Comparing the Impact of Intrinsic Motivation Techniques on On-Policy and Off-Policy Reinforcement Learning Methods in Atari Games

Ryan Partridge

MSc in Machine Learning and Autonomous Systems

The University of Bath

2022

Comparing the Impact of Intrinsic Motivation Techniques on On-Policy and Off-Policy Reinforcement Learning Methods in Atari Games

Submitted by

Ryan Partridge

For the degree of MSc in Machine Learning and Autonomous Systems

University of Bath

2022

Copyright

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Master of Science in Machine Learning and Autonomous Systems in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Abstract

Intrinsic Motivation (IM) has become a popular area of research in Reinforcement Learning over the past decade, with numerous studies presenting new techniques for improving agent exploration focusing on sparse reward environments. However, to our knowledge, there are finite experiments relating to the effects of IM techniques on policy-based variants. Our research bridges that gap by examining two popular IM techniques, Curiosity and Empowerment. We explore their impact on two foundational policy-based models, Deep Q-Networks (off-policy) and Proximal Policy Optimization (on-policy), analytically and visually using three Atari 2600 games, Space Invaders, Q*bert, and Montezuma's Revenge. We demonstrate our analysis using 18 models, six without IM, six with Curiosity and six with Empowerment, incorporating both types of models across the three environments.

Contents

List of Figures

List of Tables

Acknowledgements

Thank you to my dissertation supervisors, **Dr. Paola Bruscoli**, **Dan Beechey**, and **Tom Cannon**, for their invaluable guidance and feedback throughout the project. Additionally, I want to thank my friends and family for being supportive throughout my academic journey.

Chapter 1

Introduction

Artificial General Intelligence (AGI) has been a topic of debate since the Turing test in the 1950s (Turing, 1950) and continues to be an active research area. Over the past two decades, Reinforcement Learning (RL), a category of Artificial Intelligence (AI), has shown great promise in simulating human-like intelligence (Badia et al., 2020; Burda et al., 2018b) and could be the door to unlocking AGI due to its primary objective - maximising reward. Most RL research provides new methods and techniques for solving problems, such as Intrinsic Motivation (IM), which builds atop the original framework created by Sutton and Barto (1998). Within this framework, they developed foundational methods and implemented a concept known as policy-based agents. Divided into two types, on and off-policy, these agents utilise a behaviour policy to determine their actions within a given state and aim to learn a target policy that provides feedback on its performance in the form of rewards. Off-policy agents master a different behaviour policy from the target policy, and on-policy uses the same one.

Over the past decade, policy-based research has focused on off-policy agents intending to optimise and improve their efficiency during training (Fujimoto, Meger and Precup, 2019; Thomas and Brunskill, 2016; Munos et al., 2016), and to our knowledge, there is limited growth to research regarding on-policy agents. Notably, there has been a vast effort to combine both methods, predominantly to improve sample efficiency (Fakoor, Chaudhari and Smola, 2019; Gu et al., 2017). Comparative to IM research, we aim to tackle a different problem, agent environment exploration. With IM being a relatively new field, there is a shortage of research that explores how its techniques explicitly affect on-policy and off-policy agents.

This project focuses on tackling this limitation with a specific focus on two IM methods, curiosity and empowerment (Pathak et al., 2017; Klyubin, Polani and Nehaniv, 2005). We combine these techniques with two popular algorithms, Deep Q-Network (DQN, Mnih et al., 2013) and Proximal Policy Optimization (PPO, Schulman et al., 2017), to simulate each type of policy-based agent. The selected algorithms provide an intuitive degree of complexity while being robust for solving challenging problems, proving suitable benchmarks for the IM methods. Furthermore, we considered other Actor-Critic approaches (DDPG, A3C, and A2C) but identified them as too complex, ultimately deterring us from effectively analysing the selected IM techniques. Similarly, other IM techniques exist, such as surprise-based and competence-based motivation (Achiam and Sastry, 2017; Stout and Barto, 2010). While we aim to explore these in the future, we focused on selecting IM methods based on popularity and personal interest.

Our investigation explores how each IM method extends the algorithms numerically and impacts the agent's behaviour during simulation. Some of the questions we aim to solve in this paper include:

1. What are the modifications to the reward process, and how does this affect agent learning?
2. What impact do the techniques have on the algorithm's performance?
3. Do the agents take different actions with IM or use new strategies? If so, why?
4. How do the methods differ on-policy vs off-policy during and after agent learning?

To analyse and evaluate each algorithm's performance, we carefully selected three environments, Space Invaders, Q*bert, and Montezuma's Revenge, based on Mnih et al. (2015) analysis of a traditional DQNs performance on 49 Atari 2600 games. Bellemare et al. (2013) proposed the Atari suite in 2013, defining a series of trials for RL algorithms that also prove challenging for real players. The Atari suite quickly became a foundational benchmark for comparing RL agents against human-level performance and provides an effective tool for analysing multiple IM methods. The environments require different actions to complete their challenges, each increasing in difficulty based on the capability to obtain rewards. Space Invaders is the simplest of the three, requiring the agent to eliminate aliens approaching its spaceship as fast as possible. Q*bert, a medium difficulty, focuses on changing colours of squares on a pyramid while avoiding enemies, and Montezuma's Revenge presents the most challenging of the three, needing deep exploration of rooms while avoiding obstacles and moving opponents.

We find this project incredibly stimulating due to the nature of traditional RL methods, exclusively using extrinsically motivated rewards. Unfortunately, this limitation has proven counter-intuitive in complex environments like Montezuma's Revenge due to sparse rewards and minimal exploration. This research intends to (1) expand the academic literature in the field and (2) provide intuition on how IM methods affect on-policy and off-policy agents.

In the remainder of this study, we analyse related literature in chapter 2; explore our methodology strategies in chapter 3; examine the application's design and development in chapter 4; present our research findings in chapter 5; and lastly, discuss the project conclusions and future work in chapter 6.

1.1 Aims and Objectives

The project aims to analyse and compare the differences between the selected IM methods applied to on-policy (PPO) and off-policy (DQN) algorithms. The chosen Atari games assist this aim by acting as a tool to interpret how IM techniques affect the algorithms in various environments. To further understand this aim, we divide it into multiple objectives.

Objective 1: analyse the fundamental components of on-policy and off-policy algorithms and determine what makes them different. Additionally, build models of the algorithms programmatically to test them within the selected environments.

Objective 2: incorporate IM techniques into the models, examining the analytical modifications and their effects on the policy-based models' performance and learning process. The IM methods are combined with the extrinsic reward for both algorithms and are tested separately across all game environments.

Objective 3: analyse and evaluate the performance of the 18 model variants across each game statistically and visually, using the same hyperparameters where applicable. The models consist of three sets of six, one for extrinsic reward (models without IM) and the remaining two for the respective IM methods, curiosity, and empowerment.

Chapter 2

Related Work

Humans and animals have been intrinsically and extrinsically motivated to complete tasks in their daily lives for millennia. By psychological definition, extrinsic motivation relates to being moved to do something because of a specific rewarding outcome, while intrinsic motivation is doing something because it is inherently enjoyable (Ryan and Deci, 2000). The former was explored computationally in the early 1980s, creating what is known today as the foundation for RL. Sutton and Barto (1998), two founding members of RL, formulated a generalised framework heavily inspired by the human and animal reward process. The framework focuses on agents that learn through an iterative approach based on extrinsically motivated rewards generated by explicitly engineered reward functions that interact with an external environment. These rewards correspond to primary benefits, such as ones hard-wired evolutionarily for reproductive success (Singh, Lewis and Barto, 2009) and are hypothesised as a fundamental driver of behaviour that exhibits most, if not all, abilities studied in natural and artificial intelligence (Silver et al., 2021).

One of the earliest success stories of RL is TD-gammon. A model-free agent based on Sutton and Barto's on-policy TD(λ) algorithm combined with a multi-layered perceptron (MLP), successfully achieving a super-human level of play on the board game backgammon (Tesauro, 1995). It was the first of its kind and generated interest in using RL to solve classic games. However, attempts to follow its example in similar games such as chess, Go, and checkers failed until 20 years later (Silver et al., 2016). Pollack and Blair (1996) replicated similar success to TD-gammon without the same techniques. Hypothesising its success was due to an inherent bias from the dynamics of backgammon, and the co-evolutionary setup of its training, deeming it a rare case that only works in backgammon.

In 2013 RL had a breakthrough with Mnih et al. (2013) Deep Q-Network (DQN), a fundamental off-policy algorithm regularly used in complex sequential decision-making problems. They combined deep neural networks, Q-Learning, raw pixel inputs, and a technique called experience replay to solve a small portion of Atari games at human-level performance, opening numerous possibilities for RL research. Comparative to standard online Q-Learning illustrated by Sutton and Barto (1998), their approach proved more data-efficient with lower variance during TD updates and enabled smoother learning. Several improvements were created to increase its speed and stability, such as Double Q-Learning, Dueling Networks and Prioritised Experience Replay (van Hasselt, Guez and Silver, 2015; Wang et al., 2016; Schaul et al., 2016). Four years later, Hessel et al. (2017) combined six extensions, including the three previously mentioned, to create a more advanced version of the algorithm - the Rainbow DQN. These improvements

obtained state-of-the-art performance on 40 of 57 Atari 2600 games. While their achievements showed great promise for RL, they also expressed some hurdles, such as sparse rewards, that still need to be addressed.

DQNs were not the only algorithm to advance the field. Actor-Critic methods were shortly introduced after its success, expanding its ideas to continuous control problems (Lillicrap et al., 2019). These methods quickly began identifying similar difficulties to DQNs. One of them is the design of reward functions, referred to as reward shaping, where agents 'game' their reward function, ultimately ignoring the desired objective. For example, a cleaning robot earns reward for not seeing any mess and closes its eyes rather than cleaning anything. Or the robot is instead rewarded for cleaning messes and intentionally creates more of them to earn a greater reward (Amodei et al., 2016). Another problem involves environments with sparse or scattered rewards, primarily seen in large state spaces, where agents cannot effectively learn because they have not observed enough reward signals to reinforce their actions (Zai and Brown, 2020). These two challenges become a common issue when tackling complex environments with extrinsically motivated RL methods.

Success in intrinsic motivation (IM) techniques has proven valuable in mitigating these challenges. One example is curiosity, a term often associated with a desire to understand what you know but cannot yet fully comprehend (James, 1899; cited by Kidd and Hayden, 2015, p.2). In RL, we define curiosity as a prediction error used as a reward signal to bridge the gap between sparse extrinsic rewards by guiding the agent to explore its environment and find its next extrinsic reward. Additionally, it acts as an alternative to a well-shaped reward function by supplementing dense intrinsic rewards generated by the agent (Burda et al., 2018a). Burda et al. (2018a) performed a large-scale study on curiosity without extrinsic reward, experimenting with 54 environments of varying complexity. Naturally, it is unusual to conduct an RL experiment without extrinsic reward. However, Burda explains that curiosity provides a unique opportunity because it portrays a primary driver in development, specifically in the early stages, like babies that employ goalless exploration to learn. They found that agents can learn to obtain external rewards effectively without using an extrinsic reward during training. A deeper analysis identified that environments with stochastic dynamics generate an unprecedented issue coined the noisy-TV problem, where agents continuously change channels on a TV by seeking out environment transitions with the highest entropy. Some of their further work shows a way to eliminate the problem, a type of IM exploration bonus Random Network Distillation (RND), that they combined with a variant of PPO. Occasionally their agent managed to pass the first level of the infamous Atari game Montezuma's Revenge (Burda et al., 2018b), proving that by re-evaluating a problem and combining intrinsic and extrinsically motivated rewards, it is possible to get closer to solving complicated environments.

Another benefit of IM techniques is that they are designed to integrate with on-policy and off-policy methods due to a reform of the RL framework (Singh et al., 2010). Sutton and Barto (2018) define off-policy methods as containing higher variance and being slower to converge than on-policy methods, where on-policy are generally simpler to implement. Furthermore, they consider off-policy methods more powerful due to their applicability to numerous applications and deem them a key to learning multi-step predictive models of the world's dynamics. Their definition heavily favours off-policy methods when on-policy methods are also beneficial, especially in fields like robotics. Consider a cliff walking example (Figure 2.1) using Sarsa, an on-policy method, and Q-Learning, as an off-policy method. The agent aims to get from one side to the other as quickly as possible without falling off. Sarsa takes a longer path from the

start to the goal, represented as the safer path. And Q-Learning follows an optimal one much closer to the cliff.

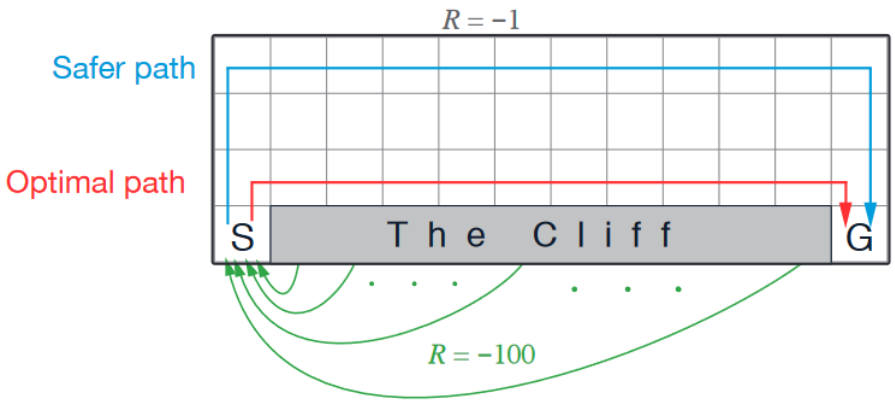

Figure 2.1: A cliff walking example using Sarsa (on-policy) and Q-Learning (off-policy). Sarsa reflects the safe path, and Q-Learning the optimal (Sutton and Barto, 2018)

As stated, Q-learning would be the best path. However, this is situational. In robotics, agents are reflected as physical robots containing expensive components, making Sarsa a more appealing option. While it creates a suboptimal policy, it reduces the likelihood of falling off the cliff due to its wider path preventing unnecessary risks of damaging expensive equipment, unlike Q-Learning. This distinction gives on-policy methods an advantage over off-policy ones when applied to real-world continuous control problems and defines a crucial difference between the methods.

A recent empirical study on on-policy methods expresses concern in successful RL experiments (Andrychowicz et al., 2020), stating that inconsistencies within algorithm implementations can impact agent behaviour. They explore approximately 50 PPO algorithms with more than 250,000 agents trained to cover different aspects of the training process, verifying their performance against the PPO paper (Schulman et al., 2017). Surprisingly, their research shows that the policy initialization scheme can significantly influence agent performance, something rarely mentioned in RL publications. Engstrom et al. (2020) highlight a broader problem: a lack of understanding of how deep RL algorithms impact agent training, encouraging the need for publications to provide more intuition on their implementations.

Fan et al. (2020) provide similar suggestions in their theoretical analysis of the off-policy method DQN. To deepen the communities understanding of the off-policy method, they examine a simplified version of DQN that still captures its key features. They focus on a statistical analysis that uses deep neural networks with rectified linear unit (ReLU) activation functions and large batch sizes. Furthermore, they reduce the DQN to a neural-fitted Q-iteration algorithm. Their results provide theoretical and statistical proof of the effectiveness of the DQN network, justifying the techniques of experience replay buffer and target networks. In practice, most off-policy methods use a replay buffer, making their work applicable to other off-policy-based methods.

The work created by Burda et al. (2018a,b), Andrychowicz et al. (2020), Engstrom et al. (2020), and Fan et al. (2020) inspired our work in this paper, aiming to expand the academic literature on the intuition behind RL implementations with the inclusion of IM techniques.

Chapter 3

Methodology

As discussed in chapter 1, this study explores the effects of IM techniques on the two types of policy-based RL methods. Throughout this section, we discuss our approach to achieving this goal. Firstly, we define our software methodology strategy and explain why it applies to this project. Next, we review the selected toolsets for conducting our research, and lastly, we outline our research methodology that highlights the practices used to compare the policy-based methods.

3.1 Project Management

To effectively manage our project, we examined three software development life cycles (SDLCs) - Waterfall, Scrum, and Kanban. SDLCs provide structure for designing, developing, and maintaining software applications.

Waterfall provides a sequential approach to SDLCs, following a five-step process: analysis, design, implementation, testing, and maintenance. Each phase must be completed before moving on to the next one and can be endlessly repeated until it is perfected (Bassil, 2012). Conversely, Scrum, an Agile methodology, focuses on iterative cycles called sprints. Each sprint is time sensitive and consists of a complete development process (such as the five steps in Waterfall), concentrating on customer satisfaction. Kanban is another Agile methodology that visualises workflows using boards and limits the number of work-in-progress tasks at each stage of development (Ahmad, Markkula and Oivo, 2013).

When considering the aims and objectives, we found that selecting Scrum presented numerous challenges due to limited opportunities for project flexibility. For example, sprints are immutable. However, the remaining methodologies, Kanban and Waterfall, both provided several benefits. Rather than using one, we combined their strengths to formulate the basis of our SDLC. Waterfall's five-step process formed the basis of our life cycle, focusing on critical tasks with predefined timeframes in a Gantt chart (Figure 3.1). Additionally, we utilised Kanban's approach for visualising the project's components, using the card list-style application Trello (Figure 3.2). We favoured this approach due to Kanban's tractability for separating the responsibilities in the Gantt chart into smaller components when required while preserving the mandated deadlines.

Figure 3.1 presents seven main stages for developing the project, covering three areas of the Waterfall life cycle (design, implementation, and testing). Firstly, we design the algorithms

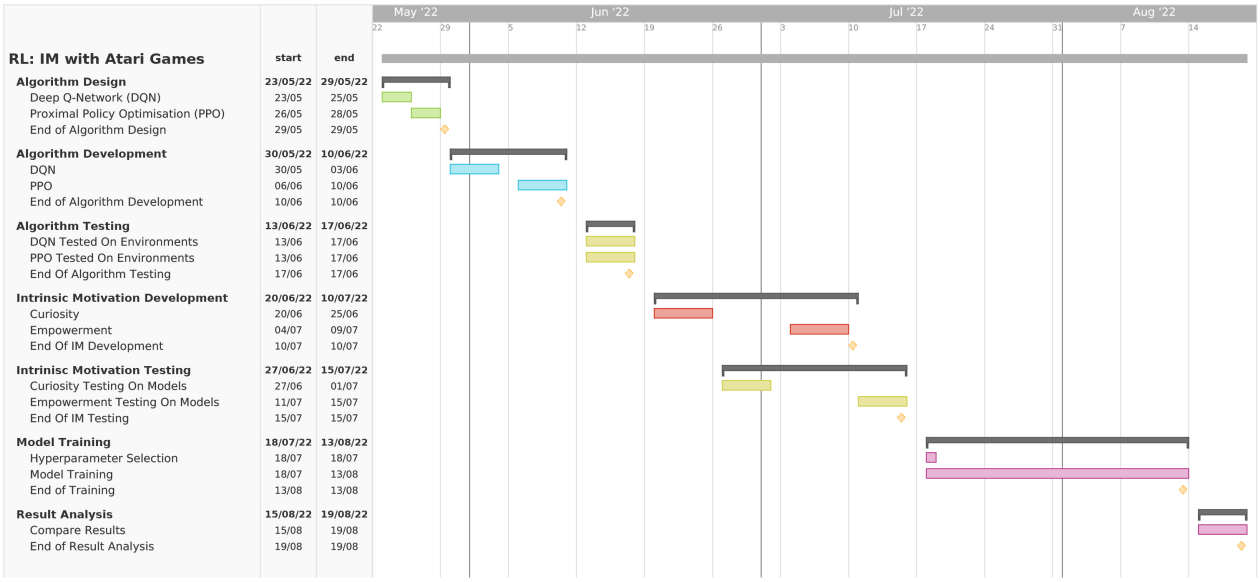

Figure 3.1: The project's Gantt chart following the Waterfall methodology.

using class diagrams that follow Object Oriented Programming (OOP) principles. Next, we build and test the algorithms in each environment, leaving additional development time between each stage for debugging. Once confident that the algorithms operate as intended, we develop, integrate and test one IM method sequentially. The approach provides flexibility for refactoring the algorithms to incorporate unconsidered components in the design stage. Lastly, we train the models in each environment and compile their results.

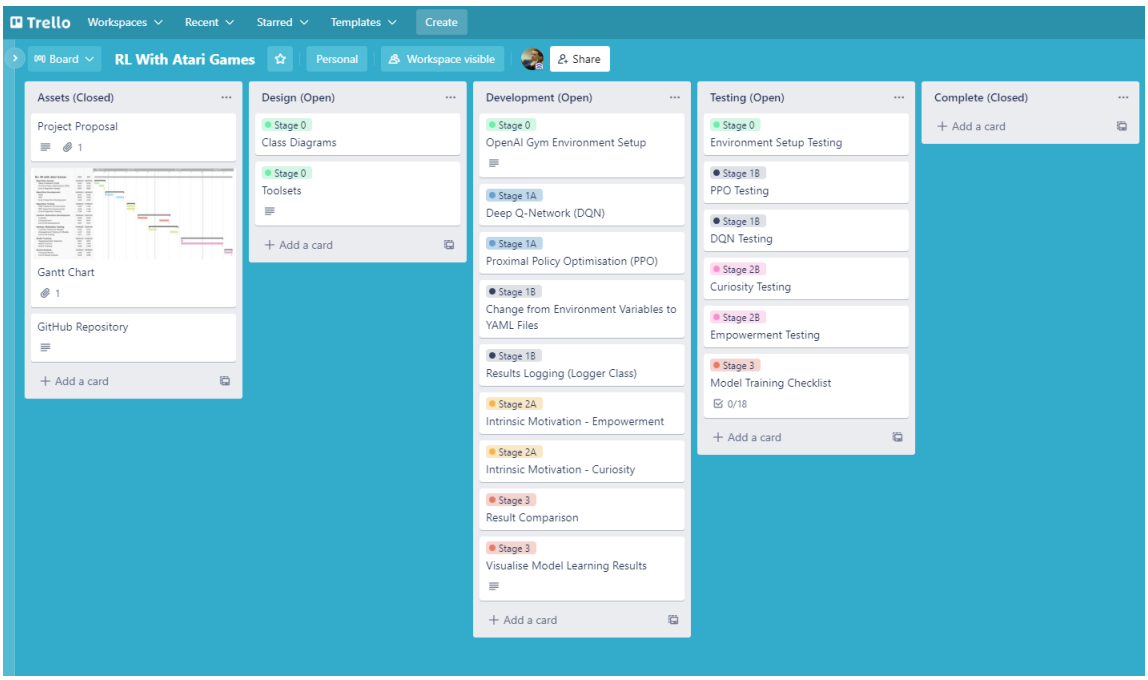

Figure 3.2: The Trello board used for the project, representing the Kanban methodology.

Furthermore, we follow a similar format for our Trello board (Figure 3.2), containing columns of cards that cover each of the previously mentioned areas of the Waterfall life cycle. We then divide the cards inside the columns into stages, ensuring consistency with the Gantt chart. We accomplish each phase in numerical (0 to 3) and subset (A to B) order, moving from the design column to the testing column until all are complete.

3.2 Software Requirements and Toolsets

Selecting the right software toolsets is crucial to improving development efficiency and reducing debugging time. For our application, we initially intended to build it as a 'one-time use' application for explicitly presenting our findings. After further consideration, we wanted to provide more benefit to the academic community and instead expanded it for wider-use cases, making it reusable and modular. Therefore, our software requirements focus on usability and modularity while maintaining our objectives for analysing the policy-based methods.

We identified Python (version 3.10) as the most beneficial programming language for our requirements compared to other languages such as Matlab and R. Python's syntax is easier to use and learn. Additionally, it contains an extensive community of developers that have created packages covering all aspects of computer science (e.g., web development, mobile development, and Machine Learning (ML)), making it highly flexible. Matlab and R, on the other hand, have a higher learning curve and more focus on ML techniques.

Another advantage to Python is that it enables virtual environment capabilities. It is common for developers to have packages and libraries already set up on a machine with the latest package versions. What if our application requires an older version of one or more packages to operate correctly? Virtual environments act as a method for separating package distributions and mitigating package version conflicts. We utilise Anaconda, a platform to manage Python environments, for this project and provide details on creating a conda environment on our GitHub repository (Partridge, 2022). We found Anaconda more valuable than other virtual environment methods, such as Python's venv built-in library, due to its seamless integration with JupyterLab, a web-based interactive development environment for viewing data and running blocks of code. We use JupyterLab's notebooks for demo code snippets and to swiftly run the application.

Moreover, Python has extensive support for Deep Learning (DL) frameworks. These frameworks provide a method for designing, building, training, and validating deep neural networks efficiently. We use PyTorch as our DL framework to ensure consistency across the algorithms and reduce development complexity. TensorFlow, another framework, was also considered but found its syntax challenging to comprehend for complex models and noticed it limits accessibility to deeper functionality, increasing complexity when resolving errors.

For more details on the Python libraries and packages used in the application, refer to Table 3.1.

Name	Description	Version
Gym	A library created by OpenAI, providing pre-built environments for RL.	0.25.2
Matplotlib	A package for visualising data and results using graphs.	3.5.1
NumPy	A numerical package for scientific computations on multi-dimensional array objects.	1.22.4
PyTorch	A tensor library for DL using graphics processing units (GPUs) or central processing units (CPUs).	1.12.1
PyYaml	A package for interacting with YAML files, a human-readable data-serialization language. We require it for organising and retrieving model hyperparameters.	6.0
SuperSuit	A library containing a collection of wrapper functions for Gym environments. We use it to vectorize PPO environments.	3.4.0

Table 3.1: Python libraries used in the project.

3.3 Research Strategy

To compare the policy-based methods, we train the algorithms on three environments (Space Invaders, Q*bert, and Montezuma's Revenge), facilitating three reward categories: extrinsic reward only (no IM), curiosity with extrinsic reward, and empowerment with extrinsic reward. Our approach allows us to effectively compare the differences between the IM techniques and the policy-based models.

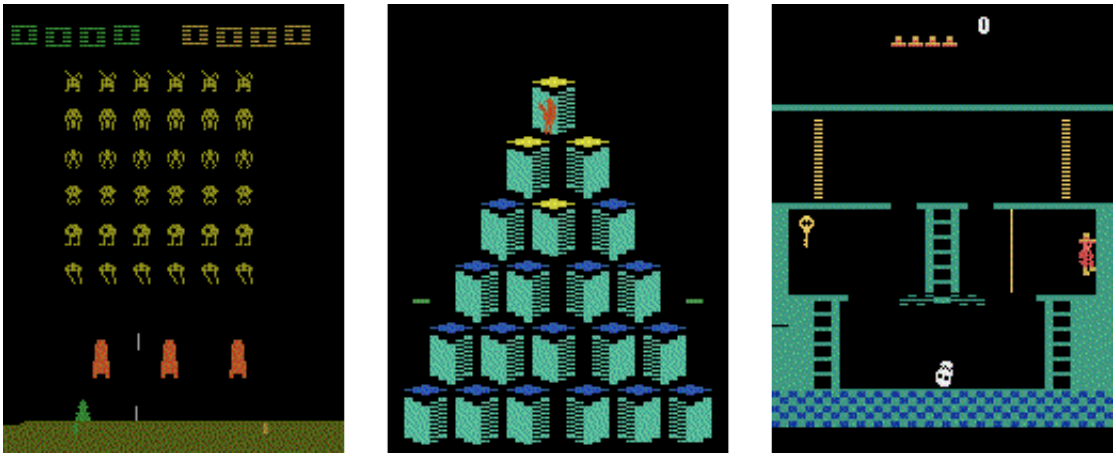

Figure 3.3: A single frame from each environment, Space Invaders, Q*bert, and Montezuma's Revenge, respectively (Farama Foundation, 2022).

The environments (Figure 3.3) provide varying degrees of complexity, allowing us to grasp the similarities and differences between the techniques and models when exposed to different circumstances. Space Invaders is the easiest to solve, containing a vast reward space with

minimal effort required to avoid episode termination. Q*bert is slightly more challenging, also possessing a high reward space but demands precise movements to avoid defeat. Montezuma's Revenge, the most challenging environment, forces agents to broadly explore their environment before obtaining a single reward. Our strategy provides us with 18 models to explore, 9 for DQN (off-policy) and 9 for PPO (on-policy), containing each reward category across all three environments.

To prevent inadvertent bias, we use identical hyperparameters across all model variants. We selected them based on previous research in the field, such as Mnih et al. (2015) and Schulman et al. (2017), with some variations to reduce training time. We outline the hyperparameters in Table 3.2.

Model/Technique	**Parameter**	**Value**
DQN and PPO	Discount factor (γ)	0.99
	Network updates	4
	Max gradient clipping	0.5
	Learning rate (α)	0.001
	Optimizer stability weight (ϵ)	0.001
DQN	Max timesteps per episode	1,000
	Target network soft update weight (τ)	0.001
	Epsilon greedy threshold	0.1
	Epsilon decay rate	0.995
	Buffer size	1,000
	Mini-batch size	32
PPO	Surrogate clip value	0.1
	Rollout size	10
	Number of environments	4
	Mini-batch size	4
	Entropy coefficient (λ)	0.01
	Value loss coefficient	0.5
Curiosity	Comparison weight (β)	0.2
	Importance weight (λ)	0.1
	Reward scaling factor (η)	1
Empowerment	Softplus scale factor (β)	1
	Reward scaling factor	00.1

Table 3.2: The hyperparameters used for the DQN, PPO, and IM methods.

Usually, we would evaluate our models' performance using metrics such as loss values. However, we take a different approach and focus on how the agents operate in their environment by comparing their actions. We believe this approach better reflects our research goals, preventing needless optimisation of model performance. Recall that our research focuses on exploring the effects IM methods have on the selected policy-based models, not solving the environments.

3.4 Testing Strategy

Software testing is an essential part of any software application and formulating the correct strategy is crucial for reducing development time and improving application usability.

Our testing strategy focuses on white-box testing, a set of techniques that require knowledge of the application source code. We use a combination of unit and structural testing to validate the components of our code during and after full component development. During development, unit testing ensured the validity of small subsets of our program. These tests were basic examples that confirmed the application worked, though, unable to provide a complete representation of how the application would perform during our research.

After component development, structural testing helped detect unforeseen errors during a standard research procedure. We found this strategy extremely effective, reducing our total development time by 10%.

Chapter 4

Design and Development

4.1 Design Requirements

The application design concentrates on modularly creating policy-based algorithms with the extension of IM techniques that can be enabled or disabled. Additionally, we focus on developer/researcher usability, covering five user requirements:

1. Hyperparameter access – users can modify hyperparameters freely without changing core application functionality.
2. Creating and training models – users can efficiently create and train models in a few lines of code given a small number of parameters.
3. Validate models – users can watch an agent perform in a selected environment and interpret its performance visually and analytically.
4. Save and load models – users can save and load a previously taught model's state.
5. Improving models – users can improve a previously saved model by training it for longer.

We utilise these requirements to formulate the foundation of the application's design, as demonstrated in the user process flowchart in Figure 4.1.

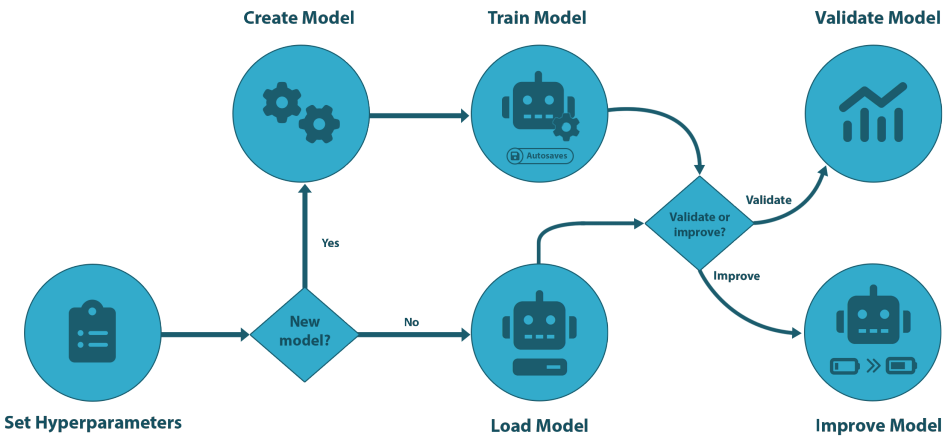

Figure 4.1: A flowchart of the applications user process.

4.2 Application Design

At a user level, we divide the application into five core components (creation, training, loading, validation, and improvement) that are accessible using a few lines of code. However, at a practical level, each component has various layers of abstraction to achieve the design requirements. Throughout this section, we explore these components in more detail.

4.2.1 Model Creation

Our first component, model creation (Figure 4.2), requires knowledge of the agent algorithm, the environment it operates in, what type of IM it uses, and access to the appropriate hyperparameters. Due to its complexity, this component relies on multiple individual classes to operate and is accessible through a single controller function. Each policy-based agent stores its functionality in separate classes. The agents inherit from a base agent class which provides utility functions usable by both types of agents while ensuring implementation consistency.

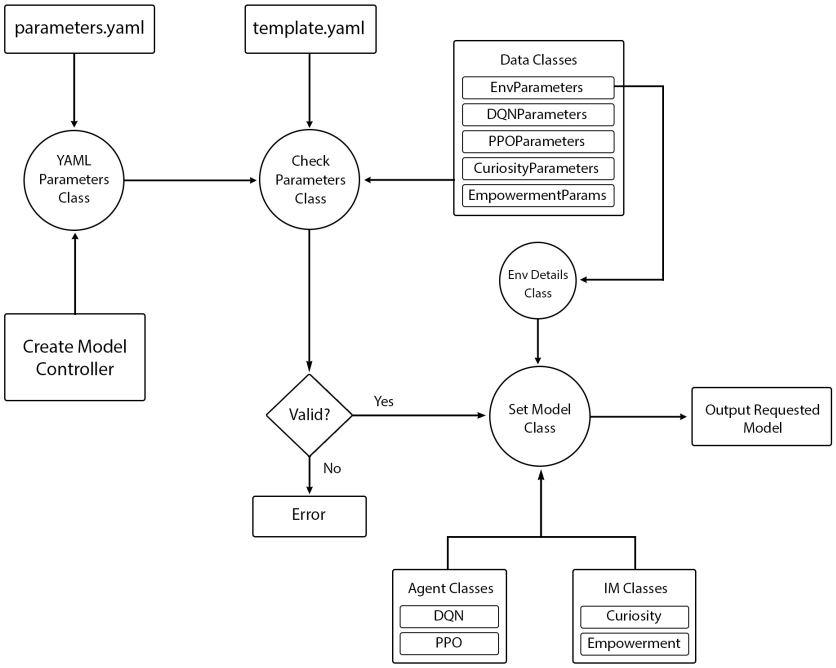

Figure 4.2: A flowchart outlining the model creation process.

When creating a model, the controller first obtains the hyperparameters from a YAML file. Traditionally, hyperparameters are declared at the top of relevant programming files and passed into application components manually. Unfortunately, this method is cumbersome and requires users to access low-level files to adjust parameters, threatening accidental functionality modifications. We mitigate this by defining hyperparameters in a bespoke YAML file, allowing flexibility for reviewing and modifying them. Comparative to other file types such as JSON and Python environment variables, YAML favours human readability, making it perfect for maintaining usability.

Next, parameters are validated against a template YAML file containing the structure of the hyperparameters. Firstly, the parameters are loaded into the application as individual data classes, defined by the template structure, and are only selected based on the users'

requirements. For example, if building a DQN algorithm with the curiosity IM technique, the YAML file parameters are only verified against DQN and IM curiosity parameters. If validation succeeds, the controller creates a model with or without the required IM method. Additionally, it creates an OpenAI gym environment dynamically, using a provided environment name and automatically allocates it to the algorithm, ready for training. When the parameters are invalid, the controller returns an error informing the user how to proceed.

4.2.2 Model Training

Model training uses the created model to train the selected agent within its respective environment by calling the model's train function. We use this function as a controller to maintain consistency between on-policy and off-policy agents. When learning the policy-based agents operate differently but have designed with some similarities for enhanced usability.

During agent training, progress updates are provided at a user-defined episode count. They include details such as the episode score, episode train loss, IM losses (if applicable), and the total real-time completed for that batch of episodes. Additionally, at a user-defined save count (e.g., every 1,000 episodes), the state of the agent is automatically saved to the device inside a saved models folder with a separate file containing additional information obtained during training. The information varies depending on the selected algorithm. For example, DQN stores actions, train losses, episode scores, and intrinsic losses. While PPO stores actions, average returns, average rewards, and various loss values.

Training can take several hours or even days to complete. Sometimes unforeseen circumstances transpire, such as computer crashes, and having to start training from the beginning again is frustrating and incredibly time-consuming. We include auto-saving to ensure that some progress is stored while models train over a long duration to mitigate this problem. We discuss each algorithm's training processes with and without IM techniques in section 4.3.

4.2.3 Loading Models

The third component, model loading, is a utility to the main application (creating and loading). Unlike model creation, which requires a manual declaration of the model, it automatically generates one from a provided filename of a previously trained model. Firstly, it verifies that the file exists within a saved model's folder before extracting the model type from the filename. Next, the filename and model type are distributed to a data loader class. The class accesses the checkpointed file, allocating associated elements such as neural network states (weights and biases), agent parameters, environment parameters, and the IM type (if applicable) to a template of the model type. The function then returns the loaded model allowing the same operations as a newly created one.

4.2.4 Validation Models

Model verification is a hypernym for analysing models visually and analytically. We designed this approach focusing on user flexibility, allowing adaptation to user requirements. For example, to visualise an agent performing in its environment, users access a video render function that accepts an agent model and displays a live video feed of the agent operating in its pre-determined environment.

Similarly, a plot results function characterises a controller for various plots to analyse agent metrics. The controller accepts the type of graph to display and the desired information to examine, automatically generating a graphic from the Matplotlib library.

4.2.5 Improving Models

Model improvement focuses on re-training agents in their pre-determined environments. It combines the functionality for loading and training models with the addition of an episode termination count. We added the component to provide a means for continuing agent learning after training is terminated (intentionally or unintentionally). It extends the loading and training functionality by calculating the difference between the total trained count from the filename and the episode termination count. Then it starts the training process at the previously saved state. For example, if we load a previous agent state trained for 20 episodes and we want to extend it to 100 episodes, we continue training from episode 20 for an additional 80.

4.3 Algorithm Implementations

In this section, we discuss the main components of our algorithm implementations and assume fundamental knowledge of RL is known.

4.3.1 Environments

We build the environments using the OpenAI Gym framework to reduce development time and guarantee stability. Additionally, it provides extensions (wrappers) for modifying functionality without accessing low-level functions. Each environment contains an observation space, a storage container for information about the environment, and an action space, denoting the available agent actions.

The observation space provides raw environment frames of 210x160 pixels with three colour channels (128-colour palette), totalling 100,800-pixel values. Using these frames is computationally demanding and memory intensive. Firstly, we apply two wrappers to pre-process the environment states, reducing the dimensionality to 84x84 pixels and transform them into grayscale images (a single colour channel), resulting in a 93% decrease in pixel values (7,056 total). Secondly, we use a third wrapper to stack four frames together to simulate the velocity of objects, increasing the final count to 28,224 pixels. Lastly, we normalize the environment frames between [0, 1] before supplying them to the CNN, as discussed in subsection 4.3.2, to increase learning and prediction performance.

The environment action spaces remain unchanged, where Space Invaders and Q*bert provide six discrete agent actions and Montezuma's Revenge with eighteen.

4.3.2 Convolutional Neural Networks (CNNs)

CNNs are a Deep Learning algorithm that accepts an image as input and provides an efficient means for capturing spatial and temporal dependencies in images using kernel filters. Moreover, they reduce input images into a simpler form while maintaining information critical to obtaining valuable predictions. They mainly consist of convolutional layers to perform feature extraction, separating distinct image features into feature maps, and fully-connected layers that accept

a flattened version of the feature maps and introduce non-linearity. The network outputs a reduced result of predictions, such as an action probability distribution or a single value.

To simplify our models, we use the same feature extraction and fully-connected layers for our DQN and PPO implementations, except for differing output layers due to varying model functionality. The architecture comprises three convolutional layers and two fully-connected (dense) layers with rectified linear unit (ReLU) activation functions. We base the convolutions on the same structure as Hessel et al. (2017) research and define the CNN layers in more detail in Table 4.1.

Layer Type	Input	Output	Kernel Size	Stride
Convolution	84x84x4	20x20x32	8x8	4
Convolution	20x20x32	9x9x64	4x4	2
Convolution	9x9x64	7x7x64	3x3	1
Dense	3,136	256	-	-
Dense	256	128	-	-

Table 4.1: The CNN architecture layer details for the DQN and PPO algorithms, without the output layers.

We select the ReLU activation function (Equation 4.1) to eliminate the vanishing gradient problem in backpropagation that others, such as sigmoid, suffer from due to slow model training. The issue occurs when derivatives successively multiply together, getting immeasurably close to zero, destabilising training, and preventing model convergence. ReLU prevents the problem by pruning negative values to 0 and retaining the positive ones (initial value x). Specifically, it allows a network to easily obtain sparsity between its network connections (Glorot, Bordes and Bengio, 2011).

$$f(x) = \max(0, x) = \begin{cases} 0, & \text{for } x < 0 \\ x, & \text{for } x \geq 0 \end{cases} \tag{4.1}$$

4.3.3 Deep Q-Network (DQN)

We create the DQN algorithm following Mnih et al. (2015) research, combining Q-Learning with an Experience Replay Buffer, Fixed Q-Targets, and a CNN. Additionally, we incorporate a modular way to integrate IM methods (Algorithm 1).

Algorithm 1: Deep Q-Learning with Intrinsic Motivation

Initialise environment *Env*.
Initialise replay memory D to capacity N.
Initialise action-value function Q with random weights θ.
Initialise target action-value function $\hat{Q}$ with weights $\theta^- = \theta$.
Initialise starting epsilon ϵ, greedy threshold ϵ_{th} and decay rate ϵ_{dr}.
Initialise intrinsic motivation method *IM*.
For *episode = 1, M* **do**
  Initialise first image $\phi_1 = x_1$ and preprocess it $S_1 = S(\phi_1)$.
  For *t=1, T* **do**
    Get action by following policy $a_t = act(Q, s, \epsilon)$ (Algorithm 2).
    Execute action a_t in *Env* and observe reward r_t and image x_{t+1}.
    Set $\phi_{t+1} = a_t, x_{t+1}$ and preprocess $S_{t+1} = S(\phi_{t+1})$.
    Store transition (S_t, a_t, r_t, S_{t+1}) in D.
    Sample random minibatch of transitions (S_j, a_j, r_j, S_{j+1}) from D.
    Set $y_j = \begin{cases} r_j, & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(S_{j+1}, a', \theta^-), & \text{otherwise} \end{cases}$
    Calculate intrinsic motivation reward and IM_{rew} and loss IM_{loss}.
    Calculate Q targets $Q_{tar} = y_j - Q(S_j, a_j; \theta) + IM_{rew}$.
    Perform gradient descent step on $(Q_{tar} + IM_{loss})^2$ with respect to θ.
  end
  Decrease $\epsilon = \epsilon_{th}$ if $\epsilon < \epsilon_{th}$, otherwise $\epsilon \cdot \epsilon_{dr}$.
  Soft update $\hat{Q}$ parameters $\theta^- = \tau\theta + (1-\tau)\,\theta^-$.
end

For fixed Q-targets, we use a local and target network. Both follow the same structure from subsection 4.3.2, where the network output layer contains 128 nodes to accept the previous layer's input and returns Q-values (a significance estimate for taking action A in state S), replicating the size of the environment action space.

Using a single network for function approximation in RL often destabilizes learning because network updates attempt to reach a moving target. We mitigate this issue by adding a second network for calculating target Q-values at every few timesteps, simulating a fixed target value and reducing learning variance. The update method is known as a 'hard' update and performs a complete copy of the local network parameters, replacing the target ones ($\theta^- = \theta$).

$$\Delta\theta = \alpha \left[R + \gamma \max_{a} \hat{Q}(S', a, \theta^-) - Q(S, A, \theta) \right] \Delta\theta \; Q(S, A, \theta) \tag{4.2}$$

Equation 4.2 defines the fixed Q-target update rule, the method for calculating a change in the network parameters $\Delta\theta$ (weights and biases). We compute it by finding the difference between the TD target $R + \gamma \max_a \hat{Q}(S', a, \theta^-)$ and the current predicted Q-value $Q(S, A, \theta)$. Then, multiply the result by the current predicted Q-value's gradient $\Delta\theta \; Q(S, A, \theta)$ and a learning

rate α. θ^- represents the target network parameters, and $\langle S, A, R, S' \rangle$ are components from a tuple of experience sampled from the replay buffer representing the current state, the action the agent has taken, the environment reward, and the next state.

We replace the 'hard' update with t-soft updates (Equation 4.3, Kobayashi and Ilboudo, 2021) that incrementally replace the parameters a small amount every timestep, decreasing the likelihood of generating wrong reference signals using an exponential moving average τ. We define τ as a hyperparameter between $[0, 1]$ and resume 'hard' updates when $\tau = 1$.

$$\theta^- = \tau\theta + (1 - \tau)\,\theta^- \tag{4.3}$$

Another benefit to Mnih et al. (2015) approach is Experience Replay buffers. In the naïve Q-Learning algorithm, tuples of experiences $E_t = \langle S, A, R, S' \rangle$ are generated sequentially, learned from, and then discarded. Sometimes, agents can learn more from previously seen experiences, especially those that rarely occur. Moreover, sequential order experiences are highly correlated. Experience Replay stores visited experiences, enabling the sampling of random batches of experiences, allowing old ones to be revisited and eliminating the sequential correlation problem. We create the replay buffer as a separate container class, housing a memory of agent experiences with callable methods that permit adding and sampling.

Algorithm 2: ϵ-Greedy Policy

Input : S current state, T greedy threshold, Q action-value function.
Output : Selected action.
Function *act(Q, s, ϵ)* **is**

$$A = \begin{cases} \text{a random action}, & T < \epsilon \\ \arg\max Q(S), & \text{otherwise} \end{cases}$$

return : A.
end

Lastly, we use an ϵ-greedy policy (Algorithm 2) with a decaying ϵ to incentivise agent exploration during the early stages of learning. Starting at $\epsilon = 1$, we reduce it by 0.995 every episode until we reach the greedy threshold of 0.1. Subsequently, agents only take actions that exploit the available rewards. If ϵ is larger than the threshold, we select a random one instead.

4.3.4 Proximal Policy Optimization (PPO)

We implement the PPO-Clip algorithm following the original paper by Schulman et al. (2017), with the addition of IM techniques and a clipped value loss function defined by Engstrom et al. (2020). We outline the algorithm details in Algorithm 3.

Algorithm 3: PPO-Clip with Intrinsic Motivation

Initialise a set of M environments *Envs*.
Initialise policy π and action-value function Q with parameters θ.
Initialise intrinsic motivation method *IM*.
For *episode = 1, e* **do**
  Collect N set of rollout trajectories $\mathcal{D}_k = \tau_i$ by running policy $\pi_k = \pi(\theta_k)$ in *Envs*.
  Compute rollout log probabilities $r_e(\theta)$ and entropy's S_e using categorical distribution.
  Compute episode rewards-to-go $\hat{R}_e$.
  Compute episode advantage estimates $\hat{A}_e$ based on Q_{ϕ_k}.
  For *minibatch = 1, t* **do**
    Set rewards-to-go minibatch $\hat{R}_t$.
    Set advantage estimate minibatch $\hat{A}_t$.
    Calculate intrinsic motivation reward IM_{rew} and loss IM_{loss}.
    Compute minibatch returns $\hat{R}_t + IM_{rew}$.
    Compute policy loss $L_t^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right]$.
    Compute value loss $L_t^{VL}(\theta) = \min \left[(V_\theta(S_t) - V_t^{targ})^2, VL_{clip} - V_t^{targ})^2 \right]$.
    Compute entropy loss $SL = \frac{\sum S_e}{T}$.
    Compute total loss $L_t(\theta) = L_t^{CLIP}(\theta) - c_1 SL_t + L_t^{VL}(\theta) c_2 + IM_{loss}$.
    Perform gradient descent step on π and Q with respect to θ.
  end
end

In PPO, agents learn directly from a sequence of experiences rather than a replay buffer like DQNs. It is simpler to implement and has a better sample complexity (empirically). We use one Neural Network as an Actor-Critic to compute an action probability distribution and a single Q-value.

The architecture follows the same layers defined in subsection 4.3.2 but with two fully-connected branches that provide different output streams. The first branch computes the action probability distribution (actor), containing an output layer with 128 input nodes and output nodes of the same size as the environment action space. The output nodes are passed through a Softmax activation function to produce the distribution. The second branch (critic) follows the same as the first branch but returns a single Q-value instead of a distribution.

To compute the loss of our network, we use a combination of a clipped surrogate objective function $L^{CLIP}(\theta)$ (Equation 4.5), clipped value loss function $L^{VL}(\theta)$ (Equation 4.6) and entropy loss SL to encourage agent exploration. We denote the complete loss formula in Equation 4.4.

$$L_t(\theta) = L_t^{CLIP}(\theta) - c_1 SL_t + L_t^{VL}(\theta) c_2 \tag{4.4}$$

$$L_t^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right] \tag{4.5}$$

$$\begin{aligned} L_t^{VL}(\theta) &= \min\left[(V_\theta(S_t) - V_t^{targ})^2, VL_{clip} - V_t^{targ})^2\right] \\ VL_{clip} &= \text{clip}\left(V_\theta(S_t), V_\theta(S_{t-1}) - \epsilon, V_\theta(S_{t-1}) + \epsilon\right) \end{aligned} \tag{4.6}$$

Where c_1 is an entropy coefficient, c_2 is a value loss coefficient for regularisation in Equation 4.4. In equation Equation 4.5, ϵ is a hyperparameter. $r_t(\theta)$ is a probability ratio $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$, with π_θ as a stochastic policy and $\hat{A}_t$ an advantage function estimate at timestep t. The expectation $\hat{\mathbb{E}}_t[...]$ indicates the empirical average over a finite batch of samples, and the second term in the surrogate function $\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t$ limits the probability ratio between the interval $[1-\epsilon, 1+\epsilon]$. Lastly, the surrogate minimises the clipped and unclipped objective, ignoring the change in probability ratio when improving the objective function, and including it when deteriorating. We define the clipped value loss function in Equation 4.6 as the smallest value between the squared-error loss $(V_\theta(S_t) - V_t^{targ})^2$ in the original paper or a clipped variant.

Another core part of our implementation is our method for calculating the rewards-to-go (returns) and advantages, a statistic representing how effective a single action is compared to an average of all actions in a state. At the start of each episode, we generate a set of rollouts (agent experiences) up to n timesteps and use them to compute the advantages and returns. We formulate the episode returns G and advantages $\hat{A}$ in Equation 4.7 and 4.8, respectively.

$$G = R + \gamma R_{mask} G_{i+1} \tag{4.7}$$

$$\hat{A}_t = G - Q_c \tag{4.8}$$

Where R is a set of rollout rewards, γ is a discount factor, and R_{mask} is a statistical representation of the next non-terminal state. Q_c signifies the rollout state values, and G_{i+1} is the next return in the rollouts, with i as the rollout index.

Since we have a discrete action space, we use a categorical distribution to compute the agent's action, log probabilities and entropy values. Using this approach simplifies our algorithm and works nicely with the PyTorch package. Additionally, we clip the environment reward values between $[-1, 1]$ during rollout generation to limit the score variance. All positive rewards become 1, negative -1 and 0 rewards remain unchanged. Reward clipping limits the scale of the error derivatives, improving agent performance.

4.3.5 Intrinsic Motivation (IM)

To simplify the application design and maintain modularity, we separate the IM methods from the main application and treat them as plugins accessible via an IM controller class. The controller manages the IM techniques, storing its models and a module of functionality easily importable into the algorithms, as shown in algorithms 1 and 3. The IM methods are conditioned based on the user input from the create model controller discussed in subsection 4.2.1, allowing them to be disabled if required.

The approach is only possible due to the elegant separation between the two types of reward (extrinsic and intrinsic), demonstrated in Equation 4.9, where the total reward R_t is the summation of extrinsic R_t^E and intrinsic reward R_t^I.

$$R_t = R_t^E + R_t^I \tag{4.9}$$

The selected IM techniques enable different strategies to improve agent exploration. Curiosity assists agents in exploring their environment in the quest for new knowledge. Additionally, it acts as a mechanism to learn skills that benefit them in the future (Pathak et al., 2017). Empowerment, however, represents the perceived amount of influence (or control) the agent has over the world. More specifically, the agent's potential to change the environment it is in (Klyubin, Polani and Nehaniv, 2005). We discuss the Curiosity and Empowerment implementations in subsections 4.3.6 and 4.3.7.

4.3.6 Curiosity

The Curiosity implementation is based on research by Zai and Brown (2020) and Pathak et al. (2017), focusing on an intrinsic curiosity module (ICM) that comprises of three independent neural networks: a forward-prediction model, inverse model, and an encoder.

Computationally, Curiosity acts as a form of desire to reduce the uncertainty in the agent's environment using a prediction error. The forward-prediction model (Equation 4.10) predicts the future state of the environment, given an encoder state from the encoder $\phi(S_t)$ and an action A_t, producing a prediction of an encoded next state $\hat{\phi}(S_{t+1})$.

$$\hat{\phi}(S_{t+1}) = f(\phi(S_t), A_t; \theta_F) \tag{4.10}$$

The inverse model (Equation 4.11) constrains the prediction model, mitigating vulnerability to trivial randomness or unpredictability in the environment, achieved by preventing states that have no effect (or are uncontrollable) on the agent from impacting the prediction error (Zai and Brown, 2020). Given the current state S_t and next state S_{t+1}, it directly interacts with the inverse model to encode both states before predicting the action $\hat{A}_t$ that was taken to achieve the transition from them.

$$\hat{A}_t = g(S_t, S_{t+1}; \theta_I) \tag{4.11}$$

The encoder is the simplest of the three models, serving as a mapping that reduces a state's dimensionality. We create the encoder using identical convolutional layers shown in the DQN and PPO architectures (subsection 4.3.2). It accepts a single state S_t and returns an encoded version $\phi(S_t) = S_t$.

We simplify the ICM by reducing it to a single network with two branches, like the PPO algorithm. One relates to the inverse model, outputting a probability distribution of actions using a Softmax function, and the second for the forward-prediction model. The ICM is stored inside the IM controller class (discussed in subsection 4.3.5) as a model variable and supplied to a corresponding curiosity module that supervises loss and curiosity return computations.

$$L_{cur} = \min_{\theta_Q, \theta_I, \theta_F} [\lambda Q_{loss} + (1 - \beta) F_{loss} + \beta I_{loss}] \tag{4.12}$$

$$R_t^I = \frac{1}{\eta} F_{loss} \tag{4.13}$$

In Equation 4.12, we minimise the curiosity loss L_{cur}, where λ is a hyperparameter of the importance weight between the Q-loss and learning rate, Q_{loss} and θ_Q correspond to the model loss and its parameters, β a coefficient scaling parameter weighting the inverse model against the forward model losses. F_{loss}, θ_F and I_{loss}, θ_I signify the forward and inverse model losses and parameters. Moreover, Equation 4.13 highlights the curiosity return at each timestep, with η a scaling factor for the reward, where $\eta > 0$.

4.3.7 Empowerment

We implement Empowerment following Kumar (2018) and Klyubin, Polani and Nehaniv (2005) research with minor modifications to apply a similar strategy to Curiosity. It comprises five networks: an encoder, a source model, a forward dynamics model, and two target networks for the source and forward models, respectively.

Empowerment return R_t^I is obtained by calculating the maximum mutual information $\mathcal{I}(S_{t+1}, A_t|S_t)$ (difference between the current and next state), over the policy π, that an agent can transfer to its environment when changing the next state S_{t+1} with its actions A_t, multiplied by η, a hyperparameter for scaling the reward. We utilise Mutual Information Neural Estimation (MINE, Belghazi et al., 2018) to learn the empowerment of a state using the forward dynamics model $p(S_{t+1}, A_t|S_t)$ to get samples from a marginal distribution $p(S_{t+1}, S_t)$ and the policy $\pi(A_t|S_t)$. We define the formula for the return in Equation 4.14 and MINE in 4.15.

$$R_t^I = \eta \cdot \max_{\pi} \mathcal{I}(S_{t+1}, A_t|S_t) \tag{4.14}$$

$$\mathcal{I}(S_{t+1}, A_t|S_t) = \mathrm{KL}\Big(p(S_{t+1}, A_t|S_t) \;||\; p(S_{t+1}|S_t)\Big) \tag{4.15}$$

Like the encoder in Curiosity, we use it to serve as a mapping to reduce a state's dimensionality. However, this encoder takes an extra step and converts the output feature maps into a state with one dimension. For example, given a state with the shape $(1, 4, 84, 84)$, the encoder reduces it to $(1, 84)$.

The forward dynamics model accepts an encoded state and the action the agent takes in that state to generate an encoded next state prediction $\hat{\phi}(S_t)$. The model follows the same procedure as the Curiosity forward-prediction model (Equation 4.10) but has a different architecture to accommodate the state reductions.

The source model accepts a matrix of a combination of an encoded state $\phi(S_t)$ and takeable actions that are one-hot encoded $\phi(A)$, returning a predicted action $\hat{A}_t$ (Equation 4.16). We treat the actions as categorical data and one-hot encode them to maintain the usefulness of their information when applied to the encoded state.

$$\hat{A}_t = g(\phi(S_t), \phi(A); \theta_S) \tag{4.16}$$

Like Curiosity, we store the models inside the IM controller class as a model variable and utilise an empowerment module to generate the loss and empowerment returns. We use the target networks to stabilise empowerment model training with hard updates ($\theta^- = \theta$).

The source and forward dynamics model's losses are updated individually and are not added to the algorithm's loss. We define the source model loss L_S as the negative difference between the action prediction for the current state $\hat{A}_t$ and the next state $\hat{A}_{t+1}$, where we pass both predictions through a Softplus ReLU activation function before calculating the difference between them (Equation 4.17). The loss equates to Jenson-Shannon's divergence (Menéndez et al., 1997), improving algorithm stability and reducing gradient bias.

$$L_S = -\Big(- \text{softplus}(\hat{A}_t) - \text{softplus}(\hat{A}_{t+1}) \Big) \tag{4.17}$$

We compute the forward dynamics model loss L_F as the squared difference between the encoded next state prediction $\hat{\phi}(S_{t+1})$, and the encoded next state $\phi(S_{t+1})$, expressed in Equation 4.18.

$$L_F = \Big(\phi(S_{t+1}) - \hat{\phi}(S_{t+1}) \Big)^2 \tag{4.18}$$

Chapter 5

Experiment Findings

We trained 18 models, six without IM, six with Curiosity, and six with Empowerment, across the three environments, for 100,000 episodes using the hyperparameters in Table 3.2. With this small amount of training, agents are still in the early learning stages and unable to operate in each environment effectively.

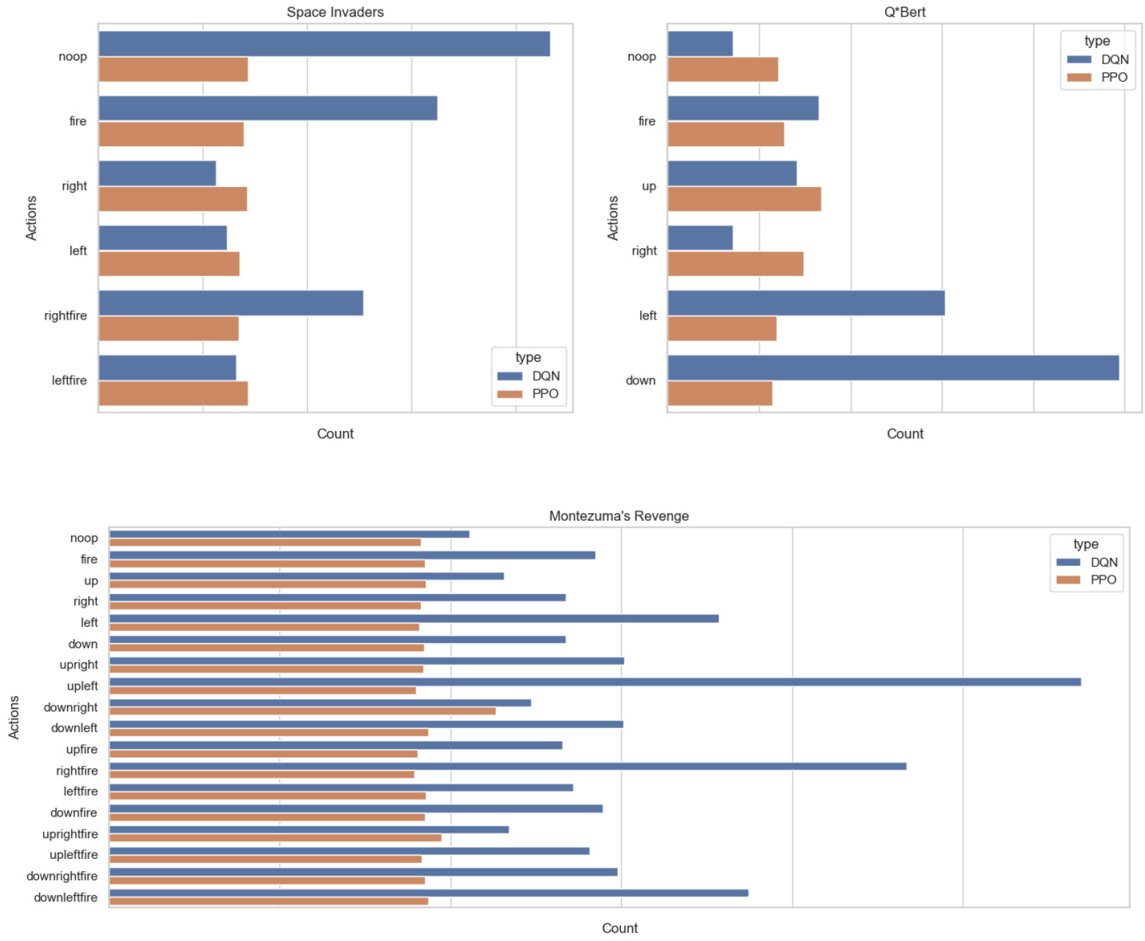

Figure 5.1: DQN and PPO agent actions taken in the three environments without IM.

In our first set of findings (Figure 5.1), DQN (off-policy) and PPO (on-policy) operate without an IM method. PPO has minimal variance in the actions taken except for the down-right action in Montezuma's Revenge. DQN, however, has a sizeable difference between some of the actions. It favours 'noop' (no operation) and the fire command in Space Invaders, moving down and left in Q*bert and up-left in Montezuma's Revenge. These actions make rational sense, as the agent is likely to hide behind the barricades in Space Invaders to avoid dying while sneakily firing at the approaching enemies. Similarly, moving down the triangle in Q*bert to maximise its reward and jumping over to the first platform in Montezuma's Revenge. Interestingly, PPO, the safer policy-based method, has struggled to decide on the best actions this early in the training stage.

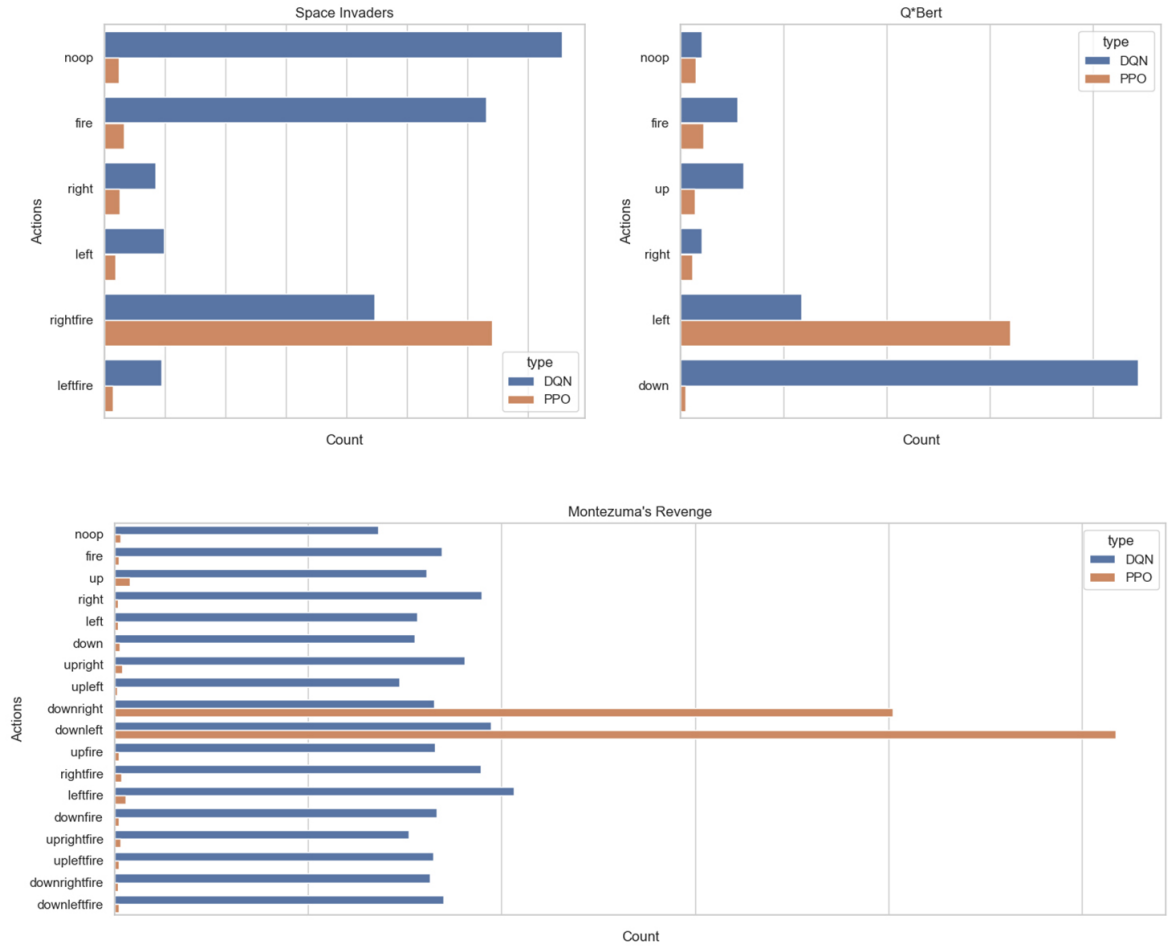

Figure 5.2: DQN and PPO agent actions taken in the three environments with an extrinsic reward and Curiosity intrinsic reward.

In our second set of findings (Figure 5.2), with the inclusion of a Curiosity intrinsic reward, the PPO agent acts substantially different, while DQN operates similarly without IM. Both algorithms prefer specific actions in the Space Invaders and Q*bert environments. Though, with Montezuma's Revenge, only PPO chooses distinct actions. We find PPO's action choices extremely unusual. Primarily selecting right-fire (moving right while firing) in Space Invaders, left in Q*bert and down-left or down-right in Montezuma's Revenge. None of these actions proves beneficial. Moving left in Q*bert results in jumping to their doom, and its action

choices in Montezuma's Revenge result in the same fate, only in the opposite direction. DQN, in contrast, acts hardly different than previously, still prioritising the 'noop' and fire actions in Space Invaders and down in Q*bert, but explores all actions equally in Montezuma's Revenge.

We theorise that the PPO agent attempts to reach the elevator block on the left side of the pyramid, supplying a high reward in Q*bert and endeavours to ignore parts of the first room in Montezuma's Revenge by dropping onto the platform below it to reach the key faster.

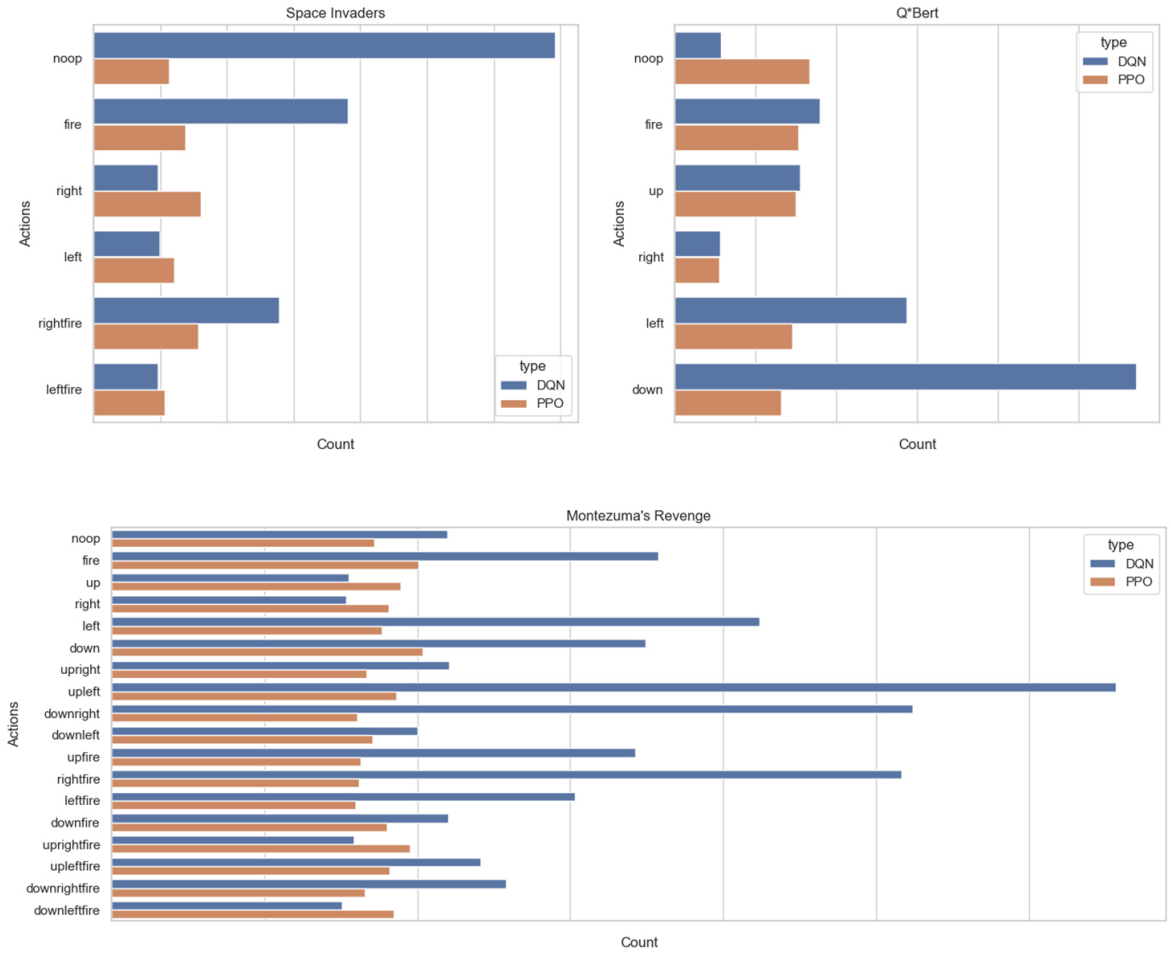

Figure 5.3: DQN and PPO agent actions taken in the three environments with an extrinsic reward and Empowerment intrinsic reward.

Lastly, the agents using the Empowerment intrinsic reward (Figure 5.3) appear associated with the first set of findings (no IM). Again, DQN frequently selects the 'noop' and fire actions in Space Invaders, down in Q*bert, and up-left in Montezuma's Revenge. Likewise, PPO varies its actions across all three environments without a distinguishable favourite.

Chapter 6

Conclusions and Future Work

Throughout this study, we explored the differences between two types of policy-based methods, DQN (off-policy) and PPO (on-policy). Highlighting their core differences and further investigating how IM affects their behaviour analytically and physically. We identify IM methods as an extension of standard RL practices, equating to a summation of an extrinsic (environment reward) and intrinsic reward (IM reward). Moreover, we analysed two popular IM methods, Curiosity and Empowerment and evaluated their performance on three Atari 2600 games, Space Invaders, Q*bert and Montezuma's Revenge, concentrating on the early stages of learning. We discovered that IM techniques are powerful tools but can provide unusual agent behaviour, such as PPO with Curiosity favouring actions that result in their death in Q*bert and Montezuma's Revenge. We expected PPO to select safer actions than DQN across all three areas (no IM, extrinsic reward with Curiosity, extrinsic reward with Empowerment) and are surprised by its results. Also, we anticipated Empowerment would have a sizeable impact on the actions they select but ultimately resulted in associated actions taken as agents without IM.

Unfortunately, limited timeframes have prevented us from providing a more extensive examination of the effects IM techniques have on a wide range of policy-based models. Therefore, we propose that further research is completed to truly understand how IM techniques adapt the behaviour of RL agents. Firstly, training the models for longer and varying hyperparameters could provide more practical insights. Additionally, exploring other IM methods, such as Surprise-based motivation and reviewing the IM techniques on other policy-based models such as Rainbow DQN, REINFORCE, and DDPG, would prove advantageous.

Word count: 8,420

References

Achiam, J. and Sastry, S., 2017. Surprise-based intrinsic motivation for deep reinforcement learning. *arXiv:1703.01732*.

Ahmad, M.O., Markkula, J. and Oivo, M., 2013. Kanban in software development: A systematic literature review. *2013 39th euromicro conference on software engineering and advanced applications*. pp.9–16.

Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J. and Mané, D., 2016. Concrete Problems in AI Safety. *arxiv:1606.06565*.

Andrychowicz, M., Raichuk, A., Stańczyk, P., Orsini, M., Girgin, S., Marinier, R., Hussenot, L., Geist, M., Pietquin, O., Michalski, M., Gelly, S. and Bachem, O., 2020. What Matters In On-Policy Reinforcement Learning? A Large-Scale Empirical Study. *arxiv:2006.05990*.

Badia, A.P., Piot, B., Kapturowski, S., Sprechmann, P., Vitvitskyi, A., Guo, D. and Blundell, C., 2020. Agent57: Outperforming the atari human benchmark. *arXiv:2003.13350*.

Bassil, Y., 2012. A simulation model for the waterfall software development life cycle. *arXiv:1205.6904*.

Belghazi, M.I., Baratin, A., Rajeswar, S., Ozair, S., Bengio, Y., Courville, A. and Hjelm, R.D., 2018. MINE: Mutual information neural estimation. *arXiv:1801.04062*.

Bellemare, M.G., Naddaf, Y., Veness, J. and Bowling, M., 2013. The arcade learning environment: An evaluation platform for general agents. *Journal of artificial intelligence research*, 47, pp.253–279.

Burda, Y., Edwards, H., Pathak, D., Storkey, A., Darrell, T. and Efros, A.A., 2018a. Large-Scale Study of Curiosity-Driven Learning. *arxiv:1808.04355*. ArXiv: 1808.04355.

Burda, Y., Edwards, H., Storkey, A. and Klimov, O., 2018b. Exploration by Random Network Distillation. *arxiv:1810.12894*.

Engstrom, L., Ilyas, A., Santurkar, S., Tsipras, D., Janoos, F., Rudolph, L. and Madry, A., 2020. Implementation Matters in Deep Policy Gradients: A Case Study on PPO and TRPO. *arxiv:2005.12729*.

Fakoor, R., Chaudhari, P. and Smola, A.J., 2019. P3o: Policy-on policy-off policy optimization. *arXiv:1905.01756*.

Fan, J., Wang, Z., Xie, Y. and Yang, Z., 2020. A Theoretical Analysis of Deep Q-Learning. *arxiv:1901.00137*.

Farama Foundation, 2022. *Gym documentation* [Online]. Available from: `https://www.gymlibrary.dev/` [Accessed 02 September 2022].

Fujimoto, S., Meger, D. and Precup, D., 2019. Off-policy deep reinforcement learning without exploration. *Proceedings of the 36th international conference on machine learning*. PMLR, pp.2052–2062. ISSN: 2640-3498.

Glorot, X., Bordes, A. and Bengio, Y., 2011. Deep sparse rectifier neural networks. *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, pp.315–323. ISSN: 1938-7228.

Gu, S., Lillicrap, T., Ghahramani, Z., Turner, R.E. and Levine, S., 2017. Q-prop: Sample-efficient policy gradient with an off-policy critic. *arXiv:1611.02247*.

Hasselt, H. van, Guez, A. and Silver, D., 2015. Deep Reinforcement Learning with Double Q-learning. *arxiv:1509.06461*.

Hessel, M., Modayil, J., Hasselt, H. van, Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. and Silver, D., 2017. Rainbow: Combining Improvements in Deep Reinforcement Learning. *Thirty-second aaai conference on artificial intelligence*, 32(1).

Kidd, C. and Hayden, B.Y., 2015. The psychology and neuroscience of curiosity. *Neuron*, 88(3), pp.449–460.

Klyubin, A., Polani, D. and Nehaniv, C., 2005. All else being equal be empowered. *Advances in artificial life*. Berlin, Heidelberg: Springer Berlin Heidelberg, vol. 3630, pp.744–753.

Kobayashi, T. and Ilboudo, W.E.L., 2021. t-soft update of target network for deep reinforcement learning. *Neural networks*, 136, pp.63–71.

Kumar, N.M., 2018. Empowerment-driven exploration using mutual information estimation. *arXiv:1810.05533*.

Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D. and Wierstra, D., 2019. Continuous control with deep reinforcement learning. *arxiv:1509.02971*.

Menéndez, M., Pardo, J., Pardo, L. and Pardo, M., 1997. The jensen-shannon divergence. *Journal of the franklin institute*, 334(2), pp.307–318.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2013. Playing Atari with Deep Reinforcement Learning. *arxiv:1312.5602*.

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

Munos, R., Stepleton, T., Harutyunyan, A. and Bellemare, M., 2016. Safe and efficient off-policy reinforcement learning. *Advances in neural information processing systems*. Curran Associates, Inc., vol. 29.

Partridge, R., 2022. *Reinforcement learning with atari games* [Online]. GitHub. Available from: `https://github.com/Achronus/rl_atari_games` [Accessed 29 August 2022].

Pathak, D., Agrawal, P., Efros, A.A. and Darrell, T., 2017. Curiosity-driven exploration by self-supervised prediction. *arXiv:1705.05363*.

Pollack, J. and Blair, A., 1996. Why did TD-Gammon Work? *Advances in Neural Information Processing Systems*. MIT Press, vol. 9.

Ryan, R.M. and Deci, E.L., 2000. Intrinsic and extrinsic motivations: Classic definitions and new directions. *Contemporary educational psychology*, 25(1), pp.54–67.

Schaul, T., Quan, J., Antonoglou, I. and Silver, D., 2016. Prioritized Experience Replay. *arxiv:1511.05952*.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O., 2017. Proximal policy optimization algorithms. *arXiv:1707.06347*.

Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Driessche, G. van den, Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T. and Hassabis, D., 2016. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587), pp.484–489.

Silver, D., Singh, S., Precup, D. and Sutton, R.S., 2021. Reward is enough. *Artificial intelligence*, 299, p.103535.

Singh, S., Lewis, R.L. and Barto, A.G., 2009. Where do rewards come from? *31st annual conference*. CogSci 2009, pp.2601–2606.

Singh, S., Lewis, R.L., Barto, A.G. and Sorg, J., 2010. Intrinsically Motivated Reinforcement Learning: An Evolutionary Perspective. *Ieee transactions on autonomous mental development*, 2(2), pp.70–82.

Stout, A. and Barto, A.G., 2010. Competence progress intrinsic motivation. *2010 IEEE 9th international conference on development and learning*. IEEE, pp.257–262.

Sutton, R.S. and Barto, A.G., 1998. *Reinforcement learning: An introduction*. MIT Press.

Sutton, R.S. and Barto, A.G., 2018. *Reinforcement learning: An introduction*. 2nd ed. MIT Press.

Tesauro, G., 1995. Temporal difference learning and TD-Gammon. *Communications of the acm*, 38(3), pp.58–68.

Thomas, P. and Brunskill, E., 2016. Data-efficient off-policy policy evaluation for reinforcement learning. *Proceedings of the 33rd international conference on machine learning*. PMLR, pp.2139–2148. ISSN: 1938-7228.

Turing, A.M., 1950. I.—COMPUTING MACHINERY AND INTELLIGENCE. *Mind*, LIX(236), pp.433–460.

Wang, Z., Schaul, T., Hessel, M., Hasselt, H. van, Lanctot, M. and Freitas, N. de, 2016. Dueling Network Architectures for Deep Reinforcement Learning. *arxiv:1511.06581*.

Zai, A. and Brown, B., 2020. *Deep reinforcement learning in action*. Manning Publications.