-episodically).

4.4.3 Tuning and Optimization

Due to the use of a real-world robot, training was constrained by physical time, unlike in most reinforcement learning applications in a simulated environment. This made full-scale hyperparameter optimization highly infeasible. Measuring policy improvement was also challenging, as evaluation itself takes a lot of time. While the primary quantitative metric of the training performance was the evolution of the reward function, qualitative observation played an important role.

Fortunately, the methods of intrinisic motivation don't need tuning, as in this work's implementation, they do not contain other hyperparameters (than for example the neural network size) whose selection wasn't justified in the text above. Stable-Baselines3 [65] also provides robust default hyperparameters for PPO, which served as a reliable baseline.

Changes in PPO hyperparameters did not lead to an improvement in training speed and only led to instability. In one training run, increasing the learning rate $10\times$ to 3×10^{-3} led to a policy collapse, where the agent was repeatedly selecting single action (that did not change the state, resulting in low rewards).

Balancing exploration/exploitation was briefly discussed in Section 1.5.3, but the idea of exploration in context of this task is not intuitive, as it may seem that the objective is to maximize the exploration of the robot. However, that is not true. The exploration in this context means that the agent selects actions that are not necessarily the actions expected to yield high reward. A typical state in this task may contain few actions that change the state (which is ideally always preferred) and the majority of actions that do not. Using more explorative policy will lead to selecting more actions that do not change the state, which is undesirable.

The PPO parameter that has direct influence on exploration is the entropy coefficient. Higher value keeps the policy more stochastic, promoting exploration. It was already discussed that excessive exploration of the agent is undesirable, so this value should be kept as low as possible. Stable-Baselines3 defaults this value to 0, so further tuning was not attempted. Another parameter that indirectly affects the exploration is learning rate, the complication its finetuning led to was described.

The final hyperparameters used for the PPO are listed in Table 4.1. It was experimented with various values of *rollout length*, the number of steps the agent collects before an update, and of *minibatch size*. Tampering with these led to a large training instability, which caused the robot to perform unoptimal actions and the model needed to be retrained.

4.5 The Physical Environment

The agent was trained to explore the user interface of an iPhone SE (2016). One of the actions of the RL agent was mapped to the Home button, ensuring

Parameter	Value
Learning rate	3×10^{-4}
Rollout length	2048
Minibatch size	64
Epochs per update	10
Discount factor (γ)	0.99
Clipping range (ε)	0.2
Entropy coefficient	0

Table 4.1 Summary of key PPO hyperparameters used for training.

that it could always return to Home screen and continue exploring. An example of a failure state, where the robot would get stuck otherwise, is an application that requires an internet connection. The application usually presents only a 'Retry' button that doesn't change the screen, as the device remains offline.

Using an operating system of iPhone, iOS, as a learning and testing environment has a high practical relevance. Its UI structure is consistent and relatively clear and can be used to highlight practical usability of the implemented methods. For a full scale, commercial testing, however, it would be beneficial to setup a dedicated user environment that provides direct feedback. Additionally, such environment could be used to train the agent in simulation, which would be much faster and could benefit from using parallel runs. The approaches, however, could remain the same.

Mapping of RL agent to the physical screen is straightforward. An action is an integer $a \in \{0, \dots, 375\}$. These form a 15×25 grid overlaid on the 50 mm × 88 mm screen, where each action corresponds a specific grid cell. The last action, 375, is reserved for the Home button.

For testing, a sparser environment with fewer applications was used. These applications excluded those that frequently misled the proximity-based detector of a new state, such as Camera, whose refocusing led the state detector to think that a new state was reached.

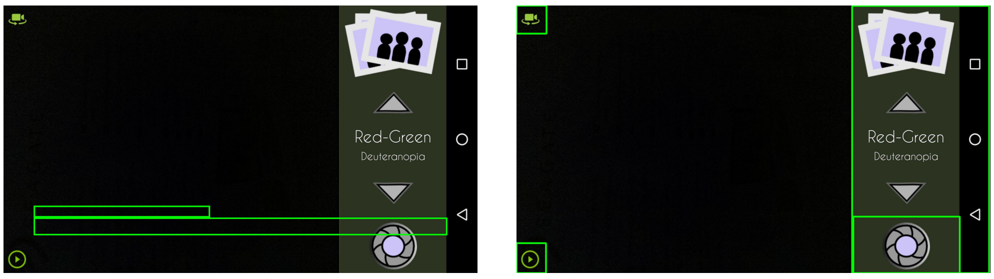
Chapter 5

Experiments

This chapter concludes this thesis with a summary of results, offers a discussion of their implications and outlines potential directions for future work.

5.1 YOLO Element Detection

The YOLOv5 model used for detection of clickable elements has shown a significant improvement during training. Key metrics displayed in Figure 5.2 are distribution focal loss and box loss, as these evaluate how well the model predicts positions and shapes of detected elements. The Distribution Focal Loss (DFL) treats bounding box localization as a classification task over discrete bins — the model produces a distribution over possible positions for each bounding box edge and DFL penalizes it based on the distance from the ground truth distribution. Box loss, as discussed in Section 3.1.1 typically uses some form of IoU-based function to quantify the difference between detected bounding box and ground truth. Both of these loss functions are decreasing, which signals improvement in model's prediction capabilities.



(a) Incorrectly labeled data. **(b)** YOLO-detected interactive elements.

Figure 5.1 YOLO generalization: relatively accurate detection achieved despite substantial label noise in the training data. Detected bounding boxes are green.

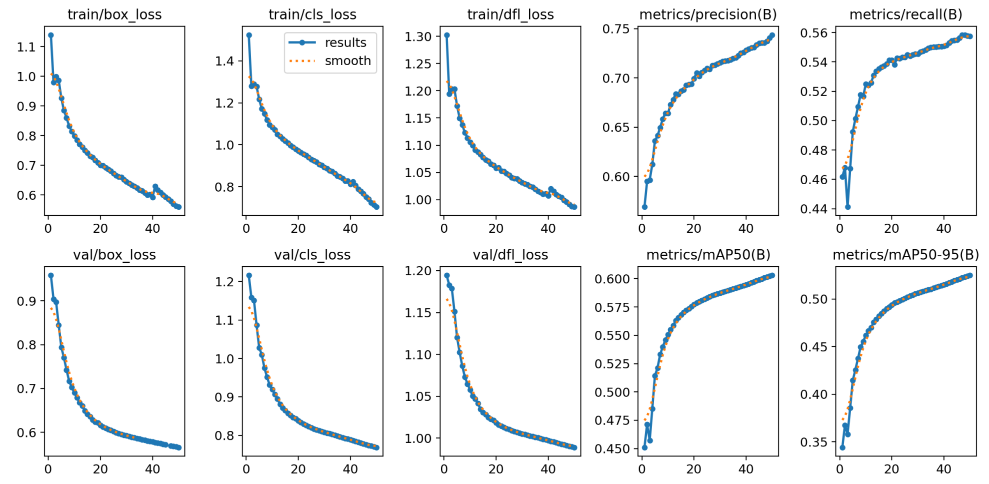


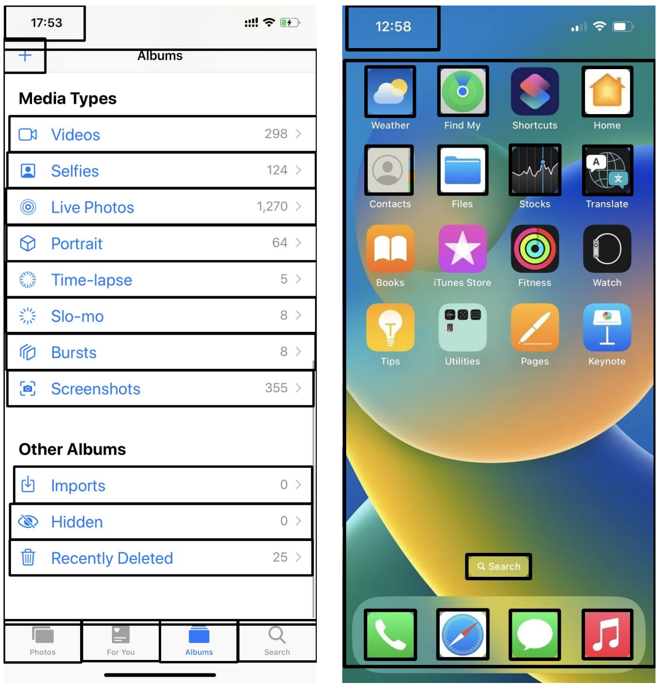
Figure 5.2 Training and validation metrics from the YOLOv5 model. The x-axis' represents training epochs. The metrics show improvement during training.

Figure 5.1 is an example of an improvement in generalization of the model. Despite the inaccuracies in the dataset annotations described in Section 4.2, the model learned to generalize and relatively correctly find the interactive elements. However, the figure still reveals some imperfections in the model's predictions.

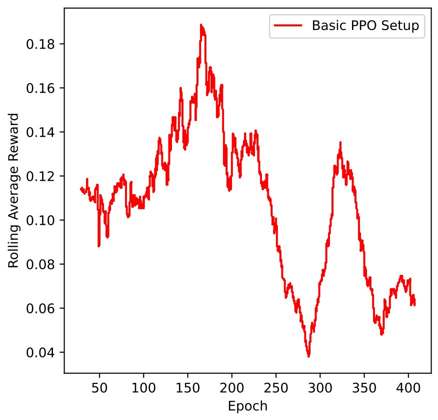
Measuring the method's portability to other mobile operating systems is limited by the lack of available datasets. Figure 5.3 shows that the method manages to detect the elements relatively correctly on some simpler screens, but fails on more complex ones. The model incorrectly detects the time as an interactive element and tends to detect the whole screen. The latter may be caused by advertisements often appearing in the training data, where the whole screen was clickable.

Some of the problems in this approach were already discussed, such as the necessity of training data and the requirement of high accuracy for practical use. Another problem is that the clickable elements may not always be distinguishable. For example, a gallery application might display a photo in full screen. Clicking on the right side of the photo, without any indication, could switch to the next photo in the gallery. Since there is no clickable element to be detected, this approach would fail.

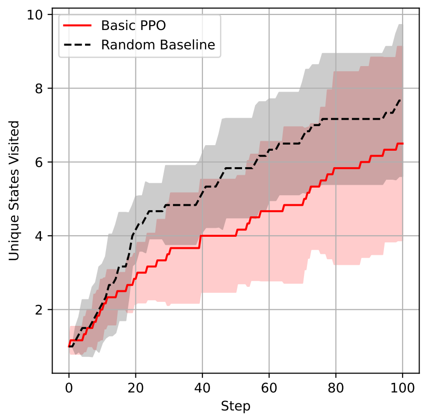
While YOLO was able to learn from noisy data and showed promising generalization, its requirement of a training dataset and the portability limitations resulting from it, as well as its reliance on visual indication of clickable elements, make this method impractical as a solution for the objective of this thesis.



■ **Figure 5.3** Generalization of YOLO to other mobile OS is limited. Even when the confidence threshold was reduced, several icons on the home screen remained undetected.



(a) The rolling average reward (averaged per steps using window size of 3000 training steps, one epoch is 100 steps) of the basic PPO setup.



(b) Cumulative discovered new states of RL agent using the basic PPO setup.

Figure 5.4 Training performance and exploration behaviour of the basic PPO setup.

5.2 Reinforcement Learning Methods

The rewards of reinforcement learning model were recorded during training. Because of the nature of how the reward is defined, this value also translates to how many new states the agent discovered on average. For example, an average of 0.1 means that the agent discovered 10 new states per 100 actions performed on average.

Due to a technical issue with servomotors, the robot's movement was slowed down. As a result, some tap actions were incorrectly registered as hold actions, allowing the robot to unintentionally select and remove applications from the environment. This likely contributed to the drop in rewards shown in Figure 5.4a.

Figure 5.4b shows that the performance of the policy actually decreased. This, as well as all the other graphs and results in this chapter, are based on an average of 6 traversals. The number of repetitions is limited by time constraints inherent to the real-world nature of the task, as the robot takes a long time to perform the actions.

The results suggest that the basic reinforcement learning setup isn't suitable for the task of this thesis. The rest of this section reflects on why this might be the case.

First of all, the iOS user interface is relatively sparse. Most of actions taken by the agent do not change the state. This system only rewards actions that not only result in state change, but also lead to a state that hasn't been previously visited. The values in Figure 5.4a do not capture how sparse this environment really is, as every 100 steps, the environment is reset, making the exploration task simpler — before an agent gets deeper in the exploration, the environment is reset again. A solution may be to make the episodes longer, but this would make the task harder and since the policy did not improve even on the 100 steps, it is unlikely that extending the episode length would lead to an improvement and it is certainly not addressing the core learning issue.

Another possible problem may be with the Home button. Since Home screen is the initial state, the action that corresponds to pressing a home button is never rewarded. Imagine an optimal exploration path through all states in the environment. Such path would require returning to the Home screen to switch between applications. When using the discretized 15×25 action space, at the beginning of training, the probability of selecting the Home button action is 1:375. Selecting this action near the end of an episode or before a long sequence without rewards would make this probability even smaller, until the agent starts to purposely avoid it.

These challenges highlight the need for a different implementation of the reward signal. A viable approach may be an intrinsic motivation. The following sections evaluate its effectiveness.

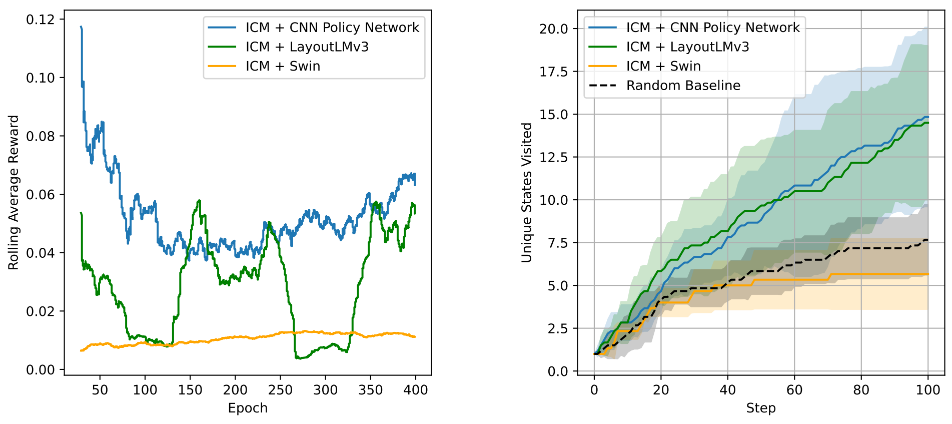
5.2.1 Intrinsically Motivated Learning

The ICM training encountered several difficulties. While some technical issues were mentioned in Section 4.1.1, a notable practical issue involved lighting conditions, as direct sunlight interfered with the learning process. The direct sunlight shining on the screen caused the camera to capture only a completely white image. The ICM quickly learned to predict this white image, so the rewards dropped. This is captured in Figure 5.5a. What is more interesting, the peaks of the rewards have grown and Figure 5.5b shows that the policy improved.

The reason may be that the policy was updated minimally due to advantage function values dropping near zero. Once the agent started receiving meaningful rewards again, the policy updates became more substantial again and allowed the performance to rapidly recover.

Figure 5.5a also displays an initial drop in ICM rewards, as the model becomes more familiar with the environment.

Another example of how the policy improves during training is depicted in Figure 5.6, which displays histograms of actions taken during training of the PPO model using LayoutLMv3 as a feature extractor. In the first 3,000 steps, the histogram shows high uniformity. After 30,000 steps of training, the histogram inclines toward the last action, which is clicking the Home button. This is expected behaviour — the agent may explore one application, and after there are no more available states to be discovered, the agent prefers to return to the Home screen. Note that some applications contained only



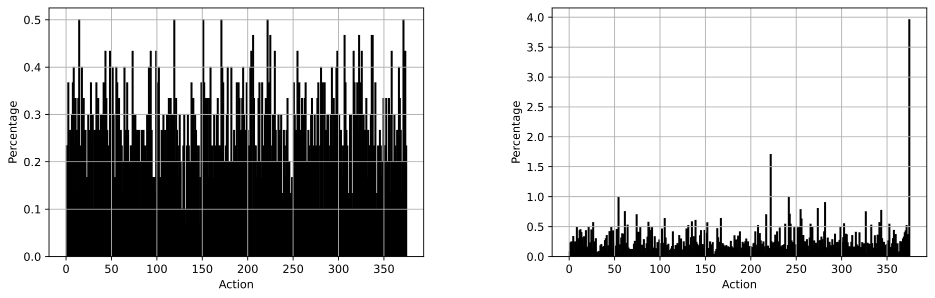
(a) Comparison of rolling averages (averaged per steps using window size of 3000 training steps, one epoch is 100 steps) of ICM rewards during training.

(b) Cumulative discovered new states of RL agents using ICM intrinsic motivation.

Figure 5.5 Training performance and exploration behaviour of ICM-based setup.

a single screen because the device was not connected to the internet, so an optimal policy would always select the Home button action in such state.

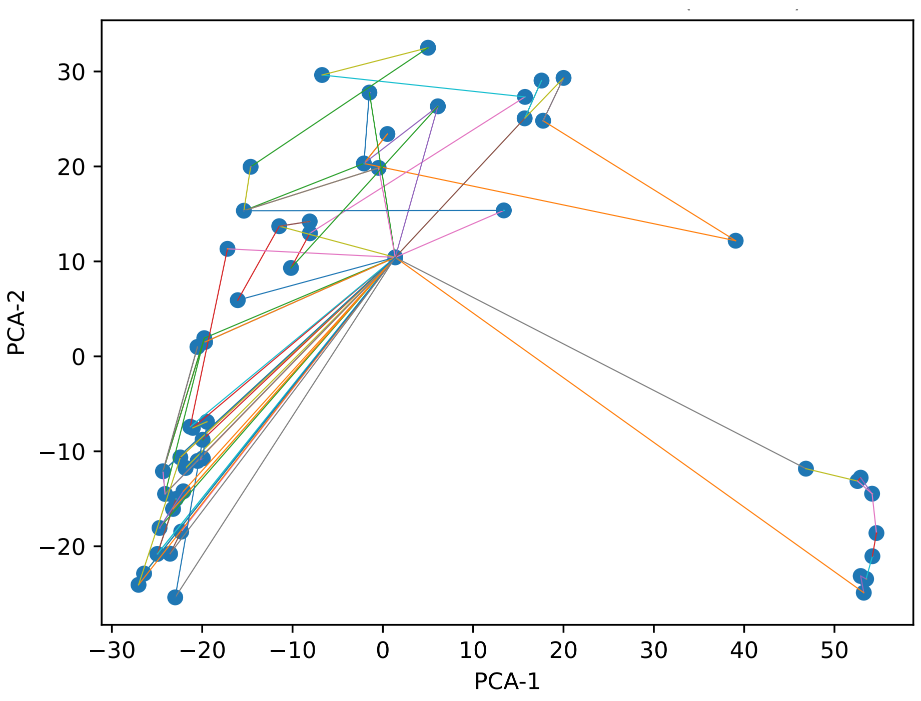
Figure 5.5b shows mean cumulative new states discovered in each step. A sample of six traversals was used, due to the high time cost of each evaluation. Standard deviation is included to capture variability in the outcomes. The graphs shows that both CNN and MLP over LayoutLMv3 features implementations of the policy network improved when using ICM as a sole reward signal and they both can achieve very similar performance. They also suggest that the exploration slows down more slowly than the random baseline, suggesting an ability to quickly exit states that yield low return (such as the screen



(a) First 3,000 steps.

(b) After 30,000 steps.

Figure 5.6 Histograms of actions taken during training of the PPO model using LayoutLMv3 and ICM reward.



■ **Figure 5.7** A PCA 2D projection of extracted features of the discovered unique states, edges highlight transistions between them.

displaying that the device is not connected to the internet).

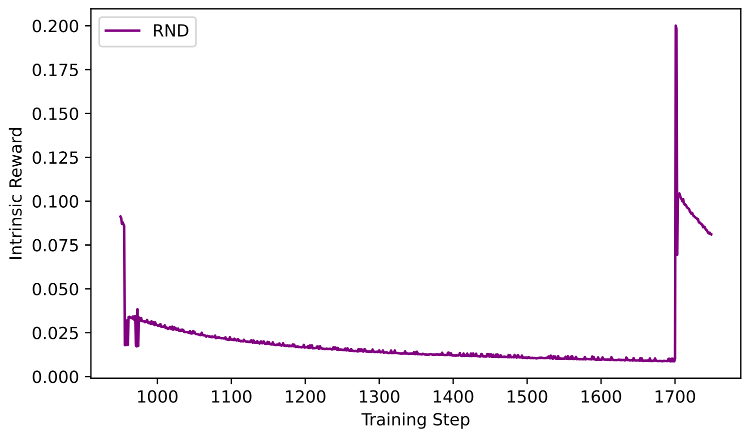
The CNN policy shows the highest explorative performance, achieving 14.83 ± 5.78 unique states and 32.33 ± 9.91 unique transitions explored on average, per 100 steps. Results when using MLP policy over LayoutLMv3-extracted features are comparable.

5.2.2 A Closer Look at the Feature Extraction

The intrinsic motivation methods used in this thesis use LayoutLMv3 feature extractor to obtain image features that preserve a strong sense of spatial layout. Figure 5.7 displays a PCA projection of extracted features of explored states in 500 steps into 2D space. The state with the most edges from it is the Home screen.

The states in the lower left corner mostly come from the Settings application. The layout used there is very consistent, so this is expected behaviour. The reason why these states often contain a transition to the Home screen is that iOS remembers the state the application was left in when the Home button was clicked. After relaunching, the applications return to the state they were in.

These findings suggest that the LayoutLMv3 feature extractor — even its pre-trained, non-fine-tuned variant — may be well-suited for the task.



■ **Figure 5.8** RND intrinsic rewards of an agent stuck in a state escapable only via the Home button.

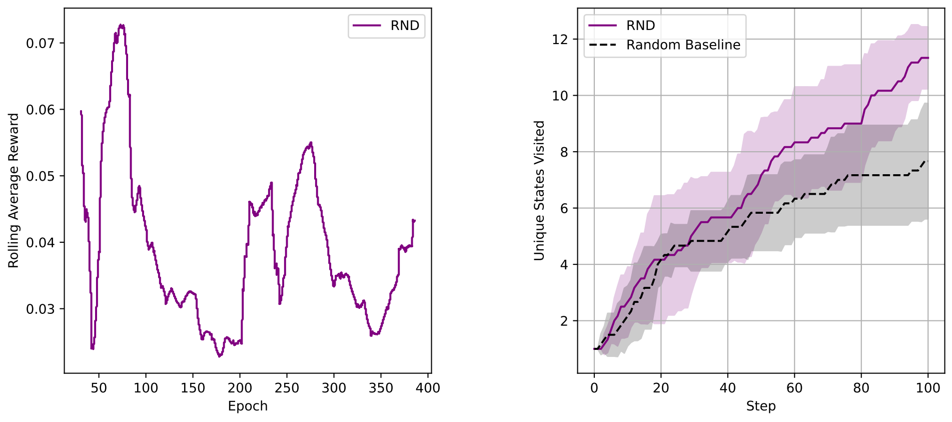
An agent was also trained using ICM with a more general Swin Transformer under the same settings. Despite showing an upward trend, the rewards in Figure 5.5a were much lower than with the LayoutLMv3 extractor, and the final model performed slightly worse than even the random baseline agent, as Figure 5.5b shows. A coefficient could be used to scale the rewards, but this result supports the expectation that the states are much closer in the feature space, which is not ideal for this task.

5.2.3 Random Network Distillation

Unlike ICM, the training with RND rewards was performed non-episodically. This may have led to agent being stuck in a state escapable only using the Home button, causing an extended period during which the reward gradually decreased as the predictor network improved in the approximation of the random network in that one state. Unlike in ICM, these periods could have lasted several hundreds training steps, causing the learning process to stagnate. This behaviour is demonstrated in Figure 5.8. Note that this is not an inherent characteristic of RND, rather a consequence of the high transition sparsity between actions in the environment.

This is also visible in Figure 5.9a, displaying intrinsic rewards during training. The initial drop and the sudden spike is caused by the agent first being stuck in a state, such as in Figure 5.8, and then suddenly escaping this state, allowing him to explore again.

Due to these issues, the RND reward may become very low for certain states. As a result, revisiting these may yield a small reward that won't lead to a significant update in the policy parameters. While this method is designed to promote exploration by rewarding novel states, this also introduces some



(a) Rolling average (averaged per steps using window size of 3000 training steps, one epoch is 100 steps) of RND reward during training.

(b) Cumulative discovered new states of RL agent using RND intrinsic motivation.

Figure 5.9 Training performance and exploration behaviour of the RND setup.

effects that are not desirable.

For instance, pressing the Home button leads to a transition to a Home screen, which still provides some reward. But because this state was visited frequently, the reward will be much lower, even though the action is essentially correct. The agent may then spend time on the Home screen, accumulating low rewards, until it transitions to a state that was previously visited, receiving little reward again. It was observed that some state transitions resulted in lower reward than the rewards obtained in the preceding state.

Despite this, the measurements and Figure 5.9b show that using RND leads to an improvement in policy, discovering 11.33 ± 1.21 states on average in an episode of 100 steps. This method even outperforms ICM in terms of number of unique transitions found, with 36.16 ± 7.81 discovered transitions on average, suggesting that RND can actually work as a driver for efficient learning.

The histogram of actions performed near the end of training in Figure 5.10 indicates that the agent becomes less inclined to return to the Home screen, which may explain the high number of discovered transitions. This behavior suggests that the agent is effectively exploring unique states within the application, such as various sub-menus and contextual windows.

5.3 Results

The results are summarized in Table 5.1. The measured values clearly demonstrate that using methods of intrinsic motivation can lead to an improvement in explorative performance of reinforcement learning agent. The naïve approach of using a reward of 1 for discovering a new state doesn't work, as it

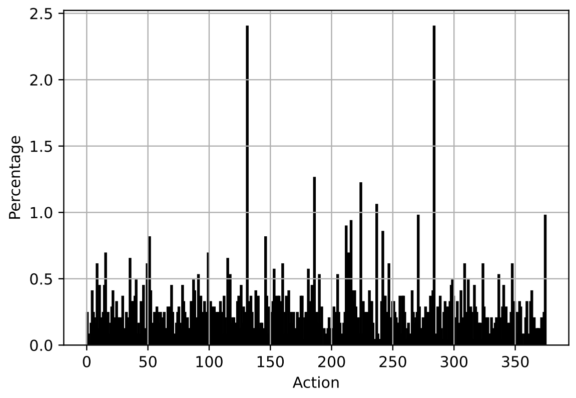


Figure 5.10 Histogram of actions performed near the end of RND training.

Model	Unique States	Transitions
Random baseline	7.67 ± 2.25	18.5 ± 6.66
Basic PPO setup	6.5 ± 2.88	15 ± 7.77
ICM + CNN Policy	$\mathbf{14.83 \pm 5.78}$	32.33 ± 9.91
ICM + MLP Policy (LayoutLMv3)	14.5 ± 4.93	31.5 ± 11.15
RND + CNN Policy	11.33 ± 1.21	$\mathbf{36.16 \pm 7.81}$

Table 5.1 Model performance comparison: average unique states visited and state transitions discovered in 100 steps (mean ± std over 6 traversals)

didn't show any signs of improvement compared to the random baseline.

5.3.1 Training Instability

Since the RL models learned slowly, it was experimented with adjusting the PPO parameters to make the process faster. In one of the training runs of the agent with RND rewards, the rollout length was reduced to make a policy update every 512 steps (originally 2048) and the batch size was reduced to 16 (from 64) to retain the number of actual updates that was performed. This caused the policy to degenerate and the agent to select mostly one action corresponding to a place in the lower left corner of the screen, as seen in action histogram in Figure 5.11.

This may have been caused by higher variance in the gradient estimates and insufficient exploration, which lead to premature convergence to a suboptimal policy.

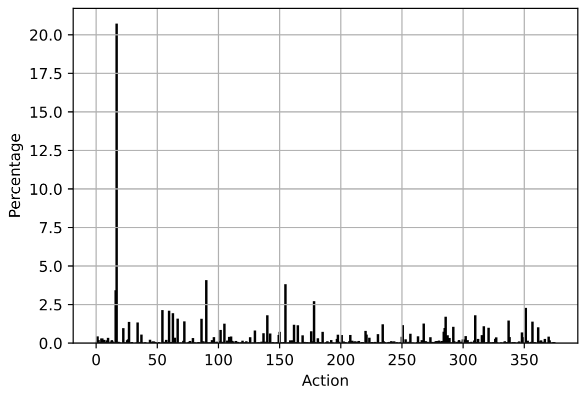


Figure 5.11 Histogram of actions selected by a degenerated policy caused by change in PPO training parameters.

5.3.2 Episodic vs. Non-Episodic Learning

Both ICM and RND employ different strategy of splitting the traversals into episodes. The agent used in the original ICM paper [41] is tested on a Doom 3-D navigation task and its episodes are terminated after 2100 time steps, or when the agent finds the goal and the objective is accomplished. The RND paper [42] states that treating the problem as non-episodic resulted in better exploration, mostly argumenting with the fact that truncating the episode causes the agent to work with incorrect intrinsic returns, similar to what was discussed in Section 5.2.

The RND paper also references [40], where it is argued that using '`done`' signal at the end of the episode can cause the reward signal to leak information about the environment and bias the agent. The agent can then learn to perform well in the span of an episode, but its performance may drastically drop after.

A simplified example that breaks this task is an app with exactly 100 states, where each tap leads to a new, unseen state at random, and resets after all are discovered. The basic PPO approach would fail, achieving maximum reward per episode without any generalization. ICM would struggle as well, since it doesn't handle stochastic transitions. RND only uses the next state, so it could handle this scenario. However, since most environments are less stochastic, this example won't practically occur and using shorter episodes may lead to a generalization and a faster training.

To evaluate this, the ICM-trained agent was run for 500 steps, that is 5× longer than the episodes the model was trained on. It managed to discover 55 unique states and 122 unique transitions. A PCA map of the states it discovered is shown in Figure 5.7. For comparison, a randomly initialized agent was run for the same duration and discovered 34 unique states and 76 unique transitions. This yields a ratio of 1.62 between the methods in terms of unique states visited, and 1.61 for unique transitions. Compared to the results in Table 5.1, where the ratios are 1.93 and 1.75, respectively, the performance slightly decreases, but the ICM agent still maintains a clear advantage.

5.4 Future Work

This work focused on integrating the intrinsic motivation into a system intended for an effective exploration of an interface of a touchscreen device. The relatively slow operation of the real-world systems compared to environments typically employed in reinforcement learning research remains a key challenge. More significant training progress would require long training time, which is impractical. The effects of this can be mitigated by refining other components of the system.

A possible area of improvement that was mentioned is reducing the action space. A suitable approach could be the Action Elimination Network [59]. It is trained to filter out actions that are unlikely to be useful, such as those that do not lead to meaningful state changes. An interesting experiment may be to measure not only whether the agent's policy improved, but also if the ratio of actions that led to a state change increased.

In the future, the interface could be modeled using a continuous action space. Methods such as Deep Deterministic Policy Gradient (DDPG) [67], its successor Twin Delayed DDPG (TD3) [68] or Soft Actor-Critic (SAC) [69] are designed for such spaces. The problem of these methods and generally of modern reinforcement learning is a low sample efficiency. A lot of newer methods address this, such as Randomized Ensembled Double Q-Learning (REDQ) [70] that promises a greater sample efficiency by updating the critic multiple times per environment step. How these methods would be implemented and whether this would lead to an improvement would need to be determined through further research.

The methods used in this work relied on advanced feature extractors. The performance of the methods presented here may be improved by collecting relevant data and finetuning the feature extractors. This may seem similar to the approach of element detection that was dismissed, but unlike in that method, finetuning feature extractors can be done in a semi-supervised manner.

One of the steps in the distant future could be completing the action space by allowing the robot to perform other actions, such as double tap, swipe or hold. These actions are supported by AIVA. An interestingly defined structured, combinatorial action space is in [48]. It allows each action to consist of multiple selections, such as action type, action target, etc. This approach may be worth investigating as a possible method of incorporating the other actions.

Conclusion

The objective of this work was to explore intrinsic motivation as a driver for reinforcement learning in a system designed to explore a complex user interface. It was shown that a simple, naïve approach is not sufficient and that using intrinsic motivation can lead to an improvement in performance of the explorative agent.

An alternative approach of using YOLO model to detect interactive elements was also evaluated. This approach was deemed impractical due to its reliance on a labeled training dataset, limited portability across platforms, and dependency on visually distinctive interactive elements.

Both intrinsic motivation techniques explored in this thesis — Intrinsic Curiosity Module and Random Network Distillation — led to an improvement and enabled the agent to perform more efficient exploration.

However, the ultimate end goal, building a system that, after a period of time, learns to efficiently explore the complex system of a touch-screen device is far from done. The performance is far from optimal, discretized action space is too large and does not contain all the actions, and the learning progresses slowly.

Despite these challenges, this work represents a meaningful step toward this ultimate end goal and makes it possible for an autonomous agent to improve in the task of navigation and understanding a complex system.

Bibliography

1. Y SOFT CORPORATION. *Reliable Test Automation Done Right*. 2025. Available also from: https://www.ysoft.com/aiva. Accessed: 2025-03-19.
2. WHITTINGTON, James C.R.; BOGACZ, Rafal. Theories of Error Back-Propagation in the Brain. *Trends in Cognitive Sciences*. 2019, vol. 23, no. 3, pp. 235–250. ISSN 1364-6613. Available from DOI: https://doi.org/10.1016/j.tics.2018.12.005.
3. LÖWEL, Siegrid; SINGER, Wolf. Selection of intrinsic horizontal connections in the visual cortex by correlated neuronal activity. *Science*. 1992, vol. 255, no. 5041, pp. 209–212. Available from DOI: https://doi.org/10.1126/science.1372754.
4. HEBB, Donald O. *The organization of behavior: A neuropsychological theory*. Psychology press, 2005. Available from DOI: https://doi.org/10.4324/9781410612403.
5. SUTTON, Richard S.; BARTO, Andrew G. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018. ISBN 0262039249. Available also from: http://incompleteideas.net/book/RLbook2020.pdf. Accessed 2025-04-06.
6. HRABAK, Pavel. *Discrete-time Markov Chains* [University Lecture]. 2024. Available also from: https://courses.fit.cvut.cz/NI-VSM/lectures/files/NI-VSM-Lec-14-Slides.pdf. Accessed 2025-04-05. (In Czech).
7. MARKOV, Andrey A. Extension of the Limit Theorems of Probability Theory to a Sum of Variables Connected in a Chain. *The Notes of the Imperial Academy of Sciences of St. Petersburg, VIII Series, Physio-Mathematical College*. 1907, vol. 22, no. 9. (In Russian).
8. SILVER, David. *Lectures on Reinforcement Learning* [University Course]. 2015. Available also from: https://www.davidsilver.uk/teaching/. Accessed 2025-04-05.

9. BELLMAN, Richard. *Dynamic Programming*. 1st ed. Princeton, NJ, USA: Princeton University Press, 1957.

10. PUTERMAN, Martin L. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 1994. Available from DOI: `https://doi.org/10.1002/9780470316887`.

11. CORMEN, Thomas H. et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN 0262033844.

12. HOWARD, Ronald A. *Dynamic Programming and Markov Processes*. Cambridge, USA: The MIT Press, 1960.

13. SZEPESVARI, Csaba. *Algorithms for Reinforcement Learning*. Morgan and Claypool Publishers, 2010. ISBN 1608454924.

14. SINGH, Satinder P.; SUTTON, Richard S. Reinforcement learning with replacing eligibility traces. *Mach. Learn.* 1996, vol. 22, no. 1–3, pp. 123–158. ISSN 0885-6125. Available from DOI: `10.1007/BF00114726`.

15. SUTTON, Richard Stuart. *Temporal credit assignment in reinforcement learning*. 1984. PhD thesis. University of Massachusetts Amherst.

16. SUTTON, Richard S. Learning to Predict by the Methods of Temporal Differences. *Mach. Learn.* 1988, vol. 3, no. 1, pp. 9–44. ISSN 0885-6125. Available from DOI: `https://doi.org/10.1023/A:1022633531479`.

17. RUMMERY, G.; NIRANJAN, Mahesan. On-Line Q-Learning Using Connectionist Systems. *Technical Report CUED/F-INFENG/TR 166*. 1994.

18. SINGH, Satinder et al. Convergence Results for Single-Step On-Policy Reinforcement-Learning Algorithms. *Machine Learning*. 2000, vol. 38, no. 3, pp. 287–308. ISSN 1573-0565. Available from DOI: `https://doi.org/10.1023/A:1007678930559`.

19. WATKINS, Christopher J. C. H. *Learning from delayed rewards*. 1989. PhD thesis. University of Cambridge, England.

20. WATKINS, Christopher J. C. H.; DAYAN, Peter. Q-learning. *Machine Learning*. 1992, vol. 8, no. 3, pp. 279–292. ISSN 1573-0565. Available from DOI: `https://doi.org/10.1007/BF00992698`.

21. TROMP, John. *Chess Position Ranking [online]* [`https://github.com/tromp/ChessPositionRanking`]. GitHub, 2022. Accessed: 2025-04-15.

22. SILVER, David et al. Mastering the game of Go without human knowledge. *Nature*. 2017, vol. 550, no. 7676, pp. 354–359. ISSN 1476-4687. Available from DOI: `https://doi.org/10.1038/nature24270`.

23. MNIH, Volodymyr et al. *Playing Atari with Deep Reinforcement Learning*. 2013. Available from DOI: `https://doi.org/10.48550/arXiv.1312.5602`.

24. BARNARD, Etienne. Temporal-Difference Methods and Markov Models. *Systems, Man and Cybernetics, IEEE Transactions on.* 1993, vol. 23, pp. 357–365. Available from DOI: `https://doi.org/10.1109/21.229449`.

25. BAIRD, Leemon C. Residual algorithms: reinforcement learning with function approximation. In: *Proceedings of the Twelfth International Conference on International Conference on Machine Learning.* Tahoe City, California, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 30–37. ICML'95. ISBN 1558603778.

26. ZHANG, Shangtong; YAO, Hengshuai; WHITESON, Shimon. Breaking the Deadly Triad with a Target Network. *CoRR.* 2021, vol. abs/2101.08862. Available from DOI: `https://doi.org/10.48550/arXiv.2101.08862`.

27. WILLIAMS, Ronald J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning.* 1992, vol. 8, no. 3, pp. 229–256. ISSN 1573-0565. Available from DOI: `https://doi.org/10.1007/BF00992696`.

28. WITTEN, Ian H. An adaptive optimal controller for discrete-time Markov environments. *Information and Control.* 1977, vol. 34, no. 4, pp. 286–295. ISSN 0019-9958. Available from DOI: `https://doi.org/10.1016/S0019-9958(77)90354-0`.

29. KONDA, Vijay; TSITSIKLIS, John. Actor-Critic Algorithms. In: *Advances in Neural Information Processing Systems.* MIT Press, 1999, vol. 12, pp. 1008–1014. Available also from: `https://papers.nips.cc/paper_files/paper/1999/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf`. Accessed: 2025-04-20.

30. SUTTON, Richard S et al. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In: *Advances in Neural Information Processing Systems.* MIT Press, 1999, vol. 12, pp. 1057–1063. Available also from: `https://proceedings.neurips.cc/paper_files/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf`. Accessed: 2025-04-21.

31. KAKADE, Sham M. A Natural Policy Gradient. In: *Advances in Neural Information Processing Systems.* MIT Press, 2001, vol. 14, pp. 1531–1538. Available also from: `https://proceedings.neurips.cc/paper_files/paper/2001/file/4b86abe48d358ecf194c56c69108433e-Paper.pdf`. Accessed: 2025-04-21.

32. PETERS, Jan; VIJAYAKUMAR, Sethu; SCHAAL, Stefan. Natural Actor-Critic. In: *Machine Learning: ECML 2005.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 280–291. ISBN 978-3-540-31692-3.

33. SCHULMAN, John et al. Trust Region Policy Optimization. In: *Proceedings of the 32nd International Conference on Machine Learning*. Lille, France: PMLR, 2015, vol. 37, pp. 1889–1897. Proceedings of Machine Learning Research. Available also from: `https://proceedings.mlr.press/v37/schulman15.html`. Accessed: 2025-04-22.

34. HUNTER, David R; LANGE, Kenneth. A Tutorial on MM Algorithms. *The American Statistician*. 2004, vol. 58, no. 1, pp. 30–37. Available from DOI: `https://doi.org/10.1198/0003130042836`.

35. KAKADE, Sham; LANGFORD, John. Approximately Optimal Approximate Reinforcement Learning. In: *Proceedings of the Nineteenth International Conference on Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002, pp. 267–274. ICML '02. ISBN 1558608737.

36. KULLBACK, S.; LEIBLER, R. A. On Information and Sufficiency. *The Annals of Mathematical Statistics*. 1951, vol. 22, no. 1, pp. 79–86. Available from DOI: `https://doi.org/10.1214/aoms/1177729694`.

37. SCHULMAN, John et al. *High-Dimensional Continuous Control Using Generalized Advantage Estimation*. 2018. Available from arXiv: `1506.02438 [cs.LG]`.

38. SCHULMAN, John et al. Proximal Policy Optimization Algorithms. 2017. Available from DOI: `https://doi.org/10.48550/arXiv.1707.06347`.

39. PASZKE, Adam et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. Available also from: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`. Accessed: 2025-04-26.

40. BURDA, Yuri et al. *Large-Scale Study of Curiosity-Driven Learning*. 2018. Available from DOI: `https://doi.org/10.48550/arXiv.1808.04355`.

41. PATHAK, Deepak et al. *Curiosity-driven Exploration by Self-supervised Prediction*. 2017. Available from DOI: `https://doi.org/10.48550/arXiv.1705.05363`.

42. BURDA, Yuri et al. *Exploration by Random Network Distillation*. 2018. Available from DOI: `https://doi.org/10.48550/arXiv.1810.12894`.

43. KENDALL, Alex; GAL, Yarin. What Uncertainties Do We Need in Bayesian Deep Learning for Computer Vision? In: *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2017, vol. 30, pp. 5574–5584. Available also from: `https://proceedings.neurips.cc/paper_files/paper/2017/file/2650d6089a6d640c5e85b2b88265dc2b-Paper.pdf`. Accessed: 2025-04-26.

44. SICILIANO, B.; KHATIB, O. *Springer Handbook of Robotics*. Springer Berlin Heidelberg, 2008. ISBN 9783540239574. Available from DOI: `https://doi.org/10.1007/978-3-540-30301-5`.

45. REDMON, Joseph et al. *You Only Look Once: Unified, Real-Time Object Detection*. 2016. Available from DOI: `https://doi.org/10.48550/arXiv.1506.02640`.

46. KHANAM, Rahima; HUSSAIN, Muhammad. *What is YOLOv5: A deep look into the internal features of the popular object detector*. 2024. Available from DOI: `https://doi.org/10.48550/arXiv.2407.20892`.

47. TIAN, Yunjie; YE, Qixiang; DOERMANN, David. *YOLOv12: Attention-Centric Real-Time Object Detectors*. 2025. Available from DOI: `https://doi.org/10.48550/arXiv.2502.12524`.

48. VINYALS, Oriol et al. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*. 2019, vol. 575, no. 7782, pp. 350–354. ISSN 1476-4687. Available from DOI: `https://doi.org/10.1038/s41586-019-1724-z`.

49. OPENAI et al. *Dota 2 with Large Scale Deep Reinforcement Learning*. 2019. Available from DOI: `https://doi.org/10.48550/arXiv.1912.06680`.

50. LOWE, D.G. Object recognition from local scale-invariant features. In: *Proceedings of the Seventh IEEE International Conference on Computer Vision*. 1999, vol. 2, 1150–1157 vol.2. Available from DOI: `10.1109/ICCV.1999.790410`.

51. LECUN, Yann et al. Handwritten Digit Recognition with a Back-Propagation Network. In: *Advances in Neural Information Processing Systems*. Morgan-Kaufmann, 1989, vol. 2, pp. 396–404. Available also from: `https://proceedings.neurips.cc/paper_files/paper/1989/file/53c3bce66e43be4f209556518c2fcb54-Paper.pdf`. Accessed: 2025-04-28.

52. GU, Jiuxiang et al. *Recent Advances in Convolutional Neural Networks*. 2017. Available from arXiv: `1512.07108 [cs.CV]`.

53. KRIZHEVSKY, Alex; SUTSKEVER, Ilya; HINTON, Geoffrey E. ImageNet Classification with Deep Convolutional Neural Networks. In: *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2012, vol. 25, pp. 1097–1105. Available also from: `https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf`.

54. DOSOVITSKIY, Alexey et al. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. 2021. Available from DOI: `https://doi.org/10.48550/arXiv.2010.11929`.

55. VASWANI, Ashish et al. Attention is All you Need. In: GUYON, I. et al. (eds.). *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2017, vol. 30, pp. 5998–6008. Available also from: `https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf`. Accessed: 2025-04-28.

56. LIU, Ze et al. *Swin Transformer: Hierarchical Vision Transformer using Shifted Windows*. 2021. Available from DOI: `https://doi.org/10.48550/arXiv.2103.14030`.

57. HUANG, Yupan et al. LayoutLMv3: Pre-training for Document AI with Unified Text and Image Masking. In: *Proceedings of the 30th ACM International Conference on Multimedia*. Lisboa, Portugal: Association for Computing Machinery, 2022, pp. 4083–4091. MM '22. ISBN 9781450392037. Available from DOI: `https://doi.org/10.1145/3503161.3548112`.

58. DULAC-ARNOLD, Gabriel et al. *Deep Reinforcement Learning in Large Discrete Action Spaces*. 2016. Available from DOI: `https://doi.org/10.48550/arXiv.1512.07679`.

59. ZAHAVY, Tom et al. Learn What Not to Learn: Action Elimination with Deep Reinforcement Learning. In: *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2018, vol. 31, pp. 3562–3573. Available also from: `https://proceedings.neurips.cc/paper_files/paper/2018/file/645098b086d2f9e1e0e939c27f9f2d6f-Paper.pdf`. Accessed: 2025-05-01.

60. OPENAPI GENERATOR CONTRIBUTORS. *OpenAPI Generator [online]*. GitHub, 2025. Available also from: `https://github.com/OpenAPITools/openapi-generator`. Accessed: 2025-05-01.

61. JOCHER, Glenn. *Ultralytics YOLOv5*. 2020. Version 7.0. Available from DOI: `10.5281/zenodo.3908559`.

62. DEKA, Biplab et al. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In: *Proceedings of the 30th Annual Symposium on User Interface Software and Technology*. 2017. UIST '17.

63. LEWIS, D. et al. Building a test collection for complex document information processing. In: *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. Seattle, Washington, USA: Association for Computing Machinery, 2006, pp. 665–666. SIGIR '06. ISBN 1595933697. Available from DOI: `https://doi.org/10.1145/1148170.1148307`.

64. DENG, Jia et al. Imagenet: A large-scale hierarchical image database. In: *2009 IEEE conference on computer vision and pattern recognition*. IEEE, 2009, pp. 248–255.

65. RAFFIN, Antonin et al. Stable-Baselines3: Reliable Reinforcement Learning Implementations. *Journal of Machine Learning Research*. 2021, vol. 22, no. 268, pp. 1–8. Available also from: `http://jmlr.org/papers/v22/20-1364.html`.

66. MNIH, Volodymyr et al. Human-level control through deep reinforcement learning. *Nature*. 2015, vol. 518, no. 7540, pp. 529–533. ISSN 1476-4687. Available from DOI: `https://doi.org/10.1038/nature14236`.

67. LILLICRAP, Timothy P. et al. *Continuous control with deep reinforcement learning*. 2019. Available from arXiv: `1509.02971 [cs.LG]`.

68. FUJIMOTO, Scott; HOOF, Herke van; MEGER, David. *Addressing Function Approximation Error in Actor-Critic Methods*. 2018. Available from arXiv: `1802.09477 [cs.AI]`.

69. HAARNOJA, Tuomas et al. *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*. 2018. Available from arXiv: `1801.01290 [cs.LG]`.

70. CHEN, Xinyue et al. *Randomized Ensembled Double Q-Learning: Learning Fast Without a Model*. 2021. Available from arXiv: `2101.05982 [cs.LG]`.

Contents of the attachment

```
│ readme.txt ........................ a brief description of the attachments
└── src
    ├── thesis ..................................... thesis LaTeX source files
    └── impl ............................ source codes for the implementation
        ├── yolo ................................training of the YOLO model
        ├── aiva_explore_mlp ................... RL agent using MLP policy
        └── aiva_explore_cnn ................... RL agent using CNN policy
└── text ..................................................... thesis text
    └── thesis.pdf .............................. thesis text in PDF format
```