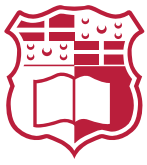
AI for Partially Observable Stochastic Games

Cristina Cutajar

Supervisor: Dr Josef Bajada

November 2025

Submitted in partial fulfilment of the requirements for the degree of Master of Science (by Research).



L-Università ta' Malta
Faculty of Information &
Communication Technology

Abstract

Reinforcement Learning (RL) has shown significant success in a variety of game environments. Despite this, some game characteristics continue to challenge RL. Jaipur, a competitive two-player turn-based board game, contains a number of such challenging features. It is characterised by partial observability, stochasticity and a large discrete action space of 25,499 possible actions. Moreover, it also contains elements of randomness, immediate and long-term rewards with different consequences, and multiple different strategies that can be adopted.

This study aims to analyse the performance of several state-of-the-art techniques and algorithms that can be implemented, to not only mitigate the complexities of applying RL on Jaipur, but also improve the performance and training efficiency. We propose the implementation of the action masking, action embedding, hierarchical RL with centralised critic and policy cloning techniques. Moreover, hyperparameter tuning was applied using the PBT algorithm, and to evaluate the effects of partial observability on the RL process, different levels of observability were provided in separate experiments.

For each implementation, the PPO, A2C, DQN and DDQN algorithms were trained with two separate policies, to reflect Jaipur's two competitive players. The scores obtained during training, as well as when the policies of each model were played against each other on 1000 unique games, were evaluated quantitatively against scores obtained by human players and by the other models.

The results demonstrate that all algorithms and techniques achieved good scores, comparable to those achieved by humans. Action masking delivered the best overall performance, with high scores achieved at high computational efficiency across most algorithms. Action embedding obtained better scores for PPO but required the longest training times, whilst the training times for DQN and DDQN were the shortest. Meanwhile hierarchical RL with centralised critic provided greater training stability, however, the scores achieved were lower across most models and the training times were significantly prolonged. Both the hyper-parameter tuning and policy cloning technique proved to be beneficial, as the performance of the algorithms increased in less training steps. Meanwhile, the varying levels of observability had minimal impact on the policies' performance, suggesting that the algorithms managed to discover strong strategies that achieved high scores even with partial observability.

Furthermore, an action selection analysis of the policies' decisions during simulated games was carried out, from which it was concluded that all the policies adopted intelligent and interesting strategies, similar to those of human players.

Acknowledgements

I would like to express my sincerest gratitude to my supervisor Dr Josef Bajada for his unwavering guidance, patience, support and invaluable advice throughout my studies. I wish to extend my heartfelt appreciation to my family and closest friends for their encouragement and support. In particular, I would like to thank my mother, Eleonora, my boyfriend, Craig, and my brother, Mattia, whose enduring support, patience and understanding have continually inspired me to pursue my dreams and accomplish my goals. I would also like to show my deepest appreciation towards my dog, Pops, for his unconditional loyalty, love and emotional support.

Related Publication

The findings of part of this work were published in the AIxIA 2023 - Advances in Artificial Intelligence conference under the title “Mastering the Card Game of Jaipur Through Zero-Knowledge Self-Play Reinforcement Learning and Action Masks” [1]. This publication presents a subset of the methods and results that were later extended and refined in this work.

Contents

List of Figures

List of Tables

List of Abbreviations

A2C Advantage Actor Critic.

A3C Asynchronous Advantage Actor Critic.

AEC Agent Environment Cycle.

AEN Action Elimination Network.

AI Artificial Intelligence.

API Application Programming Interface.

CMRL Cooperative Modular Reinforcement Learning.

CNN Convolutional Neural Network.

D-PPO Diffusion Policy Policy Optimization.

D-SAC Distributional Soft Actor Critic.

DDPQ Deep Deterministic Policy Gradient.

DDQN Double Deep Q-Learning.

DL Deep Learning.

DNN Deep Neural Network.

DQN Deep Q-Learning.

DRL Deep Reinforcement Learning.

EVG Electric Vehicle Groups.

FFAI Fantasy Football AI.

HRL Hierarchical Reinforcement Learning.

IID Independent and Identically Distributed.

LSTM Long Short-Term Memory.

MARL Multi-Agent Reinforcement Learning.

MCTS Monte-Carlo Tree Search.

MDP Markov Decision Process.

ML Machine Learning.

MLP Multilayer Perceptron.

MOBA Multi-Player Online Battle Arena.

NN Neural Network.

PBT Population Based Training.

POMDP Partially Observable Markov Decision Process.

POSG Partially Observable Stochastic Game.

PPO Proximal Policy Optimization.

ReLU Rectified Linear Unit.

RL Reinforcement Learning.

RNN Recurrent Neural Network.

SPAC Self-Play Actor-Critic.

TD Temporal Difference.

TRPO Trust Region Policy Optimization.

1 Introduction

Artificial Intelligence (AI) is a rapidly evolving field with countless of applications and ongoing research topics [2]. Its success revolves around its ability to learn how to address and solve problems, even those which are too challenging for humans to solve. Whilst certain problems can be represented formally with the use of mathematical rules, making it easy for AI agents to learn, other problems cannot be fully described formally, as humans usually solve these problems intuitively.

The concept of Machine Learning (ML) was introduced for the AI agents to learn to solve such tasks in a human-like manner, from experience and by comprehending the environment's features through structuring the gathered knowledge in a hierarchical manner [2], [3]. Such agents extract recurring trends from the provided data, which ultimately allows them to learn and build their own knowledge to tackle the problem. A subset of ML, which is more goal-oriented and which allows the agents to learn by interacting with the environment, is called Reinforcement Learning (RL) [4]. With its success in a wide variety of disciplines, RL quickly became one of the most researched branches of AI, especially in the field of applying AI on games.

RL has obtained remarkable success when applied to multi-agent adversarial games characterised by perfect-information and determinism, such as Chess, Go and Shogi, with the agents outperforming professional human players [5], [6]. The notable results significantly expedited the advancement of RL algorithms and serve as a promising avenue for tackling real-world challenges. The self-play RL techniques use the perfect-information and deterministic features to their advantage by executing a Monte-Carlo Tree Search (MCTS) to prune out the less promising search branches.

However, despite the outstanding progress made in the field of Multi-Agent Reinforcement Learning (MARL), certain multi-agent games may be characterised by features which present a challenge to the RL process [4]. Such features include partial-observability, stochasticity and when a game contains a large discrete action space. When a game is characterised by partial-observability, the RL agents do not have visibility on all of the environment state information. Meanwhile, when a game is stochastic, the agents cannot accurately predict the effect of applying a specific action on a particular state, especially when each action could have a large number of potential outcomes. Therefore, such characteristics present a layer of complexity to RL as the problem may become intractable to model and solve as a Partially Observable Markov Decision Process (POMDP) [4], [7]. Furthermore, when a game contains a large action space consisting of thousands or even millions of possible actions, the complexity of solving the RL problem increases exponentially ascribed to the extensive branching factor.

1.1 Problem Definition

The multi-agent Partially Observable Stochastic Game (POSG) "Jaipur"[1] was selected for this work. Jaipur is a competitive two-player card game in which both players take turns to trade or sell cards with the intention of getting the most points and winning the game.

The components of the game Jaipur during gameplay are displayed in Figure 1.1, which was taken from the game's digital application. The game consists of Camel cards and 6 different types of Goods cards. These cards are shuffled and stored facing down in the deck, ensuring that none of the players are aware of the card which would be dealt next. Moreover, the game also contains a 'marketplace', which is a space between both players that should always be populated with 5 cards facing up, in order for the cards to be seen by both players. Should a player take one or more cards from the marketplace, the marketplace will be replenished with the same number of cards dealt directly from the deck.

Figure 1.1 Digital Jaipur Gameplay

During gameplay, cards are only dealt to the marketplace from the deck. However, during the initial setup of the game, a number of random cards are dealt to each player. The players store Goods cards in their 'hand' and Camel cards in their 'herd'. Each player's hand can contain up to 7 cards at once and these cards are only visible to the corresponding player. Meanwhile, each player can have any amount of Camel cards in their herd. Both players have a choice on whether to display the number of Camel cards stored in their herd to their opponent, however irrespective of their choice, these cards must always be placed face up on the table. Furthermore, the game also consists of a set of tokens for each type of Goods card, as well as three other sets of Bonus tokens and 1 Camel token. These tokens all contain a specific number of points. The sets of Goods type tokens are sorted in descending order and placed face up, whilst the sets of Bonus tokens are shuffled and placed face down.

[1] Jaipur was developed by Sébastien Pauchon and published by Space Cowboys (Asmodee)

During each turn, the respective player can only perform one action, based on their knowledge of the game state information, from the following types of actions:

- take one Goods card from the marketplace,
- trade between 2 to 5 Goods cards from the marketplace,
- take all the Camel cards from the marketplace,
- sell Goods cards of the same type in exchange for tokens.

The player obtains points either by obtaining the mentioned Goods and Bonus tokens when selling Goods cards during gameplay, or by being in possession of more Camel cards than their opponent after the game terminates, and hence obtaining the Camel token. The game can terminate in any player turn, either when the deck is emptied and the marketplace contains less than 5 cards, or when there are no remaining tokens from 3 sets of Goods type tokens.

1.2 Motivation

The game Jaipur was chosen for this work as it's characteristics make it a challenging problem for self-play RL and it serves as a great example for further development to be carried out in the field of applying AI in POSGs.

The element of partially observability in Jaipur is a key feature, as a significant amount of game information is hidden from the player, making it harder for the agents to learn [8], [9]. The player's opponent hides most of the information from the player, and therefore, it is up to the player to keep track of the opponent's number of points and which Goods cards the player has in each turn. The player can attempt to keep track of the opponent's number of points by observing which tokens the opponent takes whilst selling cards. However, whilst the number of points of the Goods tokens are visible, should the opponent sell 3, 4 or 5 cards at once and take a Bonus token, the number of points of these tokens are only visible to the player who obtains them. Therefore, should the opponent take a Bonus token, the player has to calculate an estimate of the points taken by the opponent. Moreover, the player can keep track of the opponent's Goods cards by taking note of which cards are taken by the opponent from the marketplace or traded out to the marketplace. However, since the initial Goods cards provided to each player during the set-up of the game are dealt directly face down from the deck, the player can never be fully certain of what these cards are prior to the opponent either trading them out or selling them. Moreover, should the opponent also choose to hide the number of Camel cards, the player must also keep track of the amount of such cards during gameplay. The player must monitor the

opponent's trading actions and the number of Camel cards that the opponent takes from the marketplace. Furthermore, apart from lack of visibility around the opponent information, the player also has lack of visibility on the game's deck of cards. Given that these cards are shuffled and stored face down, the player does not have visibility on which card will be dealt next to the marketplace.

Jaipur also contains an element of stochasticity, as performing a specific action on a specific state may lead to a number of different outcomes [8], [10]. An example of such a stochastic action is when the agent opts to take all the Camel cards from the marketplace. There are a lot of possible card combinations with which the marketplace can be repopulated with. The cards can end up being all of high-value, or all of low-value, or a mix of both. Therefore, by performing an action, the player may inadvertently give the opponent an opportunity to take high-value cards.

Moreover, the game's very large action space is also a key element which poses a significant challenge to the RL process [8], [11], [12], [13], [14]. Jaipur has a total of 25,499 possible actions, which greatly complicates the agent's ability to learn the optimal actions to select during gameplay. As a comparison, the action space of the game Chess contains 4,672 possible actions [6]. Furthermore, in this Jaipur implementation, most of the actions from the action space are not valid in a given state, making the environment more intricate to solve [11]. The 25,499 possible actions are made up of the following actions:

- 6 actions for when a player takes one Goods card from the marketplace
- 1 action for when a player takes all the Camel cards from the marketplace
- 36 actions for when a player sells Goods cards
- 25,456 actions for when a player trades their cards with different cards from the marketplace

Furthermore, the game's actions provide the following different types of rewards and consequences:

- Immediate rewards with good long-term consequences. For example, obtaining the first Goods type token/s which contain the most points for that card type.
- Immediate rewards with bad long-term consequences. For example, selling 1 or 2 cards instead of holding onto them and waiting to sell 3, 4 or ideally 5 cards at once to obtain the Bonus token points.
- Delayed higher future reward. For example, accumulating cards to sell 3, 4 or if possible 5 cards at once to obtain the Bonus token points.

- Delayed lower future reward. For example, withholding from selling certain cards to obtain and sell 3, 4 or 5 cards at once, but the game terminates or the opponent manages to take the remaining tokens for that card type.

These different rewards and consequences further increase the complexity as the agent must thoroughly analyse the state to decide when to opt for an immediate reward and when to delay performing an action for a better long-term reward [10].

Lastly, given the game's adversarial element, different strategies can be adopted by the players based on their opponent's actions. Moreover, given that the opponents can have different play styles, this creates a dynamic environment where the RL agent can adjust their policy according to the play style of the current opponent. These elements also increased the complexity in similar work [11], [12], [13].

1.3 Aims and Objectives

The aim of this study is to successfully and efficiently develop intelligent AI agents on POSGs. Therefore, with the use of the game Jaipur, this work will focus on exploring the challenges brought about by such games to the AI learning process, as well as how these complexities can be mitigated through the use of different RL techniques. Moreover, to provide a deeper analysis of each technique, the agents will be trained with multiple RL algorithms on each implemented technique and they will be evaluated quantitatively and by analysing the agents' actions. This aim will be addressed by carrying out the following objectives:

Objective 1: Develop the Initial Environment with Action Masking
This objective will focus on implementing the initial environment along with the action masking method. This will investigate the success of action masking on POSGs and provide a basis to be used for the other experiments to be performed. The following sub-objectives will be carried out for this to be accomplished:

- Implement the Jaipur game environment and ensure proper logic flow consistent with the rules of the game
- Design the ideal action and observation spaces which adequately represent the game state information and implement a RL environment
- Make use of a RL library to implement different RL algorithms, as well as the action masking technique

Objective 2: Experiment with the Game Features and Hyper-Parameter Tuning
The aim of this objective is to explore how altering certain game features will affect the training and performance of the algorithms. Moreover, this objective also aims to

improve the performance of the trained models. Therefore, the following experiments will be conducted on which the algorithms will be trained and evaluated on:

- Increase the action space of the game by splitting the actions into further detailed actions
- Perform hyper-parameter tuning to optimise the performance of the algorithms
- Allow the agents to keep track of their opponent's gameplay information by increasing the observation space
- Provide the agents with full visibility of their opponent's gameplay information

Objective 3: Experiment with Different Techniques
This objective includes the analysis of different techniques and their ability to tackle the complexities brought about by the characteristics in POSGs, mainly large discrete action spaces. For this to be achieved, an in-depth performance evaluation will be conducted of the algorithms when trained on the following implemented techniques:

- Policy Cloning
- Action Embedding
- Hierarchical Reinforcement Learning (HRL) with the use of a Centralised Critic

1.4 Proposed Solution

This work proposes the implementation and evaluation of different RL algorithms and techniques to develop intelligent AI agents on POSGs. An initial environment with the action masking technique will be created combining the Jaipur game environment, a custom PettingZoo RL environment, with the appropriate action and observation spaces, and the use of the RLlib RL library for the Proximal Policy Optimization (PPO), Advantage Actor Critic (A2C), Deep Q-Learning (DQN) and Double Deep Q-Learning (DDQN) algorithms to be implemented. An increase in the action space of the game as well as hyper-parameter tuning will be carried out with the aim of improving the game environment and the performance of the algorithms. Moreover, two experiments will be implemented to analyse how the performance of the trained agents differ when more gameplay information is given to the agent. Finally, the policy cloning, action embedding and HRL with centralised critic techniques will also be implemented and evaluated to determine their efficiency when applied to POSGs with large discrete action spaces.

1.5 Contributions

This work aims to provide valuable insights to the field of applying AI on POSGs with large discrete action spaces. This study provides the following contributions:

- A MARL approach showcasing how AI agents can be efficiently implemented on multi-agent POSGs
- Three different techniques which can be used to address the exponential increase in complexity brought about by large discrete action spaces, which are action masking, action embedding and HRL with centralised critic
- A novel analysis comparing the performance of the three implemented techniques for handling large discrete action spaces
- A deeper insight on how partial observability effects the RL process
- An analysis of the policy cloning technique's performance
- The development of multiple successful self-play RL agents on the game Jaipur, using the PPO, A2C, DQN and DDQN algorithms

Given the characteristics of the POSG Jaipur, the contributions of this work may prove to be very beneficial not only to the field of applying AI in POSGs, but also when it comes to applying RL in real-world applications.

1.6 Document Structure

The remainder of this document is structured as follows:

Chapter 2 provides the necessary knowledge about the game mechanics of Jaipur, as well as about this work's related core techniques and libraries.

Chapter 3 delves into the methodology and results of work carried out in similar environments, as well as the different RL algorithms which proved to be successful on such similar work.

Chapter 4 explores the full details of the implementation carried out in this work to achieve the objectives.

Chapter 5 consists of a thorough explanation of how the implemented elements were tested and evaluated, along with a detailed discussion on the obtained results.

Chapter 6 provides a summary of this study, along with potential future work that can be carried out to further advance this work.

2 Background

This chapter includes the relevant background information needed to understand the research carried out in this work. This includes information about the board game Jaipur's game mechanics, as well as about the different RL techniques and libraries which are related to this study.

2.1 The Jaipur Board Game

The board game Jaipur, consists of 2 players, a total of 55 cards and 57 tokens. The cards are made up of 2 different card types; Camel (C) and Goods. Particularly, the game contains 11 Camel cards and the following Goods cards; 6 Diamond (D), 6 Gold (G), 6 Silver (Sv), 8 Silk (Sk), 8 Spice (Sp) and 10 Leather (L) cards, in descending order of value. Furthermore, for each type of card, Jaipur contains a number of tokens which are worth points, also referred to as Rupees. Specifically, the game consists of 5 Diamond, 5 Gold, 5 Silver, 7 Silk, 7 Spice and 9 Leather tokens, which amount to one less token than the respective number of cards for each Goods type. Moreover, each of these tokens has a specific number of points, according to the value of the card type, and the tokens in each of these sets are sorted in order from the highest to lowest in value. The game also contains 1 Camel token, which has a value of 5 points, to be awarded to the player who has the most Camel cards at the end of the game. Moreover, the game also consists of 3 other sets of tokens, containing 7, 6 and 5 tokens, for when the player either sells 3, 4 or 5 Goods cards at a time respectively. These tokens also have specific values, however, unlike the Goods tokens, they are shuffled individually, within each set, at the beginning of the game. Jaipur may also contain 3 Seals of Excellence tokens, which are not worth any points. These tokens are included as this game can be played as 1 to 3 game rounds. Therefore, should the players decide to play more than one round as a single game, a token is assigned to the winner of each round, to allow the players to keep track of the number of rounds won by each player to ultimately determine the final winner.

This turn-based game starts by allocating 3 Camel cards and 2 random cards from the shuffled deck to the marketplace, as well as 5 random cards from the deck to each player. The player will store their Goods cards as their hand, which should only be visible to the respective player and can contain a maximum of 7 cards at once, whilst their herd can consist of any number of Camel cards and can be visible to both players.

During each turn, the respective player can either choose to take one Goods card or all the Camel cards from the marketplace, trade cards with those present in the marketplace, or sell Goods cards. If a player selects to trade cards, they must ensure

that they exchange the same number of cards from their hand or herd with those from the marketplace. Furthermore, the type of the Goods cards taken from the marketplace must be different than that of the Goods cards that the player will remove from their hand. Meanwhile, should a player opt to sell cards, apart from only being able to sell Goods cards of the same type in each turn, they must sell a minimum of 2 cards simultaneously when selling Goods cards of type Diamond, Gold or Silver. This does not apply to the other types of Goods cards, as the player can choose to sell either 1 or more of these types of cards. Moreover, the players are unable to sell Camel cards, however they can either trade them in exchange for Goods cards from the marketplace, or they can collect them in their herd, with the intention of having more Camel cards than their opponent when the game ends in order to obtain the Camel token points.

After a player performs an action in their turn, if either the marketplace cannot be filled again with a total of 5 cards, as the deck would be emptied, or three sets of Goods type tokens finish, the game ends. The 5 Camel token points are then assigned to the player with the most Camel cards and the points are compared for the player with the most number of points to be determined as the winner.

2.2 Machine Learning

ML is a subset of AI that was developed to provide the AI agents with the ability to be trained and improved from experience data, which is the computational version of human experience [2], [3]. ML algorithms learn by extracting data from the environment and identifying patterns within it. The trained models would then have the ability to create predictions and make informed decisions when presented with new observations. ML is a very powerful and beneficial technique, as apart from enabling automation on environments where human-like experience is required, it is also able to create accurate predictions, some of which not even noticed by humans, from very large amounts of data which ultimately greatly enhances decision-making.

2.3 Deep Learning

Deep Learning (DL), which is a subset of ML and consists of either supervised, unsupervised or semi-supervised learning, makes use of representation-learning methods, along with deep neural network architectures made up of multiple processing layers [2], [15], [16]. Specifically, DL comprises multiple techniques, such as batch normalisation, dropout and non-linear activation functions including Rectified Linear Unit (ReLU), sigmoid and softmax, that enable the efficient and effective training of Neural Network (NN)s with many hidden layers [2]. These multiple processing layers

allow the DL models to learn complex data representations, with different levels of abstraction [16], by directly correlating the input with the target labels [15]. In DL, each sample provided in the input data is considered to be Independent and Identically Distributed (IID) from one another [15].

A general-purpose learning algorithm is used to teach these computational models to extract high-level features from the provided raw sensory data, by performing a vast number of data abstraction levels [16] and deriving approximations related to the environment [17]. Moreover, DL makes extensive use of the back-propagation method, which is fundamental to specify how the internal parameters, which calculate the representation in each layer from the previous layer, should be changed [16]. This enables the model to learn complex structures from the provided data whilst facilitating effective training and optimisation of the models.

The breakthroughs brought about from these techniques in how sensory data, such as images, speech, text, audio and video, can be processed have benefitted multiple fields, such as object detection and recognition, natural language processing, drug discovery and genomics, greatly [16]. Moreover, the ability to learn intricate information from high-dimensional data, without requiring a lot of dependency on humans, makes it applicable and useful to many problems in real-life domains [16].

2.4 Reinforcement Learning

RL is also a subset of ML where the training is performed by an agent interacting with the environment [17], [18]. In RL the agent considers the entire problem as a whole rather than splitting it into sub-problems and solving each sub-problem individually [4]. According to Sutton Bartol et al. [4], RL is the most similar, out of all types of ML, to how living beings learn. Furthermore, most of the core RL algorithms were based on biological learning mechanisms.

A RL environment contains the states, possible actions, and reward signal. The agent interacts with the environment to learn and dictates which actions will be applied. Moreover, the agent consists of a policy, and may also have a value function, according to the RL algorithm being used. The reward signal determines the problem's goal and provides immediate feedback to the agent on whether an action was good or not. The value function estimates the future expected accumulative reward and hence, helps the agent understand what is beneficial in the long term. This is very important as in RL, an action might not have an immediate reward, however, it may affect the subsequent states and hence result in a delayed reward [4], [15]. In certain instances, one may also have the environment's model, which allows deductions to be made on the environment's behaviour [4].

The exploration-exploitation trade-off is an important challenge in RL as, during training, it is crucial for the agent to balance exploration and exploitation of the policy [4]. The agent must exploit actions which have returned great results in the past, however, it must also explore new actions to potentially achieve an even better result.

During training, the agent performs the loop shown in Figure 2.1. It observes the environment, conducts exploration or exploitation of its policy to select an action, and performs the selected action on the environment. Upon doing so, the agent will take into consideration the new state as well as the received reward, and based on the observed feedback, the agent will learn whether the action performed is good or not for that state. Therefore in RL, the policy is learned through trial and error and with the use of the reward hypothesis, which means that the agent's goal is to select and perform the optimal actions which maximise the sum of all rewards [4]. Moreover, in RL, a correlation between similar states is learnt during training and upon learning new information, a change is easily applied to how the data is distributed [15]. These reasons make RL techniques ideal for this work.

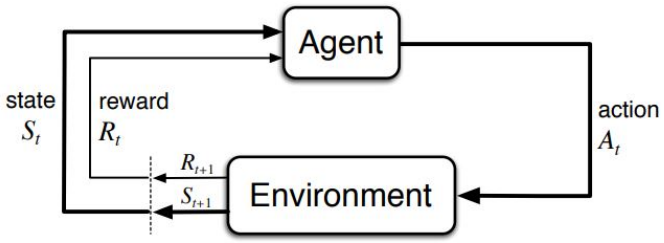


Figure 2.1 Reinforcement Learning Environment Interaction Loop [4]

To enable interactive learning in high-dimensional environments, Deep Reinforcement Learning (DRL) was created by combining RL with DL [17]. Compared to traditional RL, DRL makes use of Deep Neural Network (DNN)s to approximate the action-value function in environments characterised by large action and/or state spaces. Hence, such models are capable of handling very large amounts of data which are unstructured. This is performed through trial and error interaction on the environment with the use of the reward hypothesis, as used in the RL models, to obtain the optimal cumulative reward, whilst also performing approximations, similar to the DL models, on the obtained feedback to further optimise the policy.

2.5 Multi-Agent Reinforcement Learning

MARL refers to RL performed on an environment which is shared by multiple agents. Such environments can either be competitive, cooperative or a mix of both [19]. In a competitive environment, the agents need to work individually towards their own goals and select actions for their own benefit only. Meanwhile, in a cooperative

environment, the agents would have the same goal and hence, they would need to work alongside each other. Regardless of the type of MARL environment, each agent would have their own observations, rewards and actions. However, there may be a case where the agents share a policy. Moreover, MARL environments tend to work by taking turns for each agent to obtain their own observation, select and perform an action and receive a reward. Therefore, the changes made to the environment which resulted from the action performed by an agent would be reflected in the observation provided to the next agent [20].

2.6 Gymnasium

Gymnasium[1] is an open-source RL library for single-agent environments, developed as a maintained fork of the OpenAI Gym library [21], [22]. This library presents a simple interface which is capable of converting environments into single-agent RL environments. With the use of this Application Programming Interface (API), custom environments can be created from scratch. However, this API provides a variety of ready-made environments and examples which can be experimented with, modified or used as a reference when implementing single-agent RL environments [21]. Apart from this, Gymnasium also offers various environment wrappers for an environment to be altered without the user having to directly modify the underlying code [21].

Each Gymnasium environment, which is developed in Python, is organised as a single class structure consisting of 4 key functions and 6 variables [23]. This is performed for the RL agent-environment loop, which is displayed in Figure 2.1, to be implemented for each episode within the environments. An important feature of the Gymnasium API is that it provides a wide range of spaces to be used for action and observation spaces [21]. These are very beneficial as they provide crucial information, such as the type and quantity, of the actions that can be performed within the environment and the observations that are extracted from the environment. Moreover, unlike the Gym library, the ability for an environment to be truncated, apart from terminated, was implemented within the Gymnasium library. An environment can terminate due to an error being raised during runtime. However, if the environment is configured appropriately, it will terminate when the desired goal or process is fulfilled. Meanwhile, if the environment exceeds a specified number of steps or amount of time, it can be truncated instead of terminated. This would allow for a clear distinction between the two cases, which is beneficial for the agent's learning [21], [23].

[1] https://gymnasium.farama.org/

2.7 PettingZoo

With the aim of accelerating MARL research, the PettingZoo[2] MARL environment library was developed [24] to offer a simple and easy to implement interface for MARL. This library was developed based on the OpenAI Gym and Gymnasium libraries and hence, its interface is very similar to the interface provided by Gymnasium. In fact, PettingZoo makes use of the spaces defined by the Gymnasium library for the observation and action spaces. However, since the PettingZoo API handles the training of multiple agents within the same environment, it makes use of some additional variables and functions, when compared to the Gymnasium API [24].

Similarly, to the Gymnasium API, PettingZoo offers a number of different ready-made environments and examples to be experimented with or used as a reference for the development of custom environments. Moreover, it also provides the truncation feature, as implemented in Gymansium's API, for each agent within each environment. Furthermore, to facilitate transformation or validation modifications to be carried out on the MARL environments, PettingZoo offers multiple utility and conversion wrappers which can be easily applied within the environments.

The PettingZoo library contains two different types of MARL environments which consist of the Agent Environment Cycle (AEC) and the Parallel environments. The AEC API is used for environments which are turn-based in a sequential order whilst the Parallel API is suited for environments which consist of actions being performed simultaneously by multiple agents. The implementation of the AEC API by PettingZoo resulted in the establishment of the AEC game mode for multi-agent games [24], which is further described in Section 2.9.

2.8 Partially Observable Stochastic Games

The POSG game model refers to a multi-agent game which is characterised by partial observability and stochasticity [24]. The element of partial observability is present when not all the gameplay information is visible to the players. Meanwhile, a game is said to be characterised by stochasticity when applying the same action on the same game state may lead to different outcomes. All the agents in a POSG game model will perform their separate actions, observe the environment and receive their individual rewards together [24]. Horák and Bošansky [25] state that this type of model is one of the most versatile formal models capable of handling dynamic multi-agent environments. Despite this, from the research performed by Terry et al. [24] to develop the PettingZoo library, it was revealed that when the POSG models are used within

[2]https://pettingzoo.farama.org/

code-based games with multiple agents, they tend to not be conceptually fit. Therefore, the implementation of the AEC game model was brought about.

2.9 Agent Environment Cycle

The AEC model was developed as a more conceptually sound model to be used with code-based games [24]. Whilst in the POSG model, the multiple agents take in the observation, perform an action and receive a reward together, in the AEC model the agents will perform these actions in a sequential order, one after the other [23], [24], as displayed in Figure 2.2. Therefore, in a game with an AEC model, each agent will take in the environment's observation, select and perform the action based on their own policy and receive their own respective rewards based on the action that they performed [24]. Once an agent is finished with these steps, the next agent will get their turn and this process is repeated until the environment terminates or truncates.

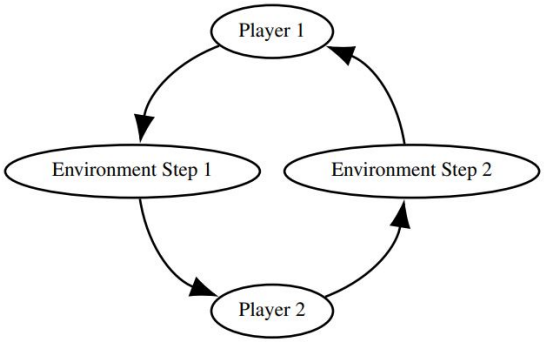


Figure 2.2 AEC Diagram of a Two Player Turn Based Environment [24]

Due to the AEC model's distinctive quality of sequential stepping, apart from serving as PettingZoo's basis, this model is especially well-suited for the majority of MARL APIs [24]. Moreover, given that the POSG and AEC models are very similar, Terry et al. [24] proved that both models are capable of representing both AEC and POSG environments.

2.10 Ray

Ray[3] is an open-source Python library which provides a number of high-level libraries related to ML such as RLlib[4] and Tune[5].

[3]https://docs.ray.io/en/latest/index.html
[4]https://docs.ray.io/en/latest/rllib/index.html
[5]https://docs.ray.io/en/latest/tune/index.html

2.10.1 RLlib

RLlib is a highly scalable library which offers a variety of different RL algorithms that are able to be implemented in an efficient manner within RL environments [26], [27]. Furthermore, this library also offers scalable abstractions which allow and aid the creation of new algorithms [27], [28]. The RLlib API contains wrappers to ensure compatibility with MARL environments, such as PettingZoo environments. This API also integrates ML frameworks including PyTorch and Tensorflow [28], [29]. The development of this library enabled rapid development in the field of RL [27]. Apart from accomplishing state-of-the-art performance in various RL and MARL environments, this library also presents countless of examples related to RL training, to further facilitate user implementation.

2.10.2 Tune

Tune is a highly efficient library for hyper-parameter turning, as well as experiment management of ML models [30]. This framework was developed to facilitate the development and implementation of hyper-parameter search algorithms as previously, most search algorithms were developed individually, pertaining to specific frameworks as closed source and required a considerable amount of computational resources. Therefore, this library contains a wide range of state-of-the-art hyper-parameter search algorithms including Population Based Training (PBT) and ASHA. Tune also integrates with various optimisation tools such as Optuna, BayesOpt and Ax, as well as ML frameworks such as PyTorch, TensorFlow and Keras. Moreover, this library provides efficiency for large-scale experiments as it allows for the search algorithms to be scaled.

2.11 Population Based Training (PBT)

PBT is a computationally efficient and easy-to-integrate hyperparameter optimisation algorithm that combines the popular random search and manual tuning techniques [31]. This algorithm is able to systematically obtain the ideal hyperparameters by training multiple NNs in parallel with the use of evolutionary principles. The information about each model's performance is shared across the networks for the hyperparameters to be iteratively refined. Any algorithm-specific hyperparameters that can be modified during training without reinitialising the model can be tuned with PBT. For each specified hyperparameter, the models start off by assigning random values from the provided list or range. Then, during optimisation, the algorithm will determine which model is performing the best, according to the selected metric, and the lower-performing models will be replaced with a copy of the best-performing

model. The new models' hyperparameter values will either be perturbed or will be set to other random values from the provided list or range. This exploration balance, whilst prioritising the best performing models, allows the algorithm to efficiently obtain the ideal values. Moreover, computational resources are automatically assigned to models that perform better.

2.12 Summary

This chapter provided a detailed explanation of the game Jaipur's rules and components, which is necessary to comprehend the game's characteristics that make it ideal for this work. Moreover, the core knowledge related to ML, DL, RL, DRL and MARL was presented in this chapter, along with brief information about the AEC and POSG game models. Furthermore, this chapter also provided an introduction to the main libraries used in this work. The literature review, presented in the next chapter, delves into research that has been performed on how similar problems were tackled, as well as on different algorithms suitable for this work.

3 Literature Review

This chapter provides an overview of studies which were conducted on games with characteristics similar to those of Jaipur, since as of yet, there are no studies that explore how RL can be applied successfully in this game. The different methodologies and algorithms adopted on such similar games, along with their results, are explored. In addition, an in-depth explanation of the different RL algorithms which can be implemented in this work is also provided.

3.1 Related Work

Given that as of yet, no publicly accessible research has been published on how RL should be performed on the board game Jaipur, research for this work was conducted on studies which explored RL on games or other environments with features similar to Jaipur. The similar features included having multiple agents, being partially observable, having stochastic actions, containing both long-term and immediate rewards or consisting of a large action space. This research was performed to gather an understanding of the challenges brought about by the mentioned features during the RL training, as well as to explore which techniques and algorithms addressed these challenges and achieved the best results.

Zha et al. [8] explored the implementation of RL on a multi-agent card game called DouDizhu. This game features characteristics very similar to those of Jaipur such as partial observability, stochasticity and a very large and complex action space consisting of 27,472 possible card combinations. However, whilst Jaipur is a two-player competitive game, DouDizhu consists of three players and has both competitive and collaborative aspects between these agents, where two of the players must work together to play against the other player. Moreover, DouDizhu also consists of a very large and vast observation space. In this work, a new algorithm called DouZero was implemented which enhances the conventional Monte-Carlo techniques to ensure resistance against over-estimation bias, by incorporating parallel agents, action encoding and DNNs. Moreover, this algorithm depends on sampling to make use of further complex neural architectures, which ultimately allow it to be able to generate a considerably greater amount of data per second with the same computational resources. Figure 3.1 displays the implemented Q-network structure including how the action encoding was applied to the environment.

The use of action encoding was a crucial addition to this work due to the large complex action space of the game. This allowed for the algorithm to be able to generalise over actions which were rarely encountered during the learning process.

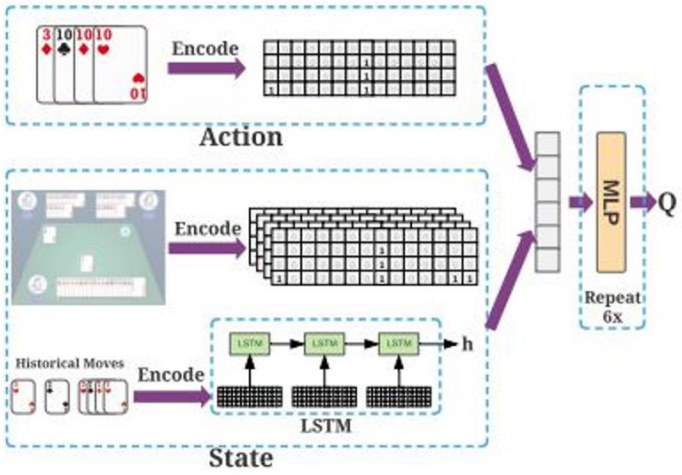


Figure 3.1 Q-network of the DouZero Algorithm [8]

The action encoding was applied to represent the game's card combinations, within both the state and action spaces, with the use of a one-hot encoding matrix of size 4 x 15, as displayed in Figure 3.2. The columns in this matrix represent the different types of cards present in the game, whilst the rows correspond to the amount of cards for each type. As shown in Figure 3.1, multiple of these matrices were used to represent the state information of the environment whilst the action is encoded into one matrix. Apart from the current state, the past state-action pairs are also encoded, by utilising a Long Short-Term Memory (LSTM), and concatenated with the matrices displaying the current state and action information. A Multilayer Perceptron (MLP) consisting of six layers and a hidden dimension of 512 is used to compute the Q-values.

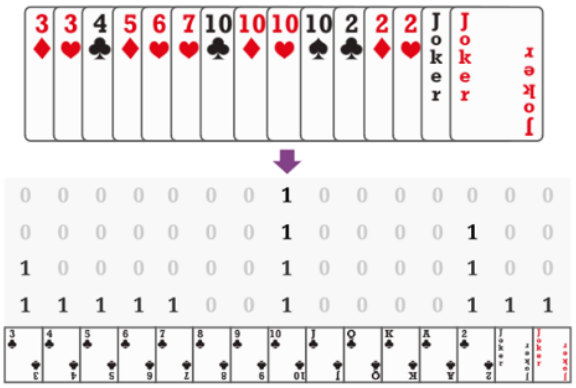


Figure 3.2 One-Hot Encoding Matrix for DouDizhu's Card Combinations [8]

Meanwhile, Yao et al. [11] introduced a hybrid RL framework to successfully develop RL agents on the multi-agent game Axie Infinity by performing action embedding to handle the game's large discrete action space. This game, which is a two-player competitive card game, consists of a very large action space of 23,149,125 possible actions which are not all valid in a given state. Moreover, the game also has a large number of possible strategies that can be adopted as, apart from its competitive

element, it also contains multiple different teams that a player can select. In the work performed by Yao et al. [11], Axie Infinity is initially formulated as a single-agent Markov Decision Process (MDP) and an action encoding technique is developed which represents each action as a 6 x 12 matrix. Moreover, the embedding function is implemented with a supervised learning technique to be able to learn the action representations according to the effects caused by the action on the state of the environment. The proposed technique, which is displayed in Figure 3.3, will initially determine the goal action based on the policy function. Moreover, the embedding function is used to map the encoded actions into a continuous space, and based on the Euclidean distance function, a small set of actions similar to the goal action will be extracted from the large action space. A state-action value function is then utilised, on the small set of similar actions, to obtain the highest valued action as the optimal action to be performed on the state. Unlike the embedding function, the policy and state-action value functions are updated iteratively during training in a manner similar to Deep Deterministic Policy Gradient (DDPQ). The goal action is used to update the policy function and the state-action value function is updated with the use of a Monte-Carlo estimate.

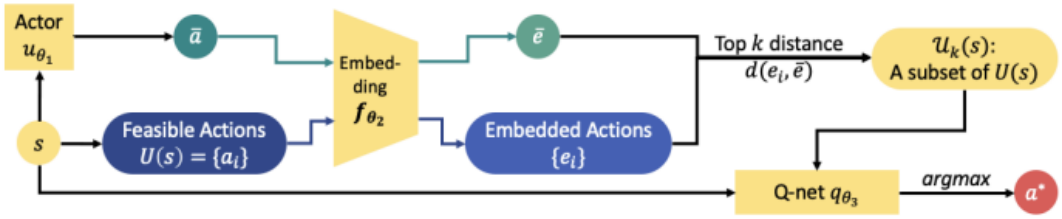


Figure 3.3 Decision Process to Apply RL Agents within Axie Infinity [11]

Furthermore, to evaluate the performance and sample efficiency of the proposed approach, the DouZero and DouZero+pooling methods were modified and applied to the game environment for the results from these three methods to be compared. Six teams with different levels of popularity were selected and three models were trained for each team, one with each of the mentioned methods. All the models were stopped at the same number of time steps, $1 * 10^7$, to allow for a better comparison to be made. During the training, the models being trained with the proposed method were obtaining the highest mean reward, for the same amount of trained samples. Moreover, the trained models were played against each other for 29,000 games each. Following the simulated games, when the models were played against different teams, the models trained with the proposed method returned an overall higher winning rate. Moreover, when played against the same team, the models trained with the proposed method surpassed the performance of the other models in all the teams, except for teams 2 and 6 when played against the models trained with

the DouZero method. These two teams consist of more diverse strategies which cause the high-reward actions to be embedded away from each other in the continuous space. Therefore, in this case, the DouZero method was more suited given that it evaluates all the feasible actions for each state, despite using higher computational resources. The obtained results displayed the proposed method's good ability to generalise across the different teams, as well as to produce high-performing models, which are efficient in sampling and make decisions in a less amount of time.

AlphaGo, AlphaGo Zero and AlphaZero are all algorithms that were developed to play the popular game "Go", which is a 2-player dynamic strategy board game with a large action space [12], [13]. However, unlike Jaipur, this game consists of perfect information. Whilst the three mentioned algorithms are based on the MCTS, DL was integrated by using DNNs to enable the MCTS to manage the game's large complexity. Furthermore, with an effort to create a less challenging AI from the AlphaZero algorithm for human players, the AlphaDDA AI was proposed by Fujita [12]. This new algorithm also makes use of DNNs implemented within MCTS however, this algorithm will intentionally opt for suboptimal actions with the purpose of adapting the algorithm dynamically based on the opponent. The algorithm will adjust according to the estimated game state value with the intention of providing the human player with an opportunity to win. This creates a more balanced gameplay for human players whilst still remaining challenging.

The importance of being able to apply RL on environments characterised by large discrete action spaces is crucial for RL to be applied on real-world problems. Therefore, a study was proposed by Dulac-Arnold et al. [14] where a new policy architecture, titled the "Wolpertinger architecture" was developed. This policy learns efficiently, with a large number of actions, by generalising the actions instead of having to analyse all the possible actions, given that this would be very time and resource consuming. As illustrated in Figure 3.4, this technique was developed on the actor-critic method. It makes use of multi-layered NNs to approximate the function for both the actor, which generates the proto-action, and the critic to refine the policy. It embeds the environment's actions in a continuous space, with the use of prior knowledge about the actions, in order to be able to generalise them. Moreover, to ensure a manageable training time, this technique implements an approximate nearest neighbour search within the continuous space, to extract a set of the closest actions mapped to the Discrete action space in logarithmic time. Moreover, the ideal action to be performed is selected with the use of the arg max function which obtains the action with the highest Q-value from the extracted set of closest actions. Moreover, for the policy to be trained, the DDPQ algorithm was used. This method was evaluated on a number of environments with an action space of up to one million where it achieved great results. Therefore, the results obtained show that this policy can be applied to

real-world problems as considering a sub-set of the full action space is not only less time consuming but also abundant for the agent to learn in a large variety of environments. However, this implementation cannot be applied when agents need to explore and learn the environment from scratch.

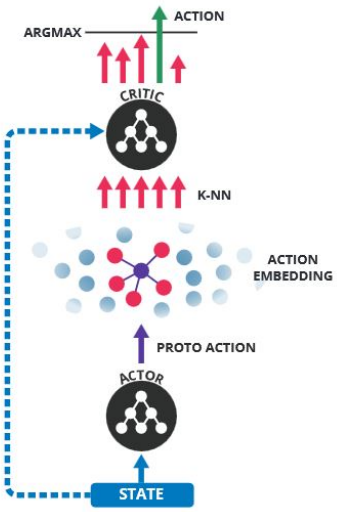


Figure 3.4 Wolpertinger Architecture. [14]

Moreover, to handle the challenge brought about to RL environments due to large discrete action spaces, the Cooperative Modular Reinforcement Learning (CMRL) method was introduced [32]. This technique was developed as an action division-based DRL method which performs task decomposition to handle complex action spaces. The technique consists of initially modelling the environment as a MDP and then making use of two modules which are applied on the modelled environment. The first module will perform action division, by utilising a generic rule-based division algorithm, to decompose the environment's large action space into a number of sub-tasks, with each sub-task containing a much smaller amount of actions that the RL agents can handle efficiently. Therefore, the MDP is converted into a multi-modular MDP, with each sub-task represented as a small MDP. The second module is then implemented, consisting of distributed learning to tackle these sub-tasks individually. This is performed with the use of multiple modular critic networks conducting training independently and in parallel. Each critic network obtains the optimal local action of the respective sub-task by learning a decomposed value function. Ultimately, the optimal global action is selected as the optimal action from the set of optimal local actions, obtained from all the sub-tasks. The CMRL technique was implemented on multiple environments which all consist of a large discrete action space. The selected environments included the Snake and Cartpole games, a coordinated multiple Electric Vehicle Groups (EVG) charging scheduling task and an energy management task for an industrial park represented as a microgrid. These environments were all specifically

chosen to represent different fields in which RL can be applied, to test the CMRL method's performance and ability to generalise across a variety of environments. Within these environments, apart from the CMRL method, the Critic, Diffusion Policy Policy Optimization (D-PPO), DQN and Distributional Soft Actor Critic (D-SAC) centralised DRL techniques were also applied to serve as a comparison while evaluating the learning efficiency and result accuracy of the proposed method. The results obtained from the simulated experiments displayed that the CMRL underwent more stability during the training process, managed to converge faster as well as obtained greater action selection accuracy than the other state-of-the-art techniques.

The study performed by Kanervisto et al. [33] discusses how the complexity brought about by large action spaces can be addressed. It evaluates the benefits of applying three different techniques on five video games with large action spaces, whilst training the agents with the PPO algorithm. The first technique consisted of removing unnecessary or harmful actions from the action space. This proved to be very beneficial for the RL process, as it improved the exploration and sample efficiency. However, this technique requires previous knowledge about the environment and it may also decrease the agent's competence. Meanwhile, the second and third techniques experimented with converting a continuous action space and a multi-discrete action space into a discrete action space. When the continuous actions were discretised, the self-play learning process was also improved. Having a continuous action space proved to complicate the RL agents' training process, as well as potentially hindering their progress. However, with the third technique the performance of the agents was not improved and this technique brings about the potential of the action space being scaled poorly. Finally, Kanervisto et al. [33] also suggested to incrementally add removed actions, if the agents manage to tackle the environment efficiently, with the aim of improving their performance.

Meanwhile, Zahavy et al. [34] introduced a new technique combining the DRL algorithm, DQN, with an Action Elimination Network (AEN), to remove the sub-optimal actions during the training. The AEN is trained and makes use of an external elimination indicator, produced by the environment, to predict which actions are invalid on the given state and hence, mask them out for the policy to not consider them. This technique obtained great performance, with a significant decrease in the training time as well as an increase in robustness, compared to the traditional DQN method, especially when applied to text-based games with a large discrete action space.

Similarly to the technique introduced by Zahavy et al. [34], the studies conducted by Tang et al. [35], Liu et al. [36] and Ye et al. [37] explored the implementation of action masking. This technique filters out the inapplicable actions for each state during the RL training process by passing an action mask within the feedback to the agent.

The study performed by Tang et al. [35], focused on evaluating the benefits of the action masking technique within RL. Given the great performance obtained by the PPO algorithm in RL environments, action masking was implemented within this algorithm. For each environment step, the environment was modified to return a mask highlighting which actions are valid and invalid in the particular state. The agent was then updated to ignore the actions marked as invalid, by only considering the valid actions when collecting the trajectory and when optimising the policy using stochastic gradient descent. Moreover, given that the agent ignores the invalid actions, a re-normalisation procedure is performed on the probability of the valid actions by applying a softmax activation function on the valid actions only. For the difference in performance to be evaluated, the PPO algorithm was implemented with and without action masking, on two separate environments with different complexity levels. Within both environments, when PPO was trained with the use of action masking, it achieved greater performance with a decrease in training time and a higher obtained return.

Multi-Player Online Battle Arena (MOBA) competitive two-player games consist of complex dynamic environments with large action and state spaces [37]. Therefore, to enable efficient RL agents within such games, a new scalable and off-policy DRL framework was developed and applied to the Honor of Kings game [37]. One of the main components within the technique is the use of action masking to prune out the invalid actions during training and, in turn, decrease the exploration complexity brought about by the large action space. The framework also makes use of control dependency decoupling, to simplify the policy training by separating the action components, as well as an attention technique, to improve the target selection during the combat phases of the game, and an LSTM, for the agent to learn skill combinations. Moreover, it also makes use of a dual-clip PPO algorithm to ensure convergence during training when deviated and large batches are used. With these components, the proposed algorithm achieved state-of-the-art performance, surpassing that of professional human players and previous MCTS methods. Moreover, the use of action masking allowed the agents to achieve the same level of performance in much less time.

With the aim of decreasing the training time taken by DRL agents to tackle decision making tasks, particularly related to the complex field of tactical driving, Lui et al. [36] proposed and applied three techniques within a realistic driving simulator. Given the partial observability aspect of environments within this field, such problems are typically modelled as a POMDP, however, typical DRL techniques still bring about certain challenges. Therefore, the first proposed technique consisted of applying action skipping instead of frame skipping with action repetition, as the latter causes instability. This technique allows for a number of null actions to be returned following an actual action, for the agent to be able to gather further experience. The second technique modifies the reward function to penalise the agent for performing certain

actions which are not ideal or which lead to risks, such as driving in a bicycle lane or running through a red light. The values provided to the agent as a penalty differ according to how dangerous or inefficient an action is. Finally, the third technique makes use of rule-based action masking where instead of hoping that the agent does not choose to perform certain undesirable actions, these actions are immediately filtered out during inference. Apart from decreasing the training time, this technique would also increase safety and reliability of the model by preventing the agent from performing dangerous actions, as even though the agent may learn not to select such actions, there would still be a possibility of them being selected due to noise within the observation. Moreover, this study incorporates these mentioned techniques within a HRL architecture. This architecture contains a learning-based module and three non-learning modules which are used to handle different elements of the problem. The learning-based module takes care of the decision making within the environment as it consists of a DRL agent that observes the environment through top-down RGB images and which makes use of Dueling DQN to learn. Meanwhile, the non-learning modules are used to handle tasks related to routing, planning and control separately. The results obtained from this work displayed that the proposed techniques proved to be very beneficial as they not only increased stability and efficiency within the DRL process, but also increased the safety aspect of the trained model.

The implementation of HRL was explored in the studies conducted by Zhang et al. [38] and Guo et al. [39], HRL is used to tackle complex RL environments by diving the main task into multiple more manageable sub-tasks to be handled separately. By simplifying the RL process, this technique also increases efficiency as well as decreases training time and action complexity. Moreover, intermediate rewards are usually applied within this technique, for the agents to learn how the sub-tasks should be tackled, which are beneficial to the RL process. Furthermore, depending on the environment, the sub-task policies may be reused across multiple tasks.

Zhang et al. [38] applied a new HRL framework to the MOBA King of Glory game which has both competitive and cooperative elements as it consists of either two players playing against each other, or ten players split into two teams. The HRL architecture, displayed in Figure 3.5, consists of the agent first selecting a macro action and then, based on which action is selected, the agent will select a micro action to perform. The applied framework makes use of imitation learning for the macro strategies to be learned, as this part of the game is computationally complex and has sparse and infrequent rewards, making it very challenging for self-play learning to be performed from scratch. Moreover, RL is applied to optimise the policy of the micromanipulation tasks. The implementation of dense reward function as well as a multi-target detection method, to represent the game's state with global features, were also applied within the environment. The results obtained displayed a greater

performance when compared to RL applied with the PPO algorithm.

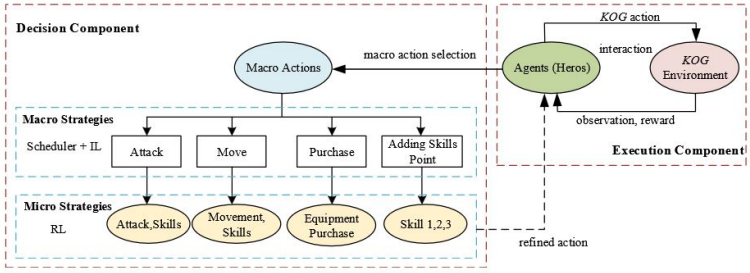


Figure 3.5 HRL Architecture for the King of Glory MOBA Game. [38]

The study carried out by Guo et al. [39] focuses on applying RL within multiple scenarios of autonomous driving by making use of a HRL option-based policy switching technique. The applied HRL structure makes use of a high-level policy to alternate between different driving strategies based on the encountered scenario. This technique employs bottom-up training where the low-level policies, which control the different driving strategies, are first trained separately through RL, with the use of the PPO algorithm, and different reward functions according to the scenarios. The high-level policy is then trained within an option-critic framework to learn when to terminate and switch low-level policies during the interaction with the environment. The applied technique was compared with and exceeded the performance of the rule-based policy switching technique, where the driving strategy is switched upon encountering a specific condition.

Moreover, Pateria et al. [40] and Hutsebaut-Buysse et al. [41]. carried out research related to the advancements made within the field of HRL, along with the challenges brought about by this technique. HRL has been making significant advancements, in various fields such as multi-agent RL. However, it has two main challenges which include how agents learn hierarchical policies as well as how subtasks can be automatically discovered. These challenges stem from such technique's limitation of being too dependent on the environment's structure. Based on the environment, such techniques may not be able to be scaled or generalised and may require further amount of training.

Meanwhile, to enhance the training process of self-play RL agents on partially observable multi-player games, the Self-Play Actor-Critic (SPAC) technique was developed [9]. As can be seen in Figure 3.6, which displays the architecture of this method, the SPAC was derived from actor-critic and policy gradient techniques with the incorporation of a comprehensive critic. Within the SPAC method, the comprehensive critic takes into consideration the opponent's actions and observations or the observations only. The underlying principle which led to this decision was that human players take note of their opponent's observations and actions during gameplay

and therefore, this should also be the case for RL agents. Similar to how humans interact and learn, the comprehensive critic will exploit the comprehensive observations to learn. However, when it comes to selecting an action to be performed within the environment, the choice will be based on the individual player's observation only. Moreover, during the SPAC learning process, following each simulated game's termination, the policies of the agents are overwritten with the winning player's policy. If the game ends with a tie, the policies will be updated separately whilst also taking into account the opponent's actions and/or observations. Furthermore, the policies will be updated with the use of similar architecture of PPO for environments characterised by stochasticity and DDPQ for environments which are deterministic.

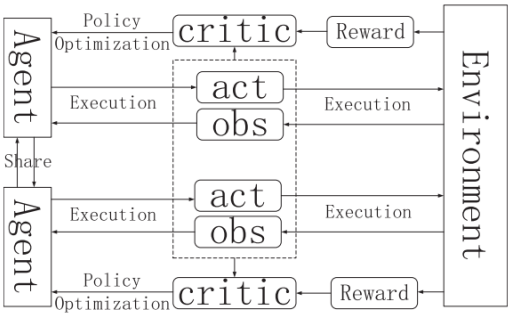


Figure 3.6 SPAC Architecture. [9]

An evaluation of the performance of this technique was conducted on four ready-made RL environments of different multi-agent adversarial and cooperative games, namely Soccer (2 v 2), Tennis, Pong and Adversarial-Push. Moreover, this study evaluated the effectiveness of the comprehensive critic by training the agents on the mentioned environments twice. The first experiment consisted of making use of the SPAC technique, hence with the use of the comprehensive critic, which in this experiment, took into consideration both the actions and observations of the opponent. Meanwhile, the second experiment was performed without the comprehensive critic where the application of either the PPO or DDPQ algorithm was used, based on the environment's characteristics, given that the SPAC employed an architecture similar to these two policies to perform the policy updates. Moreover. this work noted the benefits of applying a deterministic policy to environments with deterministic state transition features and a stochastic policy for environments with an element of randomness. Therefore, given that the Pong environment is stochastic, the SPAC with the stochastic policy and the PPO algorithm were applied separately on the environment for the experiments to be conducted with and without the comprehensive critic respectively. Meanwhile, the other three environments are considered to have a deterministic state transition. Therefore, on these environments, for the experiment to be conducted with the comprehensive critic, the agents were trained with the SPAC with the deterministic policy. Meanwhile, for the experiment

without the comprehensive critic, the DDPQ algorithm was used to train the agents. For a more sound evaluation, when the experiments were conducted without the comprehensive critic, the policies of the agents trained by the PPO and DDPQ algorithms were still being overwritten with the winner's policy, as performed within the SPAC policy. From this experiment, SPAC demonstrated better performance than PPO and DDPQ. Given that the policy updates performed by SPAC correspond to those of PPO and DDPQ, the results determine that the use of the comprehensive critic increases the performance of the self-play RL agents. Moreover, this study also evaluated the difference in the SPAC performance in competitive and cooperative settings. This was conducted on the game Pong, with immediate rewards, since this game can be played in both settings. The results of this experiment showed that SPAC had better performance in the cooperative version of Pong, which stems from the fact that in such an environment, the agent is more reliant on the opponent's observations and actions when compared to an adversarial environment. However, the performance obtained with this technique still surpasses the performance of PPO in both settings. Finally, another evaluation was also carried out to explore the efficiency of SPAC when only the opponent's observations are taken into consideration. The experiment was performed on the environment with the highest complexity, which was the Soccer (2 v 2) game, and from the results, it was determined that the agents perform better when both the opponent's observations and actions are provided. From this experiment, Lui et al. [9] concluded that by providing the agents with more comprehensive information about the environment, the better the resulting trained agents' performance will be.

To obtain a deeper insight into the effects of partial observability on the performance of RL models, a more visual approach was applied [42]. The Atari Pong game environment was used within this study. This environment was modified to be partially observable with the application of occlusion masks, both horizontal and vertical, which conceal the environment's raw pixels to eliminate two-thirds of the gameplay state information that the RL agent can observe. Given that the aim of this work is to gain a deeper understanding on which data the RL agent determines to be the most important for the action selection decision to be based upon, as well as their ability to learn to select the ideal information to observe, the agent's training was performed completely from scratch where they were not provided with any information on which gameplay data is important. Therefore, a DQN algorithm was implemented with a Convolutional Neural Network (CNN) to allow the agent to learn, from the game's raw pixels, which gameplay information is the most beneficial to observe as well as the ideal actions to be performed. In this game environment, the agent needs to select which gameplay information to view, by selecting from the mask options, as well as which action to perform. Therefore, the CNN would need to return two distinct action types, yet this could lead to a combinatorial action space. To

mitigate this risk, a shared backbone was applied to the CNN which consists of multiple convolutional layers as well as an individual head, containing further convolutional layers, for each of the two action types. The two separate action type outputs returned from both heads are combined to form one single action which is returned by the CNN. Figure 3.7 displays the architecture of the developed CNN. Moreover, the RL models were trained with the proposed technique, including both the horizontal and vertical masks, for 1000 episodes with the standard DQN training process as well as by utilising curriculum learning separately. Curriculum learning was performed with the intention of improving the performance or the training speed of the agent, with the difficulty of each episode being increased throughout the training. This was implemented by initially training the model on the normal game of Pong, which consists of a fully observable environment. When the model managed to achieve the maximum possible score of the game on this environment, the occlusion masks were added. Both trained models achieved state-of-the-art results in the environment, however, when the curriculum learning method was used, the model converged earlier. From these experiments, the trained agent was observed to be concentrating on the movement of the ball as well as on their own paddle to be able to defend and hit. Therefore, the gameplay information that the agents deemed to be the most important, in order to determine which action to select, is very similar to what human players choose to observe. Moreover, the results also present the RL model's ability to not only learn the ideal actions to select but also the ideal observation to consider.

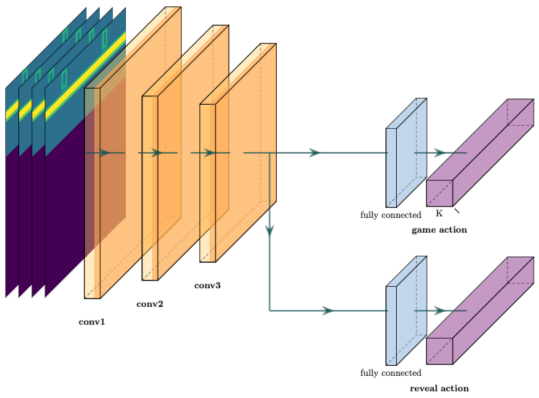


Figure 3.7 CNN Architecture [42]

The study performed by Huang et al. [43] delves into how the PPO and A2C algorithms have gained significant popularity in the field of developing AI in a variety of games including board games, card games and even intricate multi-player games.

The A2C algorithm was trained on the board game Blood Bowl in the study conducted by Justesen et al. [10]. Similarly to Jaipur, this game is a multi-agent stochastic turn-based game. However, unlike Jaipur, the environment is fully-observable and therefore, due to this element, from initial impressions, one would

assume that the environment is not that challenging for the RL process. Nonetheless, during each turn the agents can perform multiple moves and this results in the turn-wise branching factor to become excessively large for conventional algorithms to learn effectively. Moreover, the game also contains sparse and infrequent rewards which presents another layer of complexity for RL. To combat this issue, reward shaping was used, where small rewards of different values, based on the quality of the outcome obtained when an action was performed, were provided to the agents to help them understand which actions are beneficial in which states. Moreover, the Fantasy Football AI (FFAI) game engine was implemented within this work which consisted of an OpenAI Gym RL environment for the agents to be trained on. Despite the complexity of this game, the A2C algorithm managed to perform satisfactory.

Meanwhile, in the study conducted by Yu et al. [44], the PPO algorithm was implemented on four different games which all consist of cooperative multiple agents. The aim of this study was to display that the PPO algorithm can achieve notable performance when applied to multi-agent games which consist of different characteristics. In fact, the results obtained in this work exhibited that the performance achieved by PPO was equal to or superior to results obtained by off-policy algorithms in similar environments, demonstrating that this algorithm can indeed serve as a solid benchmark for cooperative MARL.

Moreover, the DQN algorithm also obtained great performance when implemented on a number of different Atari games [15], [45]. In both these studies, the algorithm was only provided the raw pixels of the video game and the score of the game as input, for the learning experience to be the same as that of a human player. Nonetheless, the algorithm managed to greatly surpass the performance of the previously trained RL models with different algorithms, on six out of the seven games that it was applied to [15]. Moreover, in three of these games, the DQN algorithm also managed to outperform professional human players. Furthermore, this algorithm was also applied to 49 distinct Atari games in the study conducted by Mnih et al. [45]. In the majority of these games, DQN obtained state-of-the-art performance, very similar to or even surpassing the scores obtained by professional human game testers and greater than the results obtained from previously applied algorithms. Consequently, the agent trained with the DQN algorithm within this work, was deemed to be the first AI agent with the ability of excelling on such a large variety of complex games.

The Temporal Difference (TD) algorithm was used to train AI agents in board games by Ghory [46], Konen [47] and Wiering et al. [48]. For such environments, this algorithm is implemented along with one or two layered NNs and the use of the backpropogation method. The combination of these elements allow for the game theoretic function to be approximated and for the agent to generalise effectively.

3.2 Reinforcement Learning Algorithms

The Jaipur game environment contains a Discrete action space, and therefore, different RL algorithms which can be applied on such a space and which have displayed state-of-the-art results on similar game environments were explored.

3.2.1 Deep Q-Network (DQN)

The DQN algorithm is a model-free, off-policy and value-based DRL algorithm as it combines DL methods with Q-learning techniques [17]. The architecture of this algorithm consists of a Q-network and target Q-network DNNs [49], [50]. The Q-network, also referred to as the online network, is updated at every time-step throughout the training, whilst the target Q-network is updated periodically and in these updates, its fixed weights are set to correspond to those of the Q-network [45], [49]. Another important element of this algorithm is the use of experience replay e, where the agent's transitions, represented as $e_t = s_t, a_t, r_t, s_{t+1}$ for each time-step t during training, are stored in a replay buffer to be sampled from during the remainder of the training process [15], [49], [50]. The transitions include the environment state observations prior to performing the action (s_t), the performed action (a_t), the received reward (r_t) and the environment observations of the resulting state after the action is applied (s_{t+1}).

With the use of experience replay and the Q-network DNN, DQN will estimate the Q-value of applying the actions on the current state, $Q_\theta(s, a)$, based on the expected sum of future rewards [17], [50]. The action with the highest Q-value is identified as the optimal action according to $Q_*(s, a) = max_\theta Q_\theta(s, a)$ [50], [51]. The discount factor γ, as well as the Q-value estimates obtained from the target Q-network, $Q_{\theta^-}(s', a')$, with the use of the parameters of the target Q-network θ^-, are used to compute the target Q-value with the target update function, presented in Equation (3.1) [45], [49], [50].

$$y_t^{DQN} = r + \gamma \max_{a'} Q_{\theta^-}(s', a') \tag{3.1}$$

The discount factor γ consists of a constant number between 0 and 1 that causes the algorithm to prioritise instant rewards, if the value is closer to 0, or long-term rewards, if the value is closer to 1 [49], [50]. Moreover, the weights of the Q-network are estimated and updated with the use of stochastic gradient descent by calculating the loss function, displayed in Equation (3.2), as the difference between the target Q-values and the Q-values [18], [45], [49].

$$L_t(\theta_t) = [(r + \gamma \underset{a'}{max} Q_{\theta_t^-}(s', a') - Q_{\theta_t}(s, a))^2] \tag{3.2}$$

The integration of the target network to update the weights of the Q-network provided a great increase in the algorithm's stability, given that the target is not updated at every time step and a fixed Q-value approximation is used instead [17], [49]. This allowed the DQN algorithm to successfully overcome the instability challenge of the DRL process, which was previously believed to be impossible to solve. Furthermore, this also proved the algorithm's ability to be applied to high-dimensional environments [17].

Moreover, this algorithm makes use of the ϵ-greedy policy to either select and perform the action with the highest Q-value, $Q_*(s, a)$, with probability $1 - \epsilon$, or select and perform a random action with probability ϵ [15], [49]. This is performed to ensure a balance between exploitation of the policy and exploration of the state space.

3.2.2 Double Deep Q-Network (DDQN)

The DDQN algorithm, which is also a model-free, off-policy and value-based DRL algorithm, was introduced from the DQN algorithm to address the DQN's disposition of overestimating action values [17].

Whilst DQN uses the target network to select and evaluate an action, DDQN separates this process of action selection and action evaluation [50]. The DDQN algorithm makes use of the Q-network to select the ideal action based on the highest Q-value as $a^* = \underset{a'}{argmax} Q_\theta(s', a')$. This algorithm will then make use of the target network to evaluate the Q-values of the selected action a^* as $Q_{\theta^-}(s', a^*)$. Furthermore, to reflect the separation of the action selection and evaluation process, the DDQN algorithm makes use of the target update function, presented in Equation (3.3), to update the Q-values.

$$y_t^{DDQN} = r + \gamma Q_{\theta^-}(s', a^*) \tag{3.3}$$

Moreover, the Dueling DDQN algorithm was developed from the DDQN algorithm with the intention of standardising action values by learning to identify the ideal states without calculating all the state action values [49].

3.2.3 Actor-Critic Algorithms

The Actor-Critic architecture was developed as a technique for TD learning to be used within the trial-and-error training process to address the limitations brought about by implementing either policy-based or value-based RL techniques on their own [17],

[52]. This technique explicitly defines the policy as independent of the value function by making use of separate memory structures. As shown in Figure 3.8, this method consists of an actor and critic components.

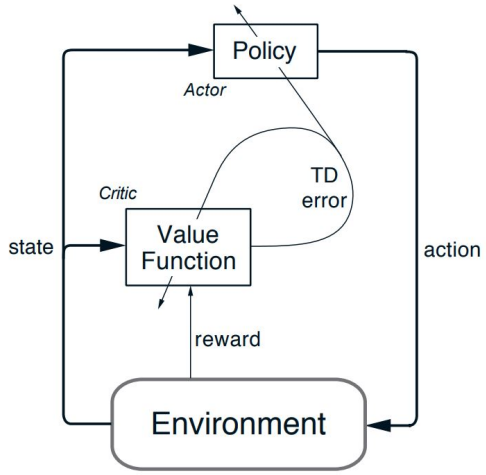


Figure 3.8 Actor-Critic Architecture [52]

The actor component uses a policy-based technique to manage the policy, perform exploration and exploitation of the possible actions and determine which action should be applied given a particular state of the environment. Meanwhile, the critic component, which makes use of a value-based method, estimates the state-value or state-action value function to evaluate the actor's decisions.

The state-value function $V(s)$ estimates the expected return of being in a specific state, whilst the state-action value function $Q(s, a)$ determines how favourable it is to apply a particular action to the given state. This feedback, which ultimately provides an estimate of how beneficial the policy is, is known as the TD error and is calculated as shown in Equation (3.4).

$$\delta_t = r + \gamma V(s') - V(s) \tag{3.4}$$

The TD error is estimated by considering the reward obtained after applying the action in state s, (r), the state-value function of state s, $(V(s))$, as well as the state-value function of the state resulting from applying the action $(V(s'))$ whilst taking into account the discount factor γ [4]. The critic presents this feedback to the actor for the policy to be improved and the expected cumulative reward to be maximised.

A key advantage of using the actor-critic technique is that the RL agents are provided with the ability of not only being able to learn from the immediate rewards but also from the long-term rewards. Moreover, actor-critic algorithms bring about the advantages of requiring less computational resources and being able to learn stochastic policies, which is very useful for competitive and non-Markovian environments.

The following sections will provide an in-depth explanation of the PPO and A2C algorithms, as well as a brief discussion on the Trust Region Policy Optimization (TRPO) and Asynchronous Advantage Actor Critic (A3C) algorithms, all of which are actor-critic techniques. However, given that all four algorithms employ the advantage function in their architecture, an explanation of this concept will be provided first.

3.2.4 Advantage Function

The advantage function is used within a number of RL algorithms, such as actor-critic methods and the Dueling DDQN algorithm, to determine the benefit of applying a specific action in comparison to applying the average action in a particular state [49], [53]. The functionality of the advantage function is to improve the stability and efficiency during the RL process. This function focuses the policy updates towards the actions that provide greater cumulative rewards and reduces the high variance present in the policy networks. It will estimate if applying action a on state s would provide a higher return than the average expected return of applying any action on state s by calculating the difference between the state-value $(V(s))$ of being in state s from the state-action value $(Q(s,a))$, also referred to as the Q-value, of applying action a on state s, as can be seen in Equation (3.5) [17].

$$A(s,a) = Q(s,a) - V(s) \tag{3.5}$$

Otherwise, instead of having to calculate both the state-action value and the state-value, Equation (3.4), used to calculate the TD error and which is explained in Section 3.2.3, can also be considered to obtain the advantage value where only the state-value function needs to be calculated [4].

3.2.5 Proximal Policy Optimization (PPO)

The PPO algorithm was developed as a policy optimisation actor-critic algorithm in 2017 as an improvement of the TRPO algorithm [17], [53].

The TRPO algorithm was designed to make use of a hard constraint on policy updates to ensure stability within the RL process, whilst still maximising the policy improvement. However, this algorithm relies on second-order optimisation techniques which make it both computationally expensive and complex to implement. Moreover, the TRPO algorithm is also not compatible with architectures that either contain noise or share parameters, between the policy and value functions or auxiliary tasks.

Therefore, the PPO algorithm was designed to maintain the benefits of the TRPO algorithm whilst only making use of first-order optimisation, consisting of a simpler implementation process as well as being more general and demonstrating

enhanced sample complexity. This algorithm utilises a proximal constraint to ensure stability in policy updates by limiting the change that can be performed to the policy between updates. Moreover, it also ensures effective policy optimisation by alternating between exploring the policy and executing a number of optimisation epochs on the sampled data. The PPO algorithm was developed with two primary variants; the PPO Penalty as well as the PPO Clip, with the latter being the most commonly used as it tends to perform the best and is also easier to implement.

The PPO Penalty controls how much the policy deviates during updates in a similar manner to the TRPO algorithm but with a simpler implementation. Unlike the hard constraint used in TRPO, the PPO Penalty variant penalises large deviations by utilising a KL-divergence penalty condition to the objective function. The penalty condition is modified accordingly during training by increasing if the KL-divergence is too large, to limit the deviation in policy during updates, or decreasing if the KL-divergence is small, for further exploration of the policy to be performed.

Meanwhile, the PPO Clip variant eliminates the KL-divergence condition and instead limits the changes performed to the policy during updates by making use of a probability ratio and applying a clipping function to the objective function. The probability ratio is calculated as $r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}$ where $\pi_\theta(a|s)$ represents the probability of selecting action a under the updated policy and $\pi_{\theta_{old}}(a|s)$ displays the probability of selecting action a under the old policy. This would allow for a ratio of how much the policy changed following the update to be calculated where $r_{\theta_{old}} = 1$. Therefore, the closer $r(\theta)$ is to a value of 1, the similar the policy remained. The probability ratio along with the clipping range of $[1 - \epsilon, 1 + \epsilon]$ and the advantage value, as explained in Section 3.2.4, are then used within the objective function presented in Equation (3.6).

$$L^{CLIP}(\theta) = \hat{E}[min(r(\theta)\hat{A}, clip(r(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A})] \tag{3.6}$$

This objective function (3.6) maximises the TRPO surrogate objective $r(\theta)\hat{A}$. However, this results in an extremely large policy update and therefore, the surrogate function is modified by clipping the probability ratio with $clip(r(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A})$. The clipping range is determined by ϵ and is used to restrict the amount of change performed to the policy by ignoring the advantages obtained outside of the clipping range. Moreover, whether the $1 - \epsilon$ or $1 + \epsilon$ clipping boundary is used depends on if the advantage value is positive or negative. Finally, this objective function will take the empirical average over a number of samples, represented by expectation $\hat{E}$, of the minimum values from the calculated clipped and unclipped values. Therefore, given that the lower bound of the unclipped objective is taken, the change in probability ratio is only not considered when it would cause the objective to improve, to prevent any large policy changes, that could decrease the policy's stability or performance, from being performed.

3.2.6 Advantage Actor Critic (A2C)

The A2C algorithm was introduced as a synchronous actor-critic algorithm from the A3C asynchronous actor-critic variant [54].

The A3C algorithm was developed in 2016 by DeepMind to allow multiple agents to be trained simultaneously in parallel on different environment instances [17], [55], [56]. As displayed in Figure 3.9, this algorithm makes use of a global network, which contains the policy and value function that are shared between all agents. During training, each agent will interact with their own environment instance to obtain experience. The optimisation of the DNN controllers will be performed where the global network is updated, with the experiences of the agents using an asynchronous gradient descent. The agent experiences are used to update the network independently of the other agents and at potentially different times. Following an agent's update, the parameters of the agent will be overwritten to correspond to the DNN's parameters for the agent's training to be continued.

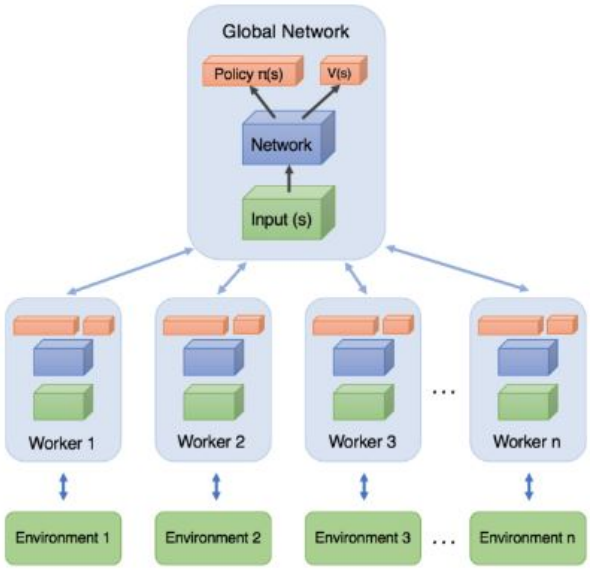


Figure 3.9 The Architecture of the A3C Algorithm [17]

The A3C algorithm proved to be very successful and achieved great results in various environments characterised by discrete or continuous action spaces as well as on 3D visual-input environments [17], [55], [56]. However, due to the asynchrounous and independent updating strategy, during training the agents end up using outdated DNN parameters until they perform an update. Moreover, the update performed would be based on old network parameters, which presents an element of instability during training and prevents the algorithm from converging efficiently [55]. Therefore, the A2C algorithm was proposed to overcome this limitation where the multiple agents are updated synchronously instead of asynchronously. For this to be performed, the

parameters of the DNN are only updated after all agents finish a batch of their independent training iterations. Following the update performed to the global network, the parameters of each agent are overwritten simultaneously.

This algorithm follows the actor-critic architecture, explained in Section 3.2.3 where a combination of policy-based and value-based methods are used for the DNN controllers to be optimised [17], [55], [56]. It makes use of an actor to handle the action-selection and a critic to calculate the state-value function and improve the policy. Moreover, it also makes use of the advantage function, explained in Section 3.2.4, to stabilise the policy updates. Furthermore, a policy gradient is used to update the actor as this algorithm is a first-order gradient technique [54].

3.3 Summary

This chapter summarisies various works which have been carried out in recent years related to the field of RL, mainly focusing on environments which are multi-agent, partially observable, stochastic and contain a large number of discrete actions. The presented summary describes the different techniques and algorithms proposed in these studies, where great results were achieved, with some game agents even outperforming professional human players. Moreover, the DQN, DDQN, PPO and A2C algorithms were described in detail along with related concepts fundamental to the explanation of such algorithms.

4 Methodology

This chapter delves into the details of the design and implementation, including all the techniques and tools used in this study, which was carried out using the Python programming language. Figure 4.1 displays the base components of the implemented solutions, and how they interact with each other to function.

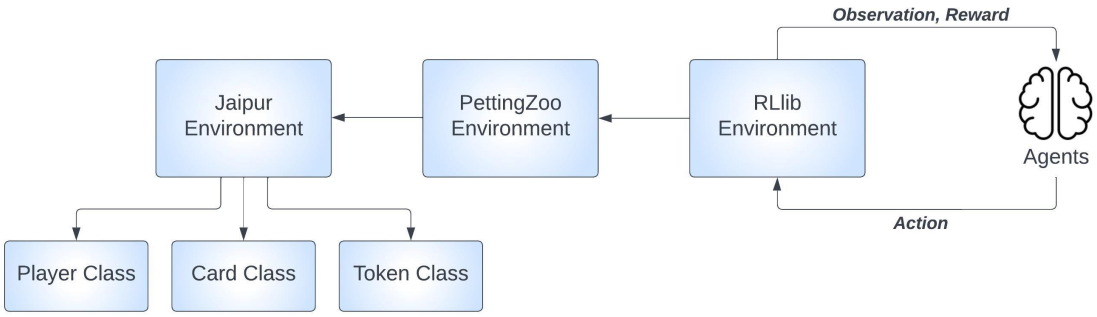


Figure 4.1 Main Components of the Implemented Solution

4.1 Initial Implementation with Action Masking

The initial part of this work consisted of implementing a functional code-based version of the game Jaipur, setting up a RL environment, integrating a RL library, in order to be able to apply and explore the performance of different RL algorithms, and applying the action masking technique. This part of the work was published in the AIxIA 2023 Conference (22nd International Conference of the Italian Association for Artificial Intelligence)[1] in the paper titled "Mastering the Card Game of Jaipur Through Zero-Knowledge Self-Play Reinforcement Learning and Action Masks" [1].

4.1.1 Implementing the Jaipur Game Environment

Within this initial implementation, the four main Jaipur actions, specified in Section 2.1, were encoded as a total of 25,469 possible actions. These actions consist of 6 actions for taking 1 card of each Goods card type from the marketplace, 25,456 actions for all the trading combinations, 1 action for taking all the Camel cards from the marketplace and 6 actions, one for each Goods card type, for selling all the cards of the same type present in the player's hand. Moreover, the architecture of the entire implementation of this game is visible in the UML diagram in Figure 4.2. This diagram displays the Player, Token, Card and Jaipur classes, along with all the variables and

[1] https://www.aixia2023.cnr.it/

functions used within these classes, all of which were implemented for a working Jaipur game to be developed.

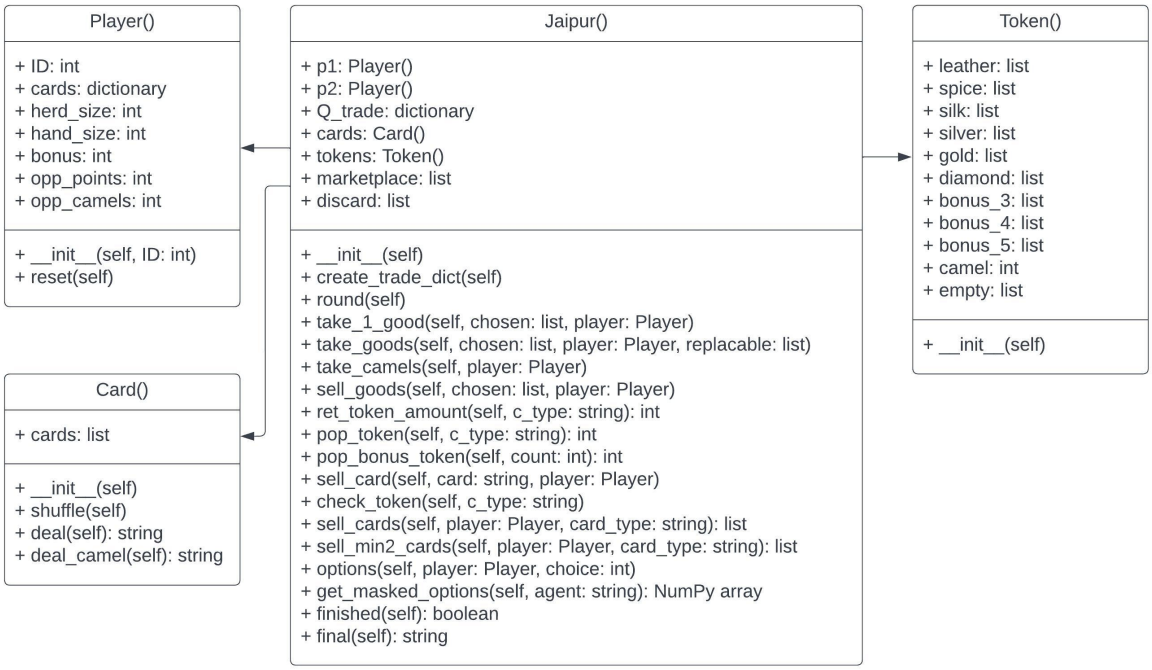


Figure 4.2 Jaipur Game Environment Architecture

The Player class is used to create the player objects and store each player's information. This class will first create the player's ID and then it will call the `reset()` function to initialise all the other variables necessary to keep track of the gameplay information of the particular player. Such information includes the player's cards, hand and herd sizes as well as points. The opponent's herd size is also included as the Camel cards can be visible to both players and a player can easily keep track of this value during gameplay. Furthermore, the Card class was implemented to initialise all the cards of the game, and store them in a variable of type `list`. This class also contains three other functions which are used during gameplay to either deal a random card, deal a Camel card or shuffle the cards. Moreover, the Token class is used to create and sort or shuffle the game tokens, according to Jaipur's rules, during the game's set-up.

Finally, the Jaipur class, which was implemented with a number of functions necessary to carry out the gameplay, will initially call the Player class to initialise the two player objects, as well as call the `create_trade_dict()` function. This function handles the creation of a dictionary which contains all the possible 25,456 trading combinations that can be performed in the game Jaipur. This dictionary takes into account the remaining 13 gameplay actions to allow for easy action mapping during policy training. Some of the more distinct functions which are found in the Jaipur class and which handle the unique mechanisms of the game Jaipur, apart from the

`create_trade_dict()` function, include the `round()`, `options()`, `take_1_good()`, `take_goods()`, `take_camels()`, `sell_goods()` and `get_masked_options()` functions.

The `round()` function is used to simulate the start of a new Jaipur round. Hence, it calls the Card and Token classes to initialise the cards and tokens as well as resets the two player objects. This function will also set-up the game environment according to the rules of the game. It will shuffle the deck of cards, assign 3 Camel cards and 2 random cards to the marketplace and deal 5 random cards to each player. It will also initialise an empty discard pile, as a `list`, where the cards will be stored after being sold. When this is all accomplished, the player turns can begin.

The players will take turns selecting an action to perform and with the use of the `options()` function, based on the action chosen, the respective function is called with the relevant information to execute that action. All the functions are equipped with various checks and error-handling conditions to ensure that all the changes performed in the environment are in line with the rules of Jaipur and to verify that all the gameplay information is updated accordingly. The `take_1_good()` function will first ensure that the player does not already have 7 cards in their hand. It will then transfer the selected card from the marketplace to the player's hand and deal another card to the marketplace as a replacement. The `take_goods()` function will first ensure that the number of cards that the player selected to take from the marketplace is the same as the number of cards that will be replaced from the player's hand. Moreover, this function will also check that the player is not trying to exchange cards of the same type and that after the exchange happens, the player will not have more than 7 cards in their hand. Should all these checks be successful, the function will trade the sets of cards accordingly. The `take_camels()` function assigns all the Camel cards found in the marketplace to the player's herd and re-populates the marketplace with cards from the deck. Meanwhile, the `sell_goods()` function will check that all the cards being sold are of the same type, that the player truly has all the cards in their hand and that the player is selling at least 2 cards if the cards are of type "Silver", "Gold" or "Diamond". This function will then distribute the respective tokens to the player, including any Bonus tokens, in exchange for the cards which will be allocated to the discard pile.

An important addition to the Jaipur class is the `get_masked_options()` function. This function is used at the start of each player's turn to determine which actions can be performed based on the game state, particularly the player's and the marketplace's cards. This function creates a NumPy `array` action mask of size 25,469 to represent all the possible actions. Each element in this array is either set to a value of 1 if the action is valid, or a value of 0 if the action is invalid.

Moreover, after each player's turn, the `finished()` function is called to check if either the deck is empty and the marketplace has less than 5 cards in it, or if 3 sets of Goods type tokens finish. Should either of these conditions be true, the game is

terminated and the `final()` function is called to delegate the Camel token to the player who has the most Camel cards and determine the winner of the game based on which player has the highest number of points. Moreover, as mentioned in Section 2.1, according to the rules of Jaipur, a game can be played as 1 to 3 rounds, however, for the training purposes of this work, each game is set to be 1 round.

Many rule based fault detection conditions were implemented within the Jaipur environment to ensure that Jaipur's gameplay is correctly replicated and that all actions performed comply with the game's rules. Section 5.1.1 delves into more detail on these implemented conditions, as well as on the multiple diligent checks that were carried out. Some test cases that were applied during these checks are displayed in Appendix B.

4.1.2 Implementing the Reinforcement Learning Environment

The multi-agent RL library PettingZoo[2] was used in this work for the RL environment to be created. This library is built on the Gymnasium framework[3] which was formerly known as OpenAI Gym[4]. The PettingZoo and Gymnasium APIs are both modern toolkits designed to facilitate the implementation of RL environments as they provide the means for testing various algorithms.

For the multi-agent RL to be performed in this work, PettingZoo's AEC API[5] was used to create a custom AEC environment. This environment includes seven functions within the environment's class. These functions are `init()`, `observation_space()`, `action_space()`, `get_observation()`, `observe()`, `reset()` and `step()`. Two crucial parts of the RL environment, which need to be established prior to implementing the remainder of the RL environment, consist of the action and observation spaces.

Designing the Action and Observation Spaces

The action and observation spaces establish and provide the agent with information regarding the environment's important features. Therefore, it is crucial for these two spaces to be developed correctly for the agent to be able to comprehend both the environment, as well as the changes made when an action is performed. Ultimately, it is this information that allows the agent to learn which actions should be taken to accomplish its goals. Since the PettingZoo API was used, these two spaces were both implemented by utilising Gymnasium's spaces.

[2]https://pettingzoo.farama.org/
[3]https://gymnasium.farama.org/
[4]https://www.gymlibrary.dev/
[5]https://pettingzoo.farama.org/api/aec/

The action space must convey all the possible actions that can be performed within the environment and the information stored in this space will remain unchanged throughout the gameplay. Therefore, the action space for both agents was implemented as a `Discrete` space of size 25,469, to reflect all of Jaipur's possible actions as integers from 0 to 25,468.

Meanwhile, the observation space of the environment must encode all the relevant information of each game state that influences the agent's action selection process. The information within this space will be updated for each player, prior to the respective player's turn, in order to reflect the game's current state from the player's perspective and allow the agent to select an appropriate action. With respect to the Jaipur game environment, the important characteristics which affect the decision-making of a player, and hence were included in the observation space, are the player's own Camel and Goods cards as well as their score, the cards present in the marketplace, the remaining tokens as well as how many Camel cards the opponent has. Moreover, an action mask was also added to the observation space due to the very large number of possible actions. This was a very crucial addition to this work as it allows for the agents to learn in a more efficient way and therefore, helps the algorithms converge much quicker. In order to properly contain all these important characteristics within the observation space, the Gymnasium's `Dict` space was utilised to create the observation space as a dictionary of two key-value pairs, with each value representing a separate `Box` space, which is a continuos n-dimensional space. As can be seen in the code snippet in Listing 4.1, the observation space contained two `Box` spaces titled 'observation' and 'action_mask' respectively.

```
1 self.observation_spaces = {agent: spaces.Dict({
2     'observation': spaces.Box(low=np.array
    ([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]), high=np.array
    ([7,7,7,7,7,5,5,5,5,5,5,5,11,221,11,5,5,5,7,7,9,7,6,5]), dtype=np.
    int16),
3     'action_mask': spaces.Box(low=0, high=1, shape=(25469,), dtype=np.int8)
    ,
4     }) for agent in self.agents}
```

Listing 4.1 Observation Spaces

The 'observation' component of the environment's observation space was originally created as a `Dict` space instead of a `Box` space, for the gameplay information to be stored in a neater way. As can be seen in the code snippet in Listing 4.2, each gameplay characteristic was encoded in its own separate space within the `Dict` space. A `Box` space was used to represent the player's Goods cards. Since there are 6 types of Goods cards and each player can have up to 7 of such cards in their hand, the `Box` space contained 6 elements with each element having the option of being a value between 0

and 7 inclusive. The space to encode the marketplace information was created in a similar manner. The marketplace can have up to 5 cards in total at once, with 7 different types of cards that can be found in the marketplace, the 6 different types of Goods cards and Camel cards. Therefore, the space representing this information was created as a `Box` space of 7 elements with each element having the possibility of having a value between 0 and 5, both values included. A `Discrete` space was used for each of the remaining three gameplay characteristics; the player's score, the player's Camel cards and the opponent's Camel cards. For the player's score, the `Discrete` space was set to a size of 222, representing the score as an integer between 0 and 221, since it can never exceed a value of 221 in a game. The spaces for the player's and the opponent's Camel cards were both created to be of size 12, having a value between 0 and 11, since the game Jaipur consists of 11 Camel cards in total.

```
self.observation_spaces = {agent: spaces.Dict({
    'observation': spaces.Dict({
        'hand': spaces.Box(low=0, high=7, shape=(6,), dtype=np.int8), #6 card types with values between 0 and 7
        'marketplace': spaces.Box(low=0, high=5, shape=(7,), dtype=np.int8), #7 card types with values between 0 and 5
        'herd': spaces.Discrete(12), #0 to 11 camels
        'score': spaces.Discrete(222), #0 to 221 points
        'opp_herd': spaces.Discrete(12), #0 to 11 camels
        }),
    'action_mask': spaces.Box(low=0, high=1, shape=(25469,), dtype=np.int8),
    }) for agent in self.agents}
```

Listing 4.2 Initial Observation Spaces

As a result of compatibility issues, with RLlib's action masking models, the 'observation' space displayed in the code snippet in Listing 4.2 had to be converted to the single `Box` space shown in Listing 4.1, which encompassed all the relevant information with the same above-mentioned values. Prior to resorting to the single `Box` space, in an effort to adapt the initial observation space to be compatible with the action masking models, multiple attempts were made to flatten the `Dict` space, however, all the solutions were either no longer uniform or became obsolete, making the endeavour futile.

Furthermore, whilst training the algorithms, information about how many Goods and Bonus tokens remained was incorporated into the 'observation' element within the environment's observation space. As can be seen in the code snippet in Listing 4.1, the values for these tokens ranged based on the number of tokens for each type. Therefore, for the Diamond, Gold and Silver types, as well as for the Bonus tokens of when the player sells 5 cards at once, the range was set from 0 to 5. For the selling 4

Action Space	$A_S = \{0,1\}^{25{,}469}$
Observation Space	$O_S = \{A_M, O\}$
Action Mask	$A_M = \{0,1\}^{25{,}469}$
Observation	$O = \{G, M, C_P, P, C_O, T\}$
Player's Goods Cards	$G = \{0,7\}^6$
Marketplace Cards	$M = \{0,5\}^7$
Player's Camel Cards	$C_P = \{0,11\}$
Player's Points	$P = \{0,221\}$
Opponent Camel Cards	$C_O = \{0,11\}$
Remaining Goods and Bonus Tokens	$T = \{\{0,5\}^4, \{0,6\}, \{0,7\}^3, \{0,9\}\}$

Table 4.1 Initial Implementation Action and Observation Spaces

cards at once Bonus tokens, the range was set from 0 to 6. For the Silk and Spice types and Bonus tokens of when a player sells 3 cards at once, the range was set from 0 to 7. Finally, for the Leather tokens, the range was set from 0 to 9. The addition of this information provided the agent with more visibility on which tokens it could benefit from, allowing it to potentially learn different and more complex strategies, such as learning to purposefully delay its rewards to receive a larger number of points by selling multiple cards at once to receive the Bonus tokens.

Meanwhile, a `Box` space of size 25,469 was used for the 'action_mask' element within the observation space, representing each of the possible actions with a value of 0, if the action is unsound, or with a value of 1, if the action can be performed. Table 4.1 displays the final action and observation spaces used within this implementation.

Developing the Reinforcement Learning Framework

After determining the appropriate action and observation spaces, and ensuring that they correctly encapsulate all the necessary information and function properly with the PettingZoo mechanisms, the RL environment was developed, as displayed in Figure 4.3.

Upon initialising the RL environment, the Jaipur class is called for the initial setup of the game. Moreover, the agents and the other necessary variables are initialised within the RL environment. For each agent, the action and observation space was created, as specified in Section 4.1.2, along with the reward, information, termination and truncation variables. All these variables are necessary for the agents to observe the environment and learn during the training. Furthermore, the environment also consists of the `agent_selection` variable which makes use of PettingZoo's `agent_selector()` function to iterate through the agents within the RL environment, in the order that they were created.

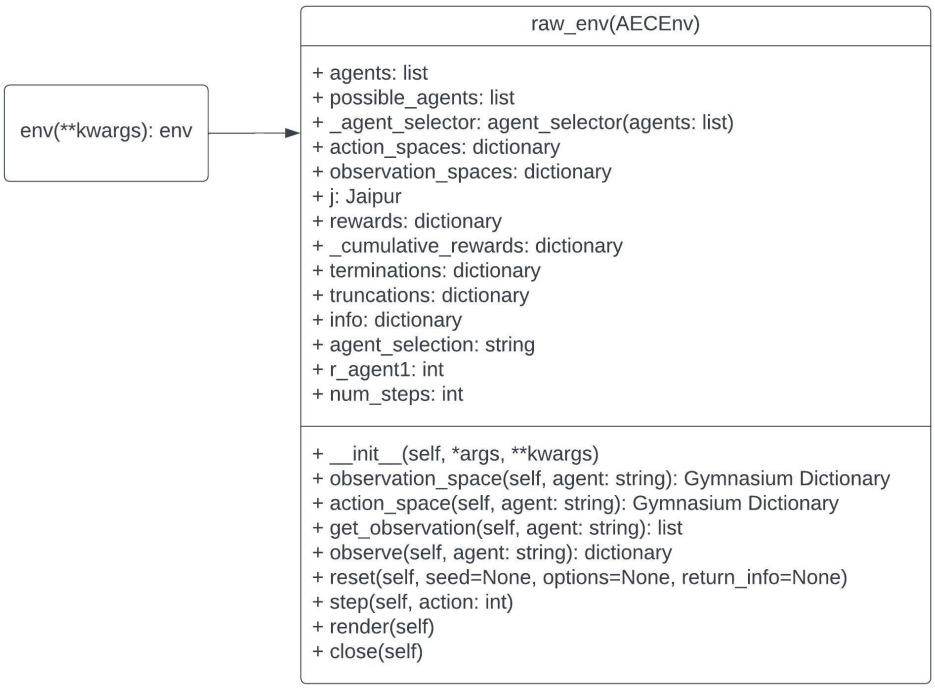


Figure 4.3 Custom PettingZoo Environment Architecture.

In compliance with PettingZoo's requirements, the RL environment consists of a `step()` function which is used in every player's turn. This function is used to gather the current gameplay information, pass it to the respective policy, based on which agent is playing, and then perform the action selected by the agent. Initially, the function will check the termination and truncation variables of the current agent, to ensure that the agent can still perform an action within the environment. Should the agent have been terminated or truncated, the function will provide a `None` action, remove the agent from within the environment and iterate to the next agent until no active agents remain. Otherwise, the action selected by the agent will be performed on the game environment by calling Jaipur's `options()` function. After the action is performed, the `finished()` function will be called from the Jaipur class, to inspect whether or not the game terminated. If this is the case, the `final()` function is called to first provide the player with the most number of Camel cards with the Camel token points and then, establish the player with the highest number of points as the winner. Moreover, when the game ends, the termination variable of both agents will be set to `True` to terminate both players immediately and ensure that they do not perform another action. Irrespective of whether the game is terminated or not, each agent's rewards are calculated after the respective player's turn, based on the difference in points from the player's previous turn to the current turn. However, the PettingZoo architecture updates both policies with the rewards following the last agent's turn. Therefore, the variable `r_agent1` is used in this environment to keep track of the first player's reward

for each round. Moreover, should the game terminate during the first player's turn, with this architecture, the rewards will not be updated accordingly. Hence, a modification was made for the policies to be updated within the specific turn of when the game is finished. In this case, should the second player obtain the Camel token points, these points are returned to the respective policy. Meanwhile, should the game terminate during the second player's turn and the first player obtains the Camel token points, the points are added to the `r_agent1` variable for the points to be recorded within the policy. Moreover, a counter was implemented within the `step()` function to ensure that the players do not perform more than 1000 rounds together in one game, otherwise, the agents will be set as truncated for that particular game to be terminated. Following all the above-mentioned checks and changes performed to the game and RL environments, the `step()` function will modify the `agent_selection` variable to point to the next agent, for the next player's turn to begin.

The `action_space()` and `observation_space()` functions, which are necessary within the PettingZoo environment, are used to return the respective agent's action and observation space respectively. The environment also consists of an `observe()` function which calls the RL environment's `get_observation()` function, as well as Jaipur's `get_masked_options()` function, to gather and update the current agent's observation space. The `get_observation()` function obtains the latest gameplay information of the current agent, including the player's cards, number of points, marketplace cards, opponent's number of Camel cards and the number of remaining tokens. This function will then structure the obtained information into the correct format and return this data to be stored in the 'observation' element within the observation space. Similarly, the `get_masked_options()` function is used to construct the action mask of the agent, based on the gameplay state, and return it to be stored in the 'action_mask' element of the observation space. Meanwhile, the `reset()` function will re-initialise the variables of the RL environment, as well as reset the game environment by calling the Jaipur environment's `round()` function.

Furthermore, a function called `env()` was implemented to call the PettingZoo wrappers, `AssertOutOfBoundsWrapper` and `OrderEnforcingWrapper`, on the custom PettingZoo environment. The `AssertOutOfBoundsWrapper` provides error handling on an environment with Discrete action spaces, to ensure that the agents' action selection remains within the set limits of the action space. Meanwhile, the `OrderEnforcingWrapper` is beneficial for ensuring that the environment functions correctly as it performs error handling of function calls and attribute access.

4.1.3 Implementing the Reinforcement Learning Algorithms

Following the implementation of the RL environment, a RL library was beneficial for this work in order to be able to experiment with different RL algorithms and evaluate their performance on the environment. Originally, the intention was to make use of the StableBaselines3[6] library however, this gave rise to certain incompatibility issues which did not have a work around. The StableBaselines3 API was previously compatible with the PettingZoo API however, this was no longer the case following some updates performed to the PettingZoo API. The PettingZoo library had switched from utilising the Gym library to the Gymnasium library, whilst the StableBaselines3 library still relied on the Gym library. Upon trying to make use of the Gym library within the PettingZoo environment, it was noted that the function designated to convert the PettingZoo environment into an environment compatible with the StableBaselines3 library was not compatible with PettingZoo's AEC environment. Moreover, upon further research, it was observed that only one algorithm offered by StableBaselines3 could make use of action masking whilst training, which was the 'Maskable PPO'. Considering the game's large action space, it would be very computationally and temporally inefficient to not make use of action masking during training as the algorithms would require a significant amount of time to converge.

Hence, further research [23], [28], [57] was performed to explore different options which resulted in the RLlib[7] library, which is offered by the Ray[8] framework, being selected as it proved to be ideal for this work. RLlib was updated to be used with the Gymnasium and PettingZoo APIs, including PettingZoo's AEC environment and it also offers various different useful models and functions, including action masking models, which can all be implemented within their ready-made algorithms.

Provided that the Jaipur RL environment has a `Discrete` action space, research was performed on which algorithms, that are compatible with such spaces, would be ideal to implement within this environment. The PPO, A2C, DQN and DQN algorithms were selected for this work as, apart from being compatible with `Discrete` action spaces, these algorithms have achieved great results when applied to similar environments. Preference was given to these algorithms, rather than those typically selected for POMDPs which make use of Recurrent Neural Network (RNN)s, as within this work, the game was not modelled as a POMDP. Given that POMDPs make use of belief states and the game has a very large action space, if it were to be modelled as a POMDP, it would result in a combinatorial explosion.

To implement the RLlib's RL algorithms within the Jaipur PettingZoo environment, the environment had to initially be configured into a `MultiAgentEnv`, which

[6] https://stable-baselines3.readthedocs.io/en/master/
[7] https://docs.ray.io/en/latest/rllib/index.html
[8] https://docs.ray.io/en/latest/index.html

is RLlib's multi-agent environment, as well as registered as an RLlib environment. The environment's configuration was accomplished with the use of RLlib's PettingZoo Interface's `PettingZooEnv()` function, whilst the environment's registration was performed using Ray's `register_env()` function.

The RLlib's action masking model `ActionMaskModel` was implemented for the algorithms to be able to utilise the RL environment's action mask and filter out the invalid actions during training. Moreover, since Jaipur is a competitive two-player game, two separate policies needed to be trained for each algorithm to prevent the agents from potentially learning to help each other and ensure that they are truly always playing against each other. Therefore, to proceed with applying the algorithms on the environment, a trainer was initialised, with the TensorFlow[9] framework, and configured to make use of the action masking model and to train two separate policies.

Each algorithm was trained with the following hyperparameters. The PPO, DQN and DDQN algorithms were configured with 1 local worker and 0 rollout workers, whilst the A2C algorithm was configured with 2 rollout workers, given that this algorithm makes use of multiple workers. Moreover, the PPO's KL divergence coefficient was configured to a value of 0 for the clip-based PPO to be used in this work, which does not apply KL divergence. The learning rate values of the PPO and A2C algorithms were both set to 0.01. The gamma value of the PPO algorithm was set to 0.9 and the gradients clip value of the A2C algorithm was set to 30. For the DQN and DDQN algorithms, the `MultiAgentPrioritizedReplayBuffer` type of replay was used due to the presence of multiple agents within the game environment. The replay buffer's prioritised replay α was set to a value of 0.6, whilst the prioritised replay β was set to a value of 0.4. Furthermore, the replay buffer size was set to a default size of 50,000, however, since it was very large, it was consuming a lot of memory, and the algorithms were requiring extended training durations. Therefore, experimentation was performed to determine the ideal size of the replay buffer to ensure both good performance and efficiency, which ultimately resulted to a value of 1,000. Moreover, the batch mode of DQN and DDQN was set to be consistent with the game round's length and hence, it was set to the *'complete_episodes'* value. The default values, as set by RLlib, were retained for the other hyperparameters of each algorithm. Appendix A presents a detailed list of each algorithm's hyperparameters, along with their corresponding values. Moreover, each algorithm was configured with its default RLlib policy network, as these architectures have demonstrated strong performance on similar problems and within this implementation.

Following the configuration, each algorithm was created with the use of RLlib's `build()` function whilst passing the registered Jaipur environment as input. The `train()` function was then used to perform the training for each algorithm. Checkpoints were

[9] https://www.tensorflow.org/

saved during the training of the algorithms with the use of the `save()` function. All the data related to the policies' training state is saved and can be restored within a checkpoint, allowing the possibility of the training process to be continued once the model is restored. The checkpoints made it easier to check the performance of the policies and select the best-performing models. Moreover, these also ensured that, in the case of a power outage or a system crash, the model currently being trained would not be entirely lost, which was very important especially due to the long training times. For a model to be restored from a particular checkpoint, the `from_checkpoint()` function, from RLlib's Algorithm class, was utilised.

The training was always performed with the use of unique games which were randomly generated. For each algorithm, in order to determine the ideal number of trained games and steps for the best-performing models to be selected as the final models, multiple models were trained and rigorously tested. Table 4.2 displays the number of steps and games trained, steps sampled and time taken for each of the final models of this part of the work. The number of trained games and steps differ across the algorithms based on when that particular algorithm presented the best results.

Table 4.2 Training Details - Initial Implementation

Algorithm	Games Trained	Steps Trained	Steps Sampled	Time Taken
PPO	38,799	2,000,000	2,000,000	44.45hrs
A2C	17,114	939,840	939,840	8.22hrs
DQN	32,134	1,027,968	1,809,912	32.21hrs
DDQN	64,261	2,056,064	3,586,768	62.72hrs

The testing was performed quantitatively on the trained models by simulating the game environment and playing the `player_1` and `player_2` policies, of each model, against each other for 1000 unique games that were not part of the training set. Given that these two policies are trained with their own distinct observations and rewards, this quantitative evaluation was conducted to analyse the performance of the two separate policies. To simulate the game environment, it is first initialised and then a set of steps are carried out continuously until the game terminates. The game state information is obtained and the respective policy, based on which player is currently playing, as well as the state's observation are provided to the `compute_single_action()` function, which is offered by RLlib. This function will return the agent's chosen action which is then applied to the environment for the next player's turn to commence. Following each set of 1000 games, the results were displayed on two different graphs, a box plot and a line graph, with the use of the Pandas[10] and Matplotlib[11] libraries. The box plot displays the mean, average and minimum values of the `player_1` and `player_2`

[10]https://pandas.pydata.org/
[11]https://matplotlib.org/

policies individually as well as per game. Meanwhile, the line graph displays the scores of both the `player_1` and `player_2` policies throughout the 1000 games. Moreover, to obtain a deeper insight on how each policy's performance was changing throughout the training, the training data was also visualised with the use of TensorFlow's TensorBoard[12] toolkit, as well as by extracting the important training data and plotting it using the above mentioned libraries. Furthermore, apart from the quantitative testing, the models were also tested by evaluating the agent's decisions. This was performed by simulating the game environment, as mentioned above, whilst displaying the agent's observations and actions. Moreover, specific game observations were also passed on to the `compute_single_action()` function to assess how the policies would perform in a particular state.

4.2 Experimenting with Game Features and Hyperparameter Tuning

The second part of this work involved experimenting with how the RL training process is affected when modifying the game environment and hyperparameters.

4.2.1 Increased Action Space

In this part of the work, the originally mentioned 6 selling actions from the 25,469 possible actions were split into more detailed actions, making up a total of 36 selling actions and leading the number of possible actions in the game environment to grow to a total of 25,499.

Previously, the agent only had the option of selecting to sell cards based on the type and all the cards present in their hand, of that particular type, would be sold in the round. Hence, the increase in actions was performed to provide the agent with more flexibility on the amount of cards, of the same type, sold from their hand in a round. Each of the 6 previously mentioned actions were changed from simply 'sell_Diamond_cards' to 'sell_2_Diamond_cards', 'sell_3_Diamond_cards' and so on, representing all the possible number of cards that the player can sell individually for each Goods card type as a separate action. Therefore, the possible actions for selling Diamond, Gold or Silver cards ranged from selling between 2 to 6 cards, as the game only consists of 6 cards of these types. Meanwhile, for the other types of Goods cards, the possible actions ranged from selling between 1 and 7 cards, as even though the game contains more than 7 cards of these types, the player can never have more than 7 cards in their hand. This change resulted in more realistic gameplay and allowed the

[12] https://www.tensorflow.org/tensorboard

agent to potentially discover certain tricks such as only selling one low-value card in a turn to avoid taking another low-value card from the marketplace. By doing so, the agent would prevent their opponent from potentially having the opportunity to obtain a high-value card when the low-value card taken from the marketplace would be replaced with a random card from the deck. Another potentially good move that can be performed with the added actions would be for the agent to avoid taking the last token of a particular card type so as to prolong the game and either sell other cards or take more Camel cards to earn the Camel token points.

To fully implement the additional actions within the game environment, the necessary game functions were updated accordingly. The `create_trade_dict()` function was updated to take into consideration the now 43 gameplay actions, prior to the trading actions. The `get_masked_options()` function's NumPy array action mask was increased to a size of 25,499 and additional conditional statements were added within this function to encode the new actions with a value of 1 or 0 depending on whether the action is valid or not within the turn. The `options()` function was also updated to reflect the added actions. Moreover, the `sell_cards()` and `sell_min2_cards()` functions were removed as these functions were previously used to gather the total number of cards present in the player's hand, of the specific type selected by the player. These functions used to also ensure that the player is selling a valid number of cards, such as more than 2 cards for the high-value cards. Since the number of cards to be sold are specified according to the action chosen, these functions are no longer needed, and a new function, titled `sell_num_cards()`, was implemented. This new function will first ensure that the player truly does have all the cards that they are trying to sell, and then it will return the number of cards for the selling action to be performed.

Furthermore, to reflect the increase in actions within the RL environment, the size of the action space, as well as the observation space's 'action_mask' element, were both increased to a total of 25,499. The updated action and observation spaces are displayed in Table 4.3. Given that this implementation represents all the possible gameplay options in greater detail, this version of the game environment was used for the remaining experiments.

Table 4.4 displays the number of games and steps trained, steps sampled as well as time taken for each of the algorithms when they were trained with the added actions using the hyperparameter values mentioned in Section 4.1.3.

4.2.2 Hyperparameter Tuning

With the intention of enhancing each algorithm's performance and training efficiency, hyperparameter tuning was performed on the PPO, A2C, DQN and DDQN algorithms, within the Jaipur RL environment consisting of 25,499 actions, mentioned in Section

Action Space	$A_S = \{0,1\}^{25,499}$
Observation Space	$O_S = \{A_M, O\}$
Action Mask	$A_M = \{0,1\}^{25,499}$
Observation	$O = \{G, M, C_P, P, C_O, T\}$
Player's Goods Cards	$G = \{0,7\}^6$
Marketplace Cards	$M = \{0,5\}^7$
Player's Camel Cards	$C_P = \{0,11\}$
Player's Points	$P = \{0,221\}$
Opponent Camel Cards	$C_O = \{0,11\}$
Remaining Goods and Bonus Tokens	$T = \{\{0,5\}^4, \{0,6\}, \{0,7\}^3, \{0,9\}\}$

Table 4.3 Increased Actions - Action and Observation Spaces

Table 4.4 Training Details - Increased Actions

Algorithm	Games Trained	Steps Trained	Steps Sampled	Time Taken
PPO	37,068	2,000,000	2,000,000	39.63hrs
A2C	16,508	939,776	939,776	8.26hrs
DQN	32,138	1,028,032	1,907,990	34.09hrs
DDQN	64,259	2,055,840	3,642,479	63.96hrs

4.2.1. Within this work, the PBT technique was applied with the use of Ray Tune's ready-made PopulationBasedTraining schedular to adjust the hyperparameters during training, given that this technique proved to be very effective at finding good hyperparameters in previous work [31]. As explained in Section 2.11, PBT works by training a group of models in parallel, replaces the lower performing models with copies of the best performing model and tunes the hyperparameters by either applying a mutation on them or selecting new hyperparameters. PBT's aim is to update the models to perform better than the best model within the group. Therefore, the training for each algorithm was performed in groups of 3 which was the maximum possible for the computational resources available. However, to ensure ample exploration of hyperparameter values, multiple tuning sessions were conducted for each algorithm. Moreover, the metric which the technique was trying to maximise was set as the mean episode score. This metric was chosen as it reflects the average cumulative score that the players obtain during the games and therefore, it requires that either both players perform well within the environment or one player performs a lot better than the other.

For all algorithms, the `resample_probability` value was set to the default value of 0.25, which could cause the resample distribution of some hyperparameters to potentially go out of the provided range. Meanwhile, multiple sets of training were performed, with values between 10 and 120 set as the `perturbation_interval` to obtain

the ideal value. The training sets which returned the best models were set with a value of 120 as the `perturbation_interval` for all the algorithms. Moreover, the `hyperparam_mutations` were set differently for each algorithm. Research was performed to identify which hyperparameters would be ideally tuned, for each algorithm, as well as to gather information about the usual values tested for these hyperparameters.

For the PPO algorithm, tuning was performed on the gamma, clip parameter, learning rate and KL divergence coefficient. The ranges for the resampling distribution of each of these parameters were set as follows. The gamma value was set to be a random floating number between and including 0.9 and 1, whilst the clip parameter value was set to be a random floating number between and including 0.01 and 0.5. The set of values provided to be chosen for the learning rate included 0.01, 1e-3, 5e-4, 1e-4, 5e-5 and 1e-5, whilst for the KL divergence coefficient the values 0, 0.2, 0.4, 0.6, 0.8 and 1 were provided. In total, 8 tuning sessions were conducted for this algorithm.

Meanwhile, for the A2C algorithm, tuning was performed on the learning rate where the provided values for the resampling distribution were 0.01, 1e-3, 5e-4, 1e-4, 5e-5 and 1e-5. Moreover, the lambda parameter was set to be assigned a random floating number between and including 0.9 and 1.0. The `entropy_coeff` and `vf_loss_coeff` parameters were also set to be a random floating number between and including 0.001 and 0.01, as well as 0.5 and 1 respectively. Finally, the values provided for the `grad_clip` parameter were 0.01, 0.5, 0.1, 1, 5, 10, 15 and 30. Tuning was performed on 13 separate sessions for the A2C algorithm.

Moreover, for the DQN and DDQN algorithms, tuning was performed on the parameters related to the replay buffer including the replay buffer's capacity, the prioritized replay α and the prioritized replay β. The capacity was set to be a random integer number, between 500 and 2000. Meanwhile, the prioritized replay α and the prioritized replay β values were set to be a random floating number from 0.5 to 0.8 and 0.3 to 0.6 respectively. Initially, 5 and 2 tuning sessions were performed for the DQN and DDQN algorithms respectively. However, whilst the DDQN achieved higher scores with the tuned hyperparameters, the DQN algorithm achieved similar scores to those achieved by the model trained with the previous hyperparameter values. Therefore, for the DQN algorithm, further hyperparameters were selected to be tuned and an additional 11 tuning sessions were conducted. For this algorithm's `grad_clip` the values 1, 5, 10, 20, 30, 40 and 50 were provided. Moreover, the `adam_epsilon` parameter was set as a random floating number between 1e-08 and 1e-4, whilst the `tau` parameter was set to be a random floating number between 0.001 and 1.0.

Despite numerous attempts, for DQN the scores returned by the initially used hyperparameters remained higher than those returned with the tuned hyperparameters. Therefore, experimentation was also performed by increasing the `perturbation_interval` to 150 and 200, for the model to be trained for longer with the

same hyperparameter values, to potentially reach higher scores. However, this attempt was futile and therefore, the hyperparameter values were modified and the manual tuning technique was also applied by manually setting the hyperparameters to a range similar to the initially used values. The `tau` parameter was modified to have a random floating number between 0.1 and 1.0. This was applied as, with the previously provided range, very small values were being selected by the PBT algorithm, which were closer to 0.001 whilst when the value was set to 1.0, prior to hyperparameter tuning, the model achieved higher scores. However, even with the updated range, small values closer to 0.1 were being selected. Therefore, the following values were provided for the `tau` parameter; 0.001, 0.01, 0.5, 0.8 and 1.0. Moreover, the size of the replay buffer was also decreased to be between 700 to 1500 however, the scores achieved were still not exceeding the results obtained by the initial model. Hence, since the tuning for the DDQN algorithm had managed to achieve higher scores, the values for the replay buffer's size, the prioritized replay α and the prioritized replay β were set as the ideal values identified from DDQN's tuning. Experimentation was then performed with these values and the results were improved.

Due to the very long training times of the algorithms, the stopping criteria for each algorithm was set to a number of iterations which allowed the algorithm to start converging without having to perform the total duration of the training. Hence, the PPO algorithm was stopped after 100 iterations whilst the A2C, DQN and DDQN algorithms were stopped after 500 iterations. Moreover, the updated hyperparameter values which resulted in the best-performing models are displayed in Table 4.5. The algorithms were then trained with these identified hyperparameters and the time taken, the number of games and steps trained as well as the number of steps sampled for each algorithm are displayed in Table 4.6.

Table 4.5 Updated Hyperparameter Values

Algorithm/s	Hyperparameter	Value
PPO	Clipping Parameter	0.4094
PPO	Learning Rate	5e-05
PPO	Gamma	0.9086
A2C	Learning Rate	0.0001
DQN, DDQN	Replay Buffer Size	2524
DQN, DDQN	Prioritized Replay α	1.1235
DQN, DDQN	Prioritized Replay β	0.5234
DQN	Adam Epsilon	6.74e-06

Table 4.6 Training Details - Hyperparameter Tuning

Algorithm	Games Trained	Steps Trained	Steps Sampled	Time Taken
PPO	46,768	2,000,000	2,000,000	44.46hrs
A2C	16,851	807,232	807,232	8.37hrs
DQN	31,361	1,003,136	1,814,355	32.15hrs
DDQN	57,784	1,848,704	3,281,354	58.46hrs

4.2.3 Increased Agent Observation

Another experiment conducted on this work was to analyse how the performance of the RL algorithms differed when further gameplay information was provided to the agent. This provided an interesting evaluation of whether the agents would indeed benefit from additional information or if they were able to maximise their efficiency through RL with the partial information of the game. Moreover, since the original observation space was already quite large, it was even more interesting to see how the agents would react to the increase of such space.

Within this implementation, the agent would be able to keep track of all the cards that the opponent takes from the marketplace, as well as an estimate of the opponent's score during gameplay. These would potentially allow the agent to perform smarter moves such as withholding certain card types to prevent the opponent from obtaining the Bonus token points of selling 3, 4 or 5 cards of that type in the same round. The player will not be aware of the cards allocated to the opponent at the beginning of the game, since these cards are assigned face down directly from the deck. Moreover, when the opponent sells cards, the agent is able to see the values of the tokens taken, except for when the opponent obtains a Bonus token when selling more than 3 cards at once. Since the Bonus tokens are shuffled and assigned randomly, in this implementation, the agent will consider the largest possible value of the respective Bonus token taken by the opponent. This would allow the agent to aim higher to win by overestimating the opponent's score instead of underestimating it. Therefore, the opponent's score estimate was increased by 3, 6 and 10 points when the opponent sells 3, 4, or 5 or more cards at once respectively.

Both the game and RL environments were modified for the necessary mechanisms to be implemented which allow the agent to be aware of the cards taken and sold by the opponent, as well as to estimate the opponent's score based on the actions performed during gameplay. From the game environment, the Player class was updated to include two new variables; `opp_cards` and `opp_points`. The `opp_cards` was created as a dictionary to keep track of the cards that the opponent takes during gameplay. This dictionary is also updated accordingly when a player sells or trades cards, where the value of each card type can never be less than 0. Meanwhile, the

Action Space	$A_S = \{0,1\}^{25,499}$
Observation Space	$O_S = \{A_M, O\}$
Action Mask	$A_M = \{0,1\}^{25,499}$
Observation	$O = \{G, M, C_P, P, C_O, T\}$
Player's Goods Cards	$G = \{0,7\}^6$
Marketplace Cards	$M = \{0,5\}^7$
Player's Camel Cards	$C_P = \{0,11\}$
Player's Points	$P = \{0,221\}$
Opponent's Goods Cards	$G_O = \{0,7\}^6$
Opponent's Camel Cards	$C_O = \{0,11\}$
Opponent's Points	$P = \{0,221\}$
Remaining Goods and Bonus Tokens	$T = \{\{0,5\}^4, \{0,6\}, \{0,7\}^3, \{0,9\}\}$

Table 4.7 Increasing the Agent's Observation - Action and Observation Spaces

`opp_points` is used to store an integer value of the opponent's estimated score. Moreover, all the functions implemented within the game environment which are used to perform an action were updated accordingly. This includes the `take_1_good()`, `take_goods()`, `take_camels()`, `sell_goods()` and `sell_card()` functions. When a player performs an action, the changes made to the player's information which are visible to their opponent will be updated, by accessing and modifying the opponent's `opp_cards` and `opp_points` variables, during the player's turn. Therefore, the `options()` function was also updated to obtain and pass on the opponent object to the respective functions based on the action chosen to be performed, for the player's information to be updated accordingly from within the opponent object. Furthermore, the `final()` function was also modified to update the opponent's score with the Camel token points, from within the player object, should the opponent have the most number of Camel cards at the end of the game.

The action space of this implementation remained as a `Discrete` space of 25,499 possible actions and therefore, the 'action_mask' element within the observation space was not modified. However, from the RL environment, the 'observation' element of the observation space was modified to include the amount of each Goods card type that the opponent has in their hand, as well as the opponent's estimated score. As shown in Table 4.7, the values for these elements were set as the same values as the player's respective elements, specified in Section 4.1.2. Moreover, the `get_observation()` function was also updated to include the additional opponent information, from the player object to the observation space.

The number of games and steps trained, as well as the time taken and the number of steps sampled for the final models of each algorithm within this

implementation are displayed in Table 4.8.

Table 4.8 Training Details - Increasing the Agent's Observation

Algorithm	Games Trained	Steps Trained	Steps Sampled	Time Taken
PPO	46,378	2,000,000	2,000,000	44.47hrs
A2C	17,231	825,952	825,952	8.58hrs
DQN	33,105	1,058,976	1,955,432	34.53hrs
DDQN	65,052	2,081,216	3,699,250	64.09hrs

4.2.4 Full Opponent Observation

To further experiment with the performance of the RL agents when more gameplay information is provided, the full opponent observation was provided to the agent. The difference between this implementation and that presented in Section 4.2.3, is that in this implementation, the player knows the exact number of points that the opponent has during each turn of the game as well as exactly which cards the opponent has at all times, including the cards provided to the opponent at the start of the game.

The game environment presented in Section 4.2.1 was used without any modifications performed. However, the RL environment was modified to include the opponent information within the player's observation. The action and observation spaces used in Section 4.2.3 were used in this implementation as the observation space had already been modified to represent the opponent's gameplay information. Moreover, the `get_observation()` function was updated to gather the gameplay information of both agents, from each player object, and encode them as necessary within the observation space.

The time taken, number of games and steps trained, as well as steps sampled, of each algorithm's final model when trained with the full opponent observation are displayed in Table 4.9.

Table 4.9 Training Details - Full Opponent Observation

Algorithm	Games Trained	Steps Trained	Steps Sampled	Time Taken
PPO	51,372	2,200,000	2,200,000	49.63hrs
A2C	16,705	799,008	799,008	8.29hrs
DQN	32,204	1,030,176	1,850,526	32.45hrs
DDQN	53,944	1,725,792	3,023,675	53.25hrs

4.3 Experimenting with Different Techniques

The third objective of this work consisted of experimenting with different techniques, that can be implemented during training, to increase the efficiency of applying RL on competitive multi-agent games characterised by a very large discrete action space, partial observability, and stochasticity.

4.3.1 Policy Cloning during Training

A policy cloning technique was implemented within this work by training the policies from scratch for a certain number of episodes, selecting the best performing policy and applying it to both policies, by creating two separate copies, and conducting further training using the copies. The intention of applying this technique within this work was to ultimately create better policies by cross pollinating the policies with the learned data of the superior policy, especially given the competitive nature of the game, where one policy may perform much better than the other. The final two policies would not be clones of each other, as they would still be trained on different observations.

The initial set of policies for the PPO, A2C, DQN and DDQN algorithms were all trained for up to half the episodes trained with the original RL environment, whilst saving multiple checkpoints. Then, the `player_1` and `player_2` policies were tested quantitatively against each other to determine which policy was performing the best. The top-performing policy was extracted and copied with the use of a number of functions provided by RLlib's Policy class. First, the `from_checkpoint()` function was used to obtain the identified best-performing policy, from the trained model, and the `get_weights()` function was used to extract the policy's weights. Afterwards, a new algorithm model was created, with the same configuration as the model of the first training set, which included two new policies, one for each agent. Following this, the new policies were overridden with the extracted weights of the best-performing policy by implementing a new function, called `RestoreWeightsCallback`, that makes use of RLlib's DefaultCallbacks class. This function calls the `set_weights()` function, from RLlib's Policy class, on both of the new policies to assign them the extracted weights of the best-performing policy. The new algorithm model was then built and the remainder of the training was performed.

Experimentation on the ideal stopping criteria of the first set of policies was conducted by performing further training on the multiple checkpoints saved from the first set of training and evaluating how the performance was affected. The time taken, number of games and steps trained, as well as number of steps sampled, for the initial set of models used for the final models, is displayed in Table 4.10. Meanwhile, Table 4.11 displays these same details for the final models, which include both the initial and

secondary training sets combined.

Table 4.10 Initial Training Details - Policy Switching

Algorithm	Games Trained	Steps Trained	Steps Sampled	Time Taken
PPO	13,463	604,000	604,000	13.75hrs
A2C	3,461	185,088	185,088	1.76hrs
DQN	8,880	283,744	515,554	9.75hrs
DDQN	24,644	788,224	1,417,677	25.87hrs

Table 4.11 Final Training Details - Policy Switching

Algorithm	Games Trained	Steps Trained	Steps Sampled	Time Taken
PPO	41,914	1,804,000	1,804,000	40.14hrs
A2C	16,941	802,528	802,528	8.32hrs
DQN	30,055	960,928	1,751,289	31.26hrs
DDQN	57,162	1,828,384	3,248,417	58.54hrs

4.3.2 Action Embedding

Action embedding consists of mapping the actions to a vector in a continuous space with the intention of representing a large discrete action space in a more efficient manner. This method helps to reduce the dimensionality and complexity of the environment, with the aim of enabling the algorithms to converge faster. Moreover, with the use of action embedding, the agents can also learn to generalise between actions by capturing similarities, which in turn also enables better exploration by allowing the agents to leverage related actions. Taking into consideration Jaipur's very large discrete action space, this method was implemented in this work.

Initially, action embedding was implemented without the use of action masking. Therefore, the game and RL environment were both modified to not make use of an action mask. The function which generates the action mask from the game environment was removed and the observation space of the RL environment was updated to the one shown in Listing 4.3. The 'action_mask' element of the observation space was removed. Moreover, given that only one other element remained, which is the 'observation' element from the previously used `Dict` space, the environment's observation space was converted directly to the 'observation' element's `Box` space. The `observe()` function was also updated to reflect the change made to the observation space by only returning the observation.

```
self.observation_spaces = {agent: spaces.Box(low=np.array
    ([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]),
    high=np.array([7,7,7,7,7,7,5,5,5,5,5,5,5,11,221,11,5,5,5,7,7,9,7,6,5]),
```

```
4       dtype=np.int16) for agent in self.agents}
```

Listing 4.3 Action Embedding Observation Space

For the action embedding to be performed, the RLlib model titled 'ParametricActionsModelThatLearnsEmbeddings' was modified and applied to the custom RLlib environment. This model learns the action embeddings by initially mapping each of the possible actions to a row in a trainable embeddings matrix, within a neural network. For each state, the model will first calculate the embedding of the observation, referred to as the 'intent', with the use of an MLP and outputs an embedding vector of which size is according to the value set to the `action_embed_size` variable. Then, for each action, it will look up the respective embedding vector from the matrix and it will compute the logits by calculating the dot product of the intent with the action's embedding vector. These logits will be used to select the action to be performed. Finally, it will then update the action embeddings and MLP during training by making use of gradient descent.

For each algorithm, multiple training sessions were conducted, with the `action_embed_size` variable being assigned values between 10 and 100, to determine the ideal size of the embedding vector whilst still taking into consideration the efficiency of the model. However, throughout the training, it was noticed that the algorithms kept mostly choosing invalid actions. Due to this, the environment kept truncating as the agents were barely performing any valid actions within the 1000 environment steps limit. Therefore, the algorithms were unable to learn which actions are valid or not based on the state. Values of up to 1000 were also attempted to be assigned to the `action_embed_size` variable however, apart from taking a very long time to train, the models were still not able to distinguish between the valid and invalid actions. This was reasonable considering that the majority of the actions from the very large action space of the game are not valid in a particular state. Moreover, given that the agents only obtain points whilst selling cards, apart from at the end of the game, with the few valid actions that the agents were performing, they were barely obtaining any points prior to the environment being truncated. To try to combat the issue of not receiving different rewards when an action is valid or not, a large negative reward of -1000 was provided when an incorrect action was selected. However, whilst there was a slight increase in the rewards obtained prior to the environment truncating, it was still not enough for the agents of the PPO, A2C, DQN and DDQN algorithms to finish a game within 1000 steps and learn properly. Moreover, given the randomness aspect of the game, the slight increase in rewards could have also resulted by coincidence rather than due to the negative rewards, as the agents were still mainly selecting invalid actions.

Therefore, a decision was taken to apply action masking with the action embedding by using the RLlib action masking and action embedding models together.

Whilst trying to apply both these models together within the RLlib environment, it was noted that only one custom model can be used at a time. Therefore, the models were combined by making use of the action embedding model and then performing the necessary changes for the game's action mask to be applied to mask out the invalid actions from the learnt action embeddings. Moreover, the game and RL environments were reverted back to the ones mentioned in Section 4.2.1, for the action mask to be generated and returned within the environment's observation. Further experimentation by training the algorithms and evaluating their performance with the different embedding vector sizes was conducted, which resulted in the final models being trained with the `action_embed_size` set to a value of 20. Table 4.12 displays the number of games and steps trained, the number of steps sampled and the time taken for the final models to be trained with this technique.

Table 4.12 Training Details - Action Embedding

Algorithm	Games Trained	Steps Trained	Steps Sampled	Time Taken
PPO	36,452	2,000,000	2,000,000	76.98hrs
A2C	27,044	1,534,112	1,534,112	14.29hrs
DQN	27,584	882,304	1,573,593	26.86hrs
DDQN	64,261	2,056,000	3,491,372	57.97hrs

4.3.3 Hierarchical RL with Centralised Critic

The HRL method utilises hierarchy levels to split the main task into multiple meaningful but more feasible sub-tasks, with each level focusing on a particular aspect of the main task. This structured approach is used with the intention of simplifying the learning process for the agents, for which reason, it was implemented within this work.

Multiple approaches were carried out to analyse how best to implement this method. Given the elements of the game, two hierarchy levels were created. The first hierarchy consisted of a pair of high-level agents, one for each player. These two agents would be used to select which action type should be performed from the four main Jaipur actions, mentioned in Section 1.1. Therefore, each high-level agent would have 4 possible actions to choose from. Meanwhile, the second level of the hierarchy consisted of a total of eight low-level agents, representing each of the four Jaipur actions separately, for each player. Hence, for each player, the main 25,499 possible actions were split between the four low-level agents as follows. The low-level agent which handles the action type of taking 1 Goods card from the marketplace had 6 possible actions. The low-level agent in charge of the selling action type had 36 possible actions and the low-level agent in charge of the action type which takes all the Camel cards from the marketplace had 1 action. Meanwhile, the remaining 25,456

actions were all assigned to the low-level agent which handles the trading of cards action type. Since the trading actions constitute most of the gameplay actions, the action masking method was still necessary within this implementation.

Initially, the RL environment was updated to include the high-level and low-level agents as separate agents within the environment. Therefore, the action space was modified, as displayed in Listing 4.4, to reflect this change with separate action spaces, according to the above mentioned possible actions for each agent. Listing 4.4 also displays the order in which the agents were set within the environment. Moreover, the `step()` function within the RL environment was modified so that instead of automatically iterating through all the agents, only the respective low-level agent, according to the action selected by the high-level agent, would perform an action and then the environment would switch to the high-level agent of the next player. The change in agents was not reflected in the Jaipur game environment. Therefore, since the game environment contained the original two players; 'player_1' and 'player_2', the necessary functions were updated so that they would return the title of the respective game player when interacting with the game environment, instead of returning the title of the high-level or low-level agent. Apart from this, the custom RLlib environment was also modified to assign an individual policy to each of these different agents. However, this implementation brought about some concerns pertaining to the fact that points are only awarded when a player sells a card or at the end of the game, if the player has the most number of Camel cards. Moreover, all the actions within the game contribute to each other since a player is not able to sell cards if they do not take cards or trade cards from the marketplace. Therefore, all the policies would need to have a shared critic for the RL agents to be able to learn how to properly play the game and hence, a decision was made for a centralised critic to be implemented within the environment.

```
action_space = ["player_1_high_level": spaces.Discrete(4),
                "player_1_take_1_card": spaces.Discrete(6),
                "player_1_sell_cards": spaces.Discrete(36),
                "player_1_trade_cards": spaces.Discrete(25456),
                "player_1_take_camels": spaces.Discrete(1),
                "player_2_high_level": spaces.Discrete(4),
                "player_2_take_1_card": spaces.Discrete(6),
                "player_2_sell_cards": spaces.Discrete(36),
                "player_2_trade_cards": spaces.Discrete(25456),
                "player_2_take_camels": spaces.Discrete(1)]
```

Listing 4.4 Hierarchical Split Action Space

Since the ready-made RLlib algorithms were used in this work, the implementation of a centralised critic was not as straightforward since the algorithms could not be directly altered. Moreover, copying and modifying the code of the algorithms would be very inefficient due to the large amounts of separate classes and

dependencies used within each algorithm. Therefore, research was performed and reference was made to RLlib's centralised critic example [13], which displays how the centralised critic can be implemented on the PPO algorithm with multiple agents. It should be noted that the particular environment of the mentioned RLlib example had the exact same action space for all agents. Nonetheless, the action space utilised by the centralised critic would need to be able to represent all the different possible actions of all the agents. Therefore, either the actions from the different action spaces of the agents have to be mapped accordingly to the full action space of the centralised critic, or the same action space, which represents all the possible actions individually, would need to be used for all the policies. Should the latter option be used, given that each agent would still have their own portion of possible actions within the full action space, action masking would need to be applied for each agent to only be presented with their own set of actions. Moreover, provided that action masking would still need to be applied for the low-level agent responsible for the very large number of trading options, the method of having the same action space for each agent and masking out the other actions was selected for this work. This resulted in the approach of implementing the centralised critic using RLlib's example to be impractical and a much simpler approach was adopted. Instead of having multiple policies, one for each of the high-level and low-level agents, and then combining them together with the use of a centralised critic, one policy was trained for each agent, as this would ultimately have the same result.

As mentioned above, the action space for this implementation had to be modified to properly encode all the possible high-level and low-level actions. Therefore, the action space displayed in Listing 4.5 was created and assigned to all of the agents. The action space was increased to a `Discrete` space of 25,502 from 25,499, to take into consideration the new high-level actions. Despite having four high-level actions, only three new actions were added since the high-level action to take Camel cards from the marketplace corresponds to its one sub-action.

```
action_space = [spaces.Discrete(25502) for agent in self.agents]
```

Listing 4.5 Hierarchical Combined Action Space

Moreover, whilst the game environment remained unchanged, the RL environment was modified to perform two steps for each agent within one gameplay turn. The first step would be for the high-level agent to select which low-level action type to perform. Subsequently, according to the action type chosen by the high-level agent, the second step would be performed by the respective low-level agent, to select and apply an action to the game environment. For each step, action masking was used to mask out the actions of the other low-level agents, as well as to mask out the invalid

[13] https://github.com/ray-project/ray/blob/master/rllib/examples/centralized_critic.py

actions, from the chosen action type, based on the gameplay information. Therefore, in the RL environment, a new variable called `sep_agent` was added and this variable was set to be initialised with a value of 'player_1_high_level' at the start of each game. This variable is used to keep track of which hierarchical agent is currently selecting an action, as the agents of the RL environment were left as 'player_1' and 'player_2' to correspond to the policies being trained and to the Jaipur game environment.

Furthermore, a new function called `agent_mask()` was implemented which returns a NumPy array of size 25,502, to correspond to all the possible high-level and low-level actions. This function takes in the respective players' action mask that was generated from the game environment, copies it and adds three extra elements, set as invalid, to represent the three extra high-level actions. This was performed as the game environment only considers the two players and therefore this environment's action mask is only applicable for the low-level actions. If the current hierarchical agent is a high-level agent, the `agent_mask()` function will iterate through the game's action mask to check if there is at least one valid action for each type of low-level action. Should this be the case, the high-level element corresponding to that low-level action type within the full action mask will be set as valid. Once all action types are checked, the low-level actions are masked out so that only the valid high-level actions can be selected by the high-level agent. Meanwhile, if the current agent is a low-level agent, this function will mask out the remaining valid actions of the other hierarchical agents. Therefore, if the high-level agent is playing, it will mask out all the valid options except for the 4 high-level possible actions. Similarly, when the high-level agent selects a low-level action to perform, the high-level actions and the possible actions of the other types of low-level agents will be masked out. Moreover, the `observe()` function was modified to pass the `sep_agent` variable to the necessary functions and to call the `agent_mask()` function in order to ultimately return the observation space with the updated observation and action mask.

As displayed in Table 4.13, apart from updating the action space, the observation space of the agents was also changed. The 'action_mask' element of the observation space was modified to a size of 25,502, corresponding to the RL environment's action space. Moreover, five new elements were added to the 'observation' element of this space, to represent each hierarchical agent type with a value of 1, if the agent is playing in the turn, or a value of 0 if otherwise. The `get_observation()` function was also updated to reflect these changes in the observation space.

Additionally, to perform the two above mentioned steps for each player, initially, the `step()` function was modified to call two separate functions, namely `_high_level_step()` and `_low_level_step()`. However, these two functions were then combined within the `step()` function to remove redundant code. The `step()` function

Action Space	$A_S = \{0,1\}^{25,502}$
Observation Space	$O_S = \{A_M, O\}$
Action Mask	$A_M = \{0,1\}^{25,502}$
Observation	$O = \{G, M, C_P, P, C_O, T, H\}$
Player's Goods Cards	$G = \{0,7\}^6$
Marketplace Cards	$M = \{0,5\}^7$
Player's Camel Cards	$C_P = \{0,11\}$
Player's Points	$P = \{0,221\}$
Opponent Camel Cards	$C_O = \{0,11\}$
Remaining Goods and Bonus Tokens	$T = \{\{0,5\}^4, \{0,6\}, \{0,7\}^3, \{0,9\}\}$
Current Hierarchical Agent	$H = \{0,1\}^5$

Table 4.13 Hierarchical Implementation Action and Observation Spaces

will first check which hierarchical agent is playing, with the use of the `sep_agent` variable. If a high-level hierarchical agent is playing, the function will check which low-level action was selected by the high-level agent and it will update the `sep_agent` variable to correspond to the low-level agent that needs to play next. The `step()` function will then terminate and a new turn with the low-level agent of the same player will be performed. When the low-level hierarchical agent is playing, the chosen low-level action is carried out on the game environment and the rewards are updated accordingly. Moreover, at the end of the turn, the `agent_selection` variable is updated to point to the next player, for the next player's policy to be used, and the `sep_agent` variable is also updated to correspond to the next player's high-level agent. It is important to note that, according to PettingZoo's requirements and similar to the implementation discussed in Section 4.1, the rewards will be returned to the policies following the turn of player 2's low-level agent, or after player 1's low-level agent's turn should the game terminate during this turn. Moreover, given that each player only has one policy, the rewards are mapped to the respective player's policy.

Table 4.14 presents the number of games and steps trained, the number of steps sampled as well as the time taken for all four algorithms to be trained with the above-mentioned implementation.

Table 4.14 Training Details - Hierarchical RL with Centralised Critic

Algorithm	Games Trained	Steps Trained	Steps Sampled	Time Taken
PPO	19,892	2,200,000	2,200,000	66.07hrs
A2C	10,133	1,128,416	1,128,416	15.04hrs
DQN	17,707	566,400	2,160,968	49.91hrs
DDQN	36,208	1,158,432	4,400,410	101.58hrs

4.4 Summary

This chapter delved into the different approaches implemented within this work. It presented the initial base implementation of this work along with the various experiments conducted by modifying the environment features and by implementing alternative methods to handle the application of RL agents within the game Jaipur. For each experiment, a thorough explanation was provided of the thought process behind the designs undertaken and how they were implemented. Moreover, all the final models in this work were trained with an Intel Core i5 6th Generation processor and 16GB RAM. The GitHub repository "AI_for_POSGs"[14] contains the full implementation of this work.

[14] https://github.com/Cristina0702/AI_for_POSGs

5 Evaluation

This chapter presents the strategies adopted to evaluate the different elements of this work. It also presents the metrics selected to assess the performance, as well as the obtained results of each algorithm, in each of the performed experiments.

5.1 Implementation Validation

Prior to training the models, testing was initially performed during the code implementation to mitigate syntax and logical errors, confirm that the code is functioning as expected, fix any dependency issues, remove any redundant computations and ensure that best practices are followed.

5.1.1 Initial Implementation with Action Masking

Given that the initial implementation serves as the basis of all the performed experiments, the majority of the checks were carried out on this implementation.

Implementing the Jaipur Game Environment

In the implemented Jaipur game environment, it was crucial that no logical errors were present, to ensure that no changes are performed to the environment which do not correspond to the rules of the game. Hence, many meticulous checks were performed and implemented within this environment.

Each of the functions present within this game environment are equipped with anomaly detection mechanisms to ensure that the provided data is valid and that the changes requested to be performed within the environment comply with Jaipur's game mechanics. Should an attempt be made to perform incorrect behaviour on the environment, the function would return an error message and terminate, preventing the behaviour from being applied. If this were to occur, the player's turn would be concluded and the next player's turn would commence. This was implemented, instead of suspending the entire game, as the agents may try to perform an action during training, which happens to be invalid on that particular gameplay state. Whilst this is acceptable during training, such an action should not be performed within the game environment. The functions also contain relevant assertions which are checked prior to, or following, an action being applied to the environment. For example, when the player selects the option to take and trade cards, a check is performed to ensure that if the action were to be performed, the player's hand size would not exceed 7 cards and the specified cards are truly present within the player's cards and the marketplace. If

this is the case, the changes to the environment would be applied and an assertion would be performed to verify that the player's hand size did not exceed 7 cards.

Initially, each function within the game environment was tested individually to ensure that the mechanisms were working correctly. Both valid and invalid data was provided to each function and the resulting outcomes were checked. This included ensuring that the respective actions and changes were all being performed, with the expected output data being provided accurately as required. It also included testing that all the implemented checks were truly taking into consideration the game state as well as the action selected to be performed, to correctly determine whether or not the requested changes corresponded to the rules of the game. This was important to ensure that all the elements within the game environment were being modified correctly, if the selected action was valid, or not modified at all without causing any issues, if the selected action was invalid. Appendix B displays an example of the test cases applied on the functions of the game environment.

Following all the individual checks, the game environment was simulated to ensure that the functions were working appropriately together, without any errors being raised. Moreover, during the gameplay simulation, multiple game rounds were executed and manual checks were carried out to ensure that the game variables were being updated correctly. Therefore, all the environment's variables, prior to and after an action was performed, as well as the selected actions, were being displayed throughout the entirety of the games. During the gameplay simulation, the actions to be performed were either being selected manually, by inputting the correct integer value which corresponds to an action, or by selecting a random integer during each player's turn, with the use of Python's `randint()` function, where the range was set from 0 to 25469, to correspond to all the possible actions. This was carried out for both valid and invalid actions to be attempted to be performed on the environment.

Implementing the Reinforcement Learning Environment

As clarified in Section 4.1.2, given that the action and observation spaces had to be determined prior to implementing the RL environment, testing was initially carried out on these two spaces.

Designing the Action and Observation Spaces

The designed action and observation spaces were checked with the use of the `contains()` and `sample()` functions, which are both provided by Gymnasium within its' Space class.

The `contains()` function is utilised to check if an element can be contained in a designed space. Therefore, this function was used on both the action and observation

spaces, to verify that they were designed to take in the necessary values. A number of valid values, including the minimum and maximum possible values for each element, were passed as individual parameters to the `contains()` function. Moreover, to ensure that the ranges of the spaces were set correctly, one value less than the minimum value and one value more than the maximum value were also passed to this function. Furthermore, to verify that the observation space would be compatible with the game environment's observation, examples of how the actual game observation would be stored within this space were also provided to the `contains()` function.

Meanwhile, the `sample()` function provides examples of the possible values that can be present in the given space. Therefore, this function was used mainly on the action space to check which actions the agent could select. Moreover, an important feature within this function is that it can also take in a mask of type `np.int8` as a parameter to filter out values according to the mask. This was very beneficial to the RL environment, as the use of the action mask was tested out to ensure that the invalid actions are completely filtered out from the action space. Moreover, this function was also used on the observation space to obtain examples of which values can be stored in this space and ensure that everything was in order.

Developing the Reinforcement Learning Framework

The developed RL environment was tested with the use of the `api_test()` function, which is offered by the PettingZoo library. This function works by simulating the provided custom PettingZoo environment for a given number of cycles, whilst automatically running a number of tests and checks to ensure that the environment complies with PettingZoo's structure and requirements. This function ensures that the variables found in the custom PettingZoo environment are all assigned with valid data, according to the corresponding data types, and updated when required. Moreover, it also ensures that all the functions are being called when necessary and each function is returning the appropriate data types. In this case, the function simulated the RL environment for 1000 cycles to ensure that the interactions with the RL environment were all being performed as required.

Nonetheless, since this function fails to check for logical errors, the RL environment was also checked manually, similar to how the game environment was checked. The environment was simulated numerous times whilst the important variables, which should be updated within the custom PettingZoo environment, were displayed during runtime to verify that they were being assigned and modified correctly. Such variables included the current agent, the selected action, the observation, the rewards, as well as the termination and truncation variables. It was also verified that the termination variable is changed, for both agents simultaneously,

only when the Jaipur game terminates when a player wins the game. Meanwhile, both agents' truncation variable is updated if the environment is simulated for more than 1000 steps. When the termination or truncation variables are set to `True`, it was confirmed that the agents were not allowed to perform any more actions within the environment and that the returned rewards included the rewards obtained in the last round of both players. Moreover, whilst simulating the environment, the use of the game's action mask on the RL environment was observed to make sure that the agent was always selecting a valid action. This was accomplished by implementing a check within the environment's `step()` function which displays an error if an invalid action is selected. A sample of the test cases applied on this environment's methods are displayed in Appendix C.

Implementing the Reinforcement Learning Algorithms

Since the custom PettingZoo environment was converted into an RLlib environment, the action and observation spaces of the RLlib environment were displayed to ensure that they remained the same. After the algorithms were configured, a few iterations of training were carried out, for each algorithm, with the game and RL environment variables being displayed. The same manual checks mentioned in Section 5.1.1 were conducted to verify that everything remained functioning correctly. Apart from this, the displayed gameplay information was compared with the results of the policies. This was performed to confirm that the rewards of each agent were being stored correctly and the number of episodes trained corresponded to the number of games played, as the policies should consider each game as one episode. From the displayed iterations, it could also be verified that all the algorithms were managing to finish each game within less than 1000 turns. Furthermore, during the training of the algorithms, it was observed that no errors were being raised, which confirmed that all the components were working as necessary, including the integrated RLlib action masking model, given that the agents were always selecting a valid action.

5.1.2 Experimenting with Game Features and Hyperparameter Tuning

For each experiment conducted where the action space was increased from the initial implementation, additional observations were provided to the agents and hyperparameter tuning was performed, the necessary checks, which are discussed in the below sections, were carried out.

Increased Action Space

Ensuing the addition of the selling actions, given that changes were performed in both the game and RL environments, a number of checks were carried out on both these environments. For the game environment, the list of actions prior to the trading actions was shown, along with their corresponding index, to ensure that no action was accidentally omitted and that the starting index of the trading dictionary was updated correctly. Moreover, the index value of the last action within the trading dictionary was also displayed to ensure that the number of possible values was also calculated correctly. It was also verified that this value was reflected correctly in the size of both the action mask and the RL environment's action space. Moreover, to ensure that no logical errors were present within the game environment, a number of games were simulated whilst displaying the important gameplay information. From the displayed gameplay steps, manual checks were performed to ensure that the action chosen corresponded to the expected index and that the corresponding number of cards, that were indeed all of the correct type, were being sold. Apart from reapplying the relevant test cases shown in Appendices B and C, Appendix D displays the test cases applied on the new function which was added to the game environment.

Moreover, the PettingZoo `api_test()` and `contains()` functions were executed again. The `api_test()` function was used to verify that the environment remained up to PettingZoo's standards. Meanwhile, the `contains()` function was used to ensure that the range of the action space was set adequately. The RLlib environment's action and observation spaces were also displayed to verify that both the action space, as well as the 'action_mask' element within the observation space, were updated with the correct values. Moreover, by simulating the environment for a few training iterations, it was ensured that everything was functioning appropriately as no errors were being raised. This also verified that the function which creates the action mask was updated correctly as the agents were not selecting an invalid action.

Hyperparameter Tuning

During hyperparameter tuning, checks were performed to ascertain that hyperparameter tuning was being conducted as intended. Therefore, the logs were inspected to ensure that different hyperparameters values were being selected, and that the values which were obtaining the highest episode scores were indeed being chosen as the ideal. The mean, maximum and minimum episode scores obtained from the ideal hyperparameters values during the hyperparameter tuning were analysed to ensure that they were higher than the scores obtained from the previous models that were trained with the previous hyperparameters. However, since during hyperparameter tuning, the models were not fully trained due to the very long training

times, in order to confirm that the hyperparameter tuning was truly effective, the algorithms were trained with the identified ideal hyperparameters. The results, including the player's individual scores, from these models were then compared to the results obtained prior to performing the hyperparameter tuning. Furthermore, given that none of the environments were modified, no additional checks were performed.

Increased Agent Observation

Within the game environment, given that not only were changes made to the environment's functions but new variables were also added, additional checks were implemented within the functions. Moreover, manual checks were also carried out by simulating the environment and by applying the necessary test cases presented in Appendices B, C and D. Whilst the environment was simulated, the added variables, as well as the necessary gameplay information which influence the data stored in these new variables, were displayed. Additionally, the environment was simulated multiple times to ensure that all the actions performed by the opponent were being updated correctly within the player's information.

Furthermore, even though the observation space of the RL environment was increased, since the added ranges were the same as the ranges used for the player's cards information, which had already been set and checked, the `contains()` function was not used. However, given that the function used to update the 'observation' element of the observation space was modified, the data stored in this element was checked to ensure that it was being stored in the correct order. Therefore, the environment was simulated whilst the player's information, including their knowledge about the opponent's cards and score, was displayed along with the structured observation space. The returned data was compared to verify that the variables were being reflected in the correct position within the 'observation' element of the observation space.

Furthermore, the `api_test()` function was used once again, to ensure that there were no issues with PettingZoo's requirement due to the increase in the observation space. Moreover, after converting the environment to an RLlib environment, the action and observation spaces were displayed to confirm that the observation space was also updated accordingly.

Full Opponent Observation

Even though this experiment appears to be similar to the experiment mentioned in Section 4.2.3, the implementation differed between both experiments. Since the game environment was not modified from the environment implemented in Section 4.2.1, no further checks were required. Moreover, the observation space was updated to be the

same as the one used in Section 4.2.3, which had already been checked. However, changes were made to the RL environment's `get_observation()` function and these changes were checked by simulating the environment, displaying the gameplay information, along with the observation space, and ensuring that the observation space's 'observation' element was populated with the expected accurate information in the correct order. A sample of the test cases applied on this function are shown in Appendix C.

5.1.3 Experimenting with Different Techniques

Upon implementing the policy cloning, action embedding and HRL with centralised critic techniques, the necessary checks were conducted to ensure that no logical errors were present and that the implementation was sound.

Policy Cloning during Training

This experiment revolved around cloning the policies during training. Therefore, in this case, the testing was performed when the policies were extracted and copied rather than on the environments, since the environments remained unchanged. To be certain that the `from_checkpoint()` function from the RLlib Policy class was extracting the correct policies from the trained model, a test was conducted were both policies of a trained model were extracted and a number of games were played, with each policy representing a player, to obtain the quantitative results. The same thing was performed with the trained model, without extracting the policies, for the quantitative results to be obtained and compared to make sure that they were the same. Once it was confirmed that the policies were being extracted correctly, the weights were copied. The ideal policy whose weights were copied was compared with the policies of the new model after the weights were overridden, to ensure that the weights of all three policies were the same. By doing so, it was not only confirmed that the weights were copied accurately, but also that the new model would start from two identically trained policies. This was accomplished by making use of the RLlib `get_weights()` function to obtain the weights of each policy which were then displayed and compared iteratively.

Action Embedding

Initially, when the observation space and the `observe()` function of the custom RL environment were modified, the PettingZoo `api_test()` function was called to ensure that the changes made to the environment did not cause any issues. Moreover, during the `api_test()` simulations, the environment information was displayed to ensure that everything remained working appropriately. Furthermore, upon applying the action

embeddings model to the custom RLlib environment, the algorithms were trained for a few iterations whilst displaying the important variables within the model. From these iterations, it was confirmed that the action embedding model was not only implemented correctly, but was also functioning correctly within the environment, as the logic behind how the variables were being modified was sound and no errors were being raised. However, throughout the training it was noted that the actions selected by the algorithms were mostly invalid and the environment kept truncating. Therefore, further training was performed to analyse whether the algorithms would manage to learn with more training iterations. The trained models were evaluated quantitatively, by checking the scores they were obtaining during the training and after the trained policies were applied on simulated games. The scores obtained were very low and the policies were still unable to finish a game, therefore, it was determined that the models did not manage to learn. This process was repeated with the different values set as the `action_embed_size` variable, as well as when the negative rewards were provided to the agent upon selecting an invalid action. Whilst the rewards increased slightly with the use of the negative rewards, as mentioned in Section 4.3.2, none of these trained models were able to play a game within 1000 steps. Moreover, when the negative reward was added to the environment, testing was performed to ensure that this reward was being adequately returned only when the agent selects an invalid action.

Upon deciding to implement action masking with the action embedding, testing was performed to check whether both models could be implemented within the RLlib environment. This was carried out by displaying the variables and changes applied, within both models and the RL environment, to check whether both the action embedding and action masking were being performed. From this testing, it was noted that when more than one model is provided, RLlib only considers the last provided model. Therefore, since only one model can be used at a time, the models were combined and the functionality of the combined model was evaluated. The variables controlling the action embedding and action masking were displayed whilst performing a few test training iterations to analyse how the values of the variables were being modified throughout the training. To further ensure that there were no logical errors in the combined model, the process behind both models individually was also displayed and compared with that of the combined model. Moreover, to determine the ideal value for the `action_embed_size` variable, the models were trained with the different values and their performance was analysed and compared quantitatively.

Hierarchical RL with Centralised Critic

When the HRL technique was implemented, testing was initially performed on the separate action spaces, with the use of the `contains()` and `api_test()` functions. The

`contains()` function was used to ensure that the ranges of the action spaces were assigned correctly, whilst the `api_test()` function was applied to verify that the implementation of the different action spaces was acceptable, according to PettingZoo's requirements, and that no errors were being raised. Moreover, by simulating the environment with the `api_test()` function, it was verified that the environment was switching, as required, from the high-level agent of the current player to the corresponding low-level agent of the same player, and then switching to the high-level agent of the next player. By displaying the gameplay information during these environment simulations, it was also confirmed that the correct player was being returned from the game environment. Moreover, a few training iterations were carried out to ensure that the agents were being assigned the correct policy.

However, when the approach taken was altered, testing was performed again to ensure correct functionality throughout the implementation. The updated full action space was tested with the use of the `contains()` function to ensure that all the high-level and low-level actions were encoded in the action space. Moreover, after performing all the other changes to the functions and observation space of the RL environment, the PettingZoo `api_test()` function was called once again, mainly to ensure that no conflicts were caused with PettingZoo's requirements following the changes applied to the `step()` function. The environment was also simulated to manually check that the elements were being modified accordingly, particularly the `sep_agent` variable and the observation space. From the observation space, it was checked that the correct current hierarchical agent was being encoded in the 'observation' element and that the action mask, stored within the 'action_mask' element, was masking out the other hierarchical agent's actions as well as the invalid actions of the selected action type. It was also ensured that the correct players were being returned to the game environment and that it remained functioning appropriately. Appendix C displays some of the test cases applied whilst checking the RL environment. Moreover, by displaying the environment variables from the simulated environment, it was verified that the rewards were being returned appropriately and during training, it analysed that the rewards were all being assigned to the respective player policy. Moreover, by training a few test iterations whilst displaying the information being stored in the policies, it was also confirmed that everything was in order and that all the necessary gameplay information was being stored in the corresponding policies.

5.2 Experimental Results

Whilst training the models, quantitative tests were conducted during training, to determine each model's optimal training duration for the best possible performance to be achieved, as well as on the trained models, to evaluate and compare their performance. This section will delve into how testing and evaluation was carried out, including an explanation of the metrics selected to be analysed, as well as present, discuss and compare the results obtained from the final models. Moreover, this section will also delve into an action selection analysis which was performed to gather an understanding of the trained agents' behaviour during gameplay.

5.2.1 Quantitative Testing

An analysis of the following metrics was conducted for the trained models to be evaluated quantitatively:

- Player 1 Mean, Max and Min scores
- Player 2 Mean, Max and Min scores
- Episode Mean, Max and Min scores
- Episode length
- Number of games trained

Taking into consideration that Jaipur is a score-based game, where each player's goal is to obtain the highest number of points to win, the metrics concerning the scores obtained during gameplay were chosen to be analysed. Specifically the metrics which display the mean, maximum and minimum rewards obtained by the players individually, as well as cumulatively in each game, were selected as these values clearly reflect the players' performance throughout the games. Moreover, the episode length was also taken into account, as this value represents how many turns were played by both players in a game. This metric is important to evaluate in conjunction with the obtained rewards, since a game might finish after a few rounds resulting in the players not having had enough time to obtain a high number of points. Furthermore, the number of games trained was also selected to analyse and compare the performance of the different algorithms based on the amount of experience they needed to achieve the results.

During the training, the above mentioned metrics were evaluated by visualising each model's training data with the use of TensorFlow's TensorBoard[1] toolkit, as

[1] https://www.tensorflow.org/tensorboard

mentioned in Section 4.1.3. This was performed to gather an understanding of how the policies were behaving throughout the training and identify the ideal training duration of the models. Appendix E displays the line graphs of the values obtained during training, of each of the above-mentioned metrics, from all the final models. To further ensure that the ideal model was selected as the final model, each model's training log was inspected to gather a deeper understanding of each model's performance and select the model checkpoint which provided the greatest results.

Furthermore, in addition to the above evaluation, the trained policies of each algorithm were played against each other on 1000 unique games. This evaluation was repeated for multiple checkpoints of each model to further ensure that the ideal version was selected as the final model. Moreover, this aided in obtaining a more accurate understanding of each model's performance, by further observing the individual policies' performance, including their consistency and robustness, on different scenarios. This test also provided better insight into the differences in performance between the two policies of each algorithm.

The quantitative results obtained by the final models during the last training iteration, as well as after the 1000 simulated games, were compared to the scores usually obtained by human players in the game. In order to gather an idea of these baseline scores, online research was performed and the forums [2] and [3] were discovered. Both these forums contain discussions relating to the average and highest scores obtained by the players in the board game Jaipur. Moreover, an Imgur post[4] was also discovered which contains graphically represented data following approximately 67 rounds of the Jaipur game. The data gathered from these sources show that the individual player typically obtains between 60 to 75 points in a good game, with their highest score typically being between 80 to 100 points. Moreover, some players reported getting scores as low as 16 during gameplay.

The subsequent sections will display the following for each of the final models, along with an evaluation:

- how both players' individual and combined scores varied throughout the training
- the quantitative results, of the above mentioned metrics, obtained from the last training iteration
- the quantitative results obtained following the 1000 games played

[2] https://boardgamegeek.com/thread/702405/what-your-best-round-score-jaipur
[3] https://boardgamegeek.com/thread/3129477/100-points
[4] https://imgur.com/gallery/GizBVGW

Initial Implementation with Action Masking

The results obtained from the last training iteration of the final models within the initial implementation are presented in Table 5.1. Moreover, the results obtained after the final models were played for 1000 games are displayed in Figures 5.2, 5.4, 5.6, 5.8. From these results, it can be seen that all four algorithms generated scores similar to each other and corresponding to those obtained by human players. The DDQN algorithm obtained a similar performance between both policies, while the PPO algorithm's policies presented the greatest difference where the player 2 policy was obtaining very high results with an average of over 71. Moreover, within this implementation, the DQN policies obtained the highest combined scores.

Figures 5.1, 5.3, 5.5, and 5.7 display the change in the average score of the players individually and together, during each algorithm's training process. From these figures, it can be observed that the PPO algorithm encountered some changes made to the policies where the player 1 policy was initially obtaining the greatest rewards and then the player 2 policy discovered a better policy. These policies had a stable performance after 1 million trained steps with a peak in the combined scores, due to the player 2 policy's increase in score, at around 2 million steps. Similarly, the A2C player 1 and player 2 policies were surpassing each other throughout the training until the player 2 policy discovered a better strategy and exceeded the other policy's performance, with both policies displaying a stable convergence after around 600,000 steps. Moreover, the combined average reward kept slightly increasing with minimal fluctuation after around 500,000 steps. Furthermore, the values obtained by the DQN and DDQN policies were fluctuating throughout the training, however, the combined scores were overall improving. Whilst for the DQN algorithm, the player 2 policy started off as the strongest policy and then the player 1 policy surpassed its performance, for the DDQN algorithm, both policies were similar in performance with the player 1 policy starting off and ending with slightly better performance.

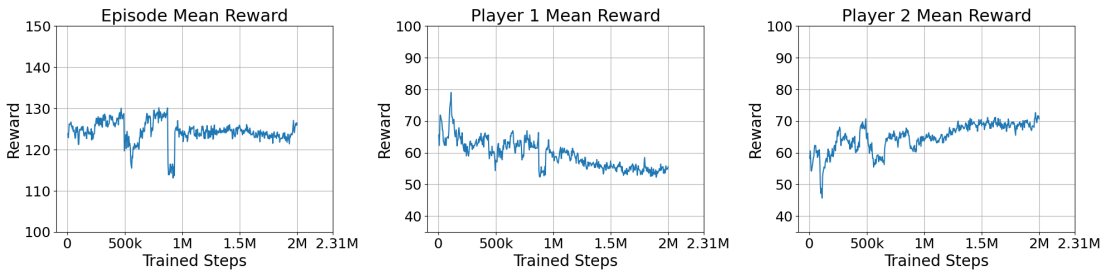


Figure 5.1 PPO - Episode, Player 1 and Player 2 Average Scores - Initial Implementation

Table 5.1 Quantitative Results from the Last Training Iteration - Initial Implementation

Metrics	Algorithms			
	PPO	A2C	DQN	DDQN
Number of Games Trained	38,799	17,114	32,134	64,261
Episode Mean Reward	126.21	122.82	128.28	125.59
Episode Max Reward	143	135	142	140
Episode Min Reward	110	103	114	109
Player 1 Mean Reward	55.55	55.02	67.36	64.87
Player 1 Max Reward	71	78	85	87
Player 1 Min Reward	37	36	48	42
Player 2 Mean Reward	70.66	67.8	60.92	60.72
Player 2 Max Reward	96	88	77	91
Player 2 Min Reward	51	48	38	35
Episode Mean Length	55.11	54.29	54.4	54.32

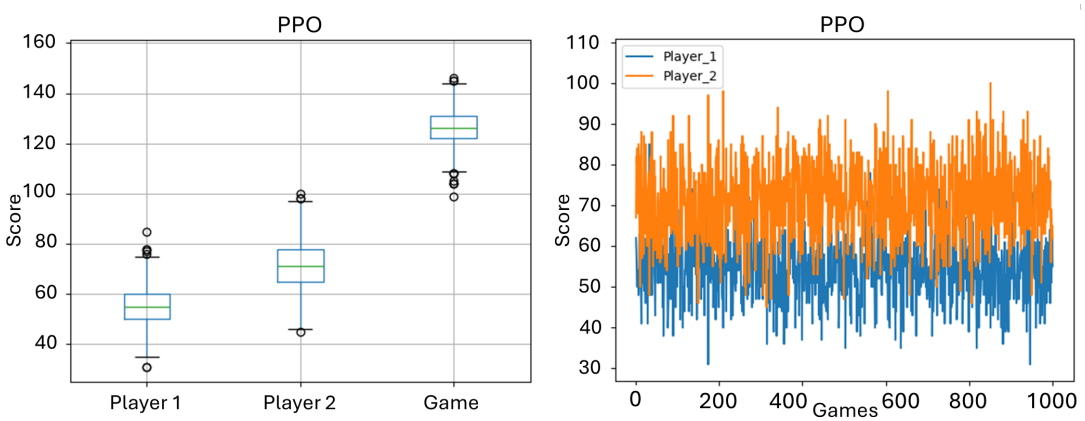


Figure 5.2 PPO - Quantitative Results after 1000 Games - Initial Implementation

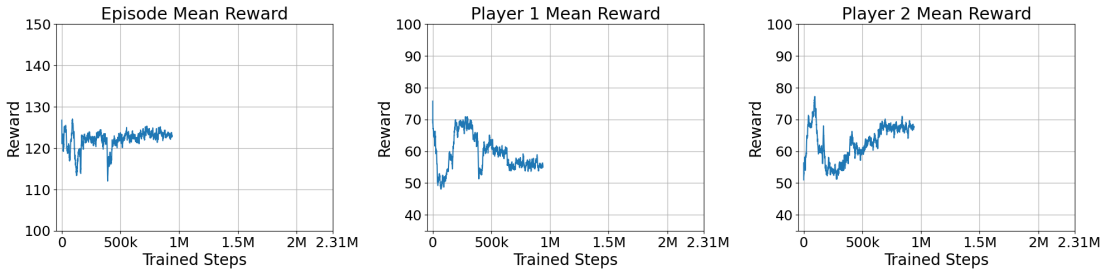


Figure 5.3 A2C Episode, Player 1 and Player 2 Average Scores - Initial Implementation

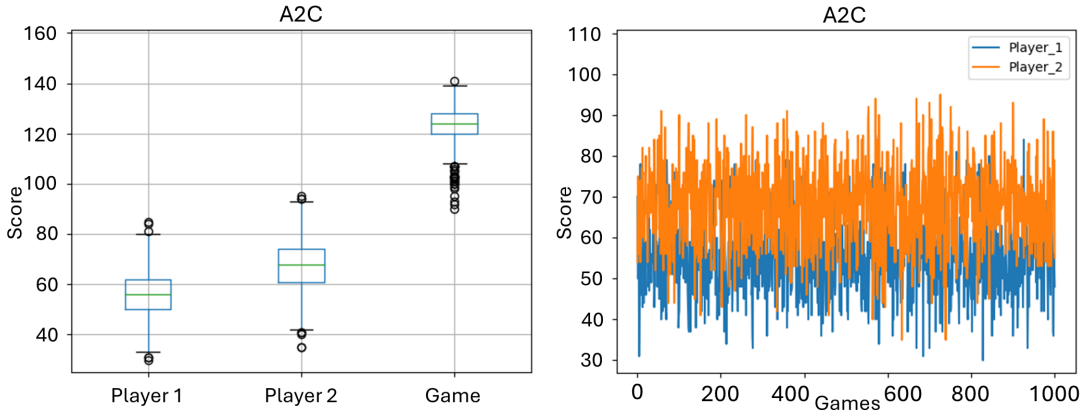


Figure 5.4 A2C - Quantitative Results after 1000 Games - Initial Implementation

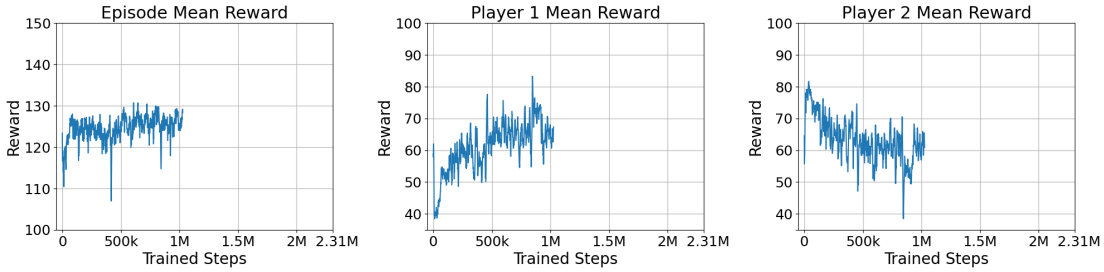


Figure 5.5 DQN Episode, Player 1 and Player 2 Average Scores - Initial Implementation

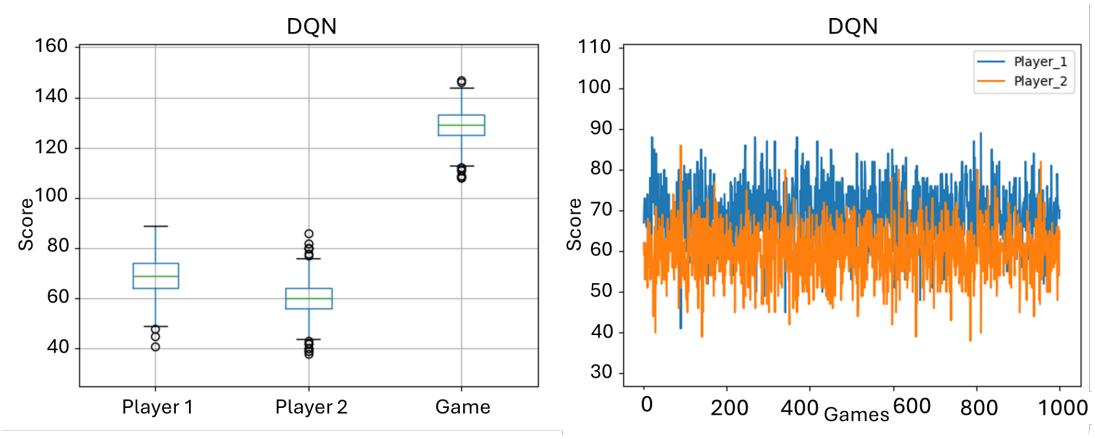


Figure 5.6 DQN - Quantitative Results after 1000 Games - Initial Implementation

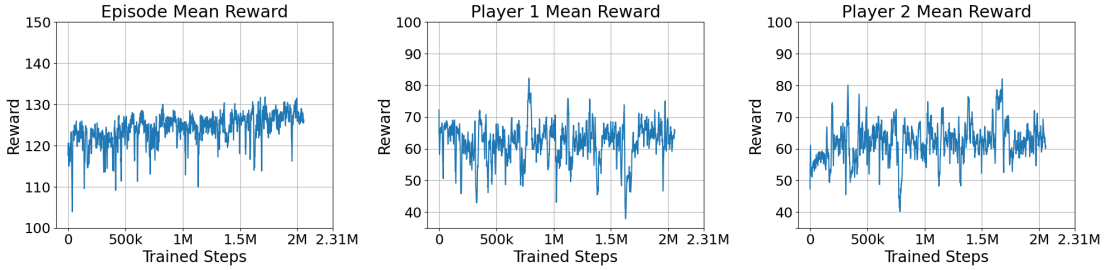


Figure 5.7 DDQN Episode, Player 1 and Player 2 Average Scores - Initial Implementation

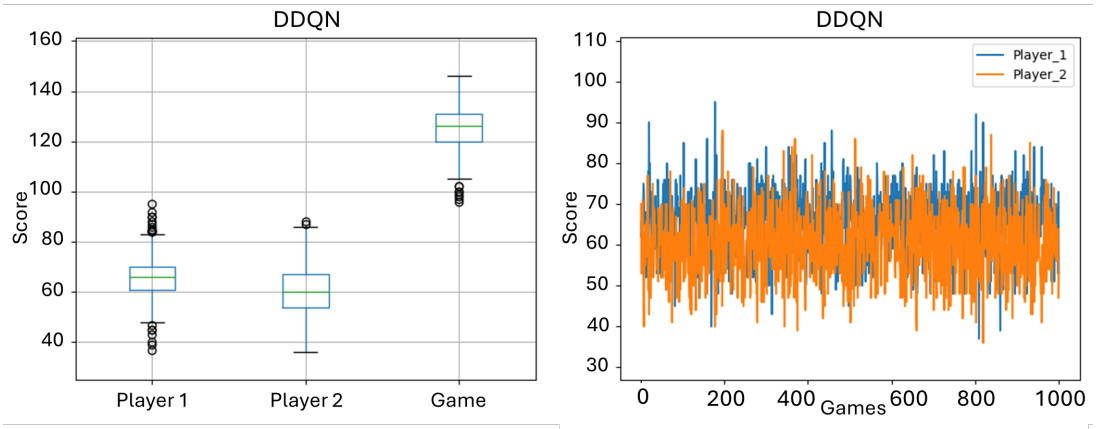


Figure 5.8 DDQN - Quantitative Results after 1000 Games - Initial Implementation

Increased Action Space

The results obtained from the last iteration of the final models' training, when the action space was increased to contain more detailed actions, are presented in Table 5.2. Meanwhile, Figures 5.10, 5.12, 5.14, 5.16 display the results obtained following the 1000 played games. These results display similar scores across all four algorithms which correspond to those achieved by human players. Within this implementation, the DDQN algorithm obtained the highest combined and average scores for both players. Meanwhile, the PPO and A2C algorithms presented the highest maximum scores obtained from both policies individually. Moreover, unlike the scores obtained with the initial implementation, both trained policies of each algorithm achieved scores close to each other, with the greatest difference noted between the policies of the DQN algorithm. However, given the lack of a significantly superior policy, the combined scores were slightly lower than those presented by the initial implementation models, except for the A2C algorithm which obtained slightly higher results. Despite this, the maximum scores achieved by all policies were similar to those of the initial implementation.

The changes in the average score obtained by the players individually and together, for all algorithms, are displayed in Figures 5.9, 5.11, 5.13, and 5.15. The scores of the PPO policies together decreased slightly throughout the training, as both policies balanced each other out, with the player 1 policy starting off as superior, after which the player 2 discovered a better policy that surpassed the performance of the player 1 policy. However, the performance of both policies individually became close to each other, with the player 2 policy ultimately having a slightly better competitive advantage. Meanwhile, for the A2C algorithm, the combined average scores initially decreased up until 500,000 steps, where they increased to a higher amount. Moreover, the player 1 policy, which started off with weaker performance compared to the player 2 policy, began discovering better strategies as its rewards were increasing whilst the

rewards of the player 2 policy kept fluctuating. However, the player 2 policy kept managing to improve its policy and it remained mostly better throughout the training. This policy ultimately ended up having a slightly better performance than the player 1 policy. Meanwhile, for the DQN policies, after around 300,000 steps, similar performance was being obtained, with the player 2 policy performing slightly better than the player 1 policy. Apart from this, their combined average reward kept increasing with less fluctuations than before. Similarly, the gap between the rewards obtained by the individual DDQN policies became small and consistent after around 900,000 steps, where the average game scores, which kept overall increasing throughout the entirety of the training, started stabilising at nearly 130 points.

Table 5.2 Quantitative Results from the Last Training Iteration - Increased Action Space

Metrics	Algorithms			
	PPO	A2C	DQN	DDQN
Number of Games Trained	37,068	16,508	32,138	64,259
Episode Mean Reward	123.1	124.21	124.54	127.33
Episode Max Reward	135	137	137	147
Episode Min Reward	101	108	103	109
Player 1 Mean Reward	59.76	61.21	61.33	62.82
Player 1 Max Reward	82	85	81	87
Player 1 Min Reward	40	43	39	45
Player 2 Mean Reward	63.34	63	63.21	64.51
Player 2 Max Reward	90	80	80	83
Player 2 Min Reward	37	42	46	44
Episode Mean Length	55.04	56.19	57.98	54.4

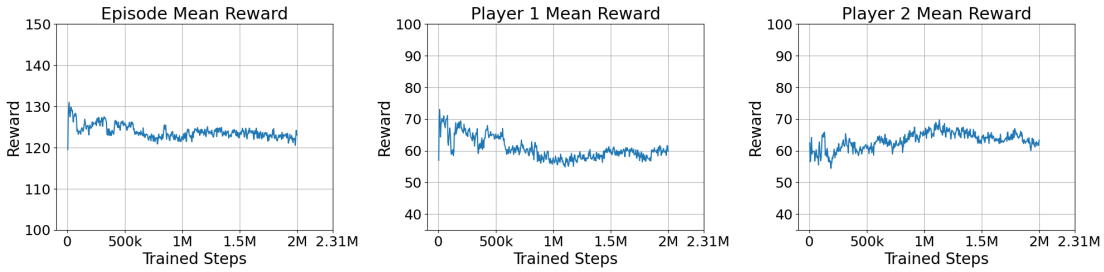


Figure 5.9 PPO Episode, Player 1 and Player 2 Average Scores - Increased Action Space

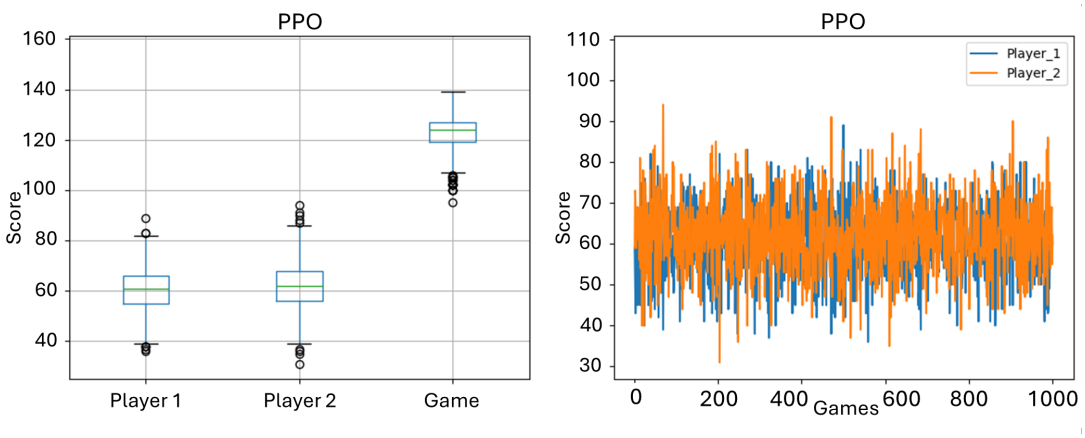


Figure 5.10 PPO - Quantitative Results after 1000 Games - Increased Action Space

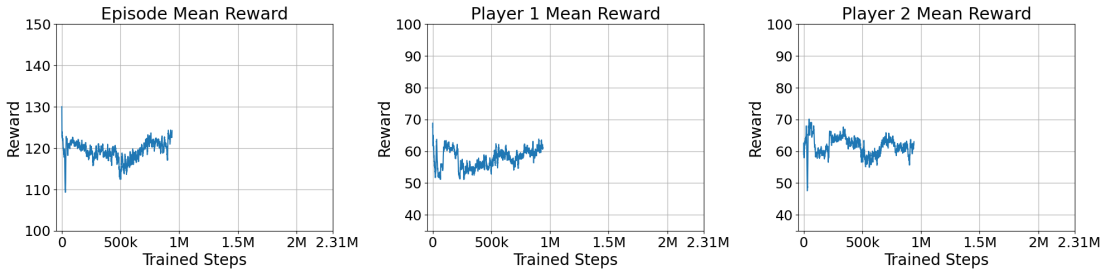


Figure 5.11 A2C Episode, Player 1 and Player 2 Average Scores - Increased Action Space

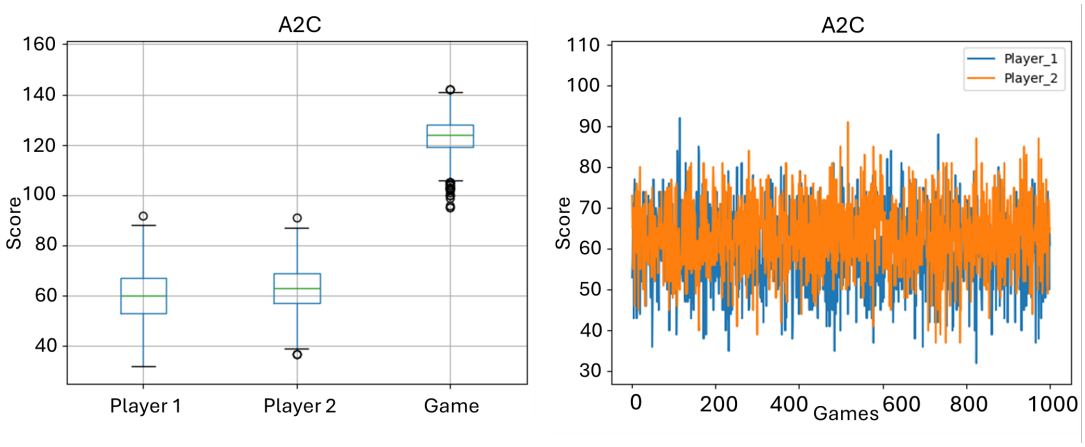


Figure 5.12 A2C - Quantitative Results after 1000 Games - Increased Action Space

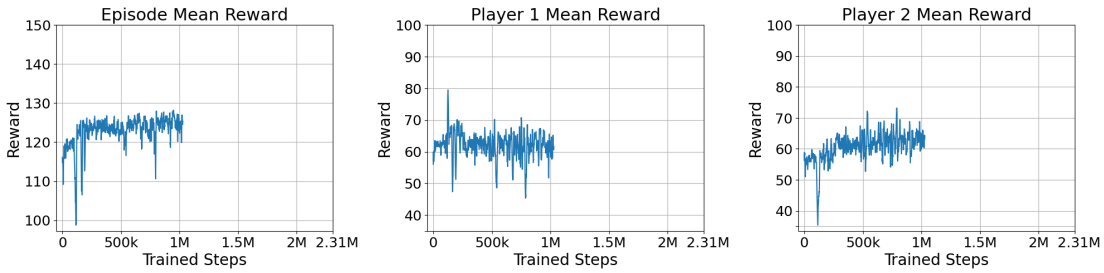


Figure 5.13 DQN Episode, Player 1 and Player 2 Average Scores - Increased Action Space

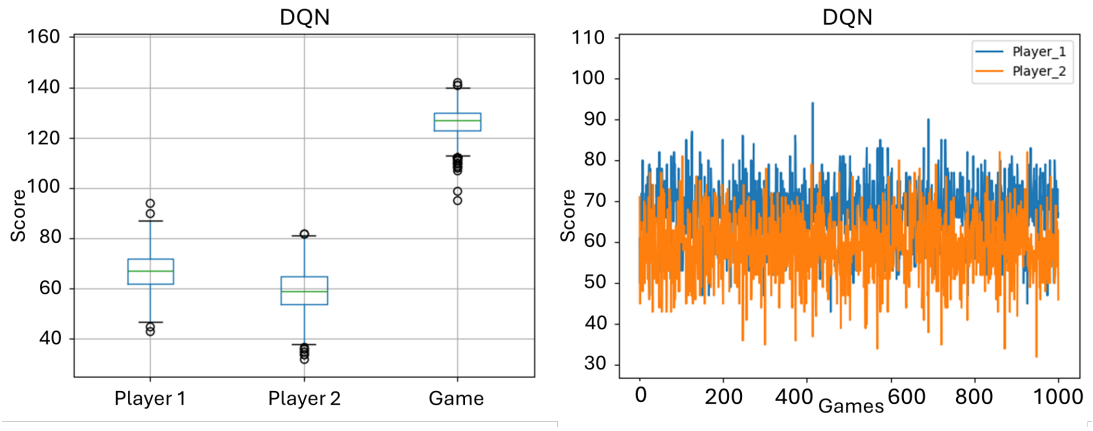


Figure 5.14 DQN - Quantitative Results after 1000 Games - Increased Action Space

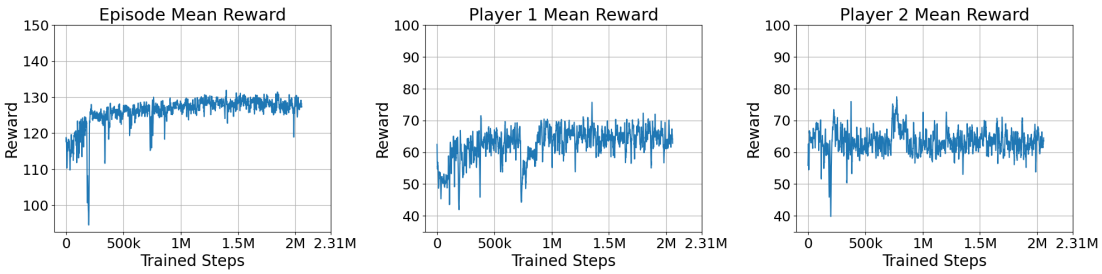


Figure 5.15 DDQN Episode, Player 1 and Player 2 Average Scores - Increased Action Space

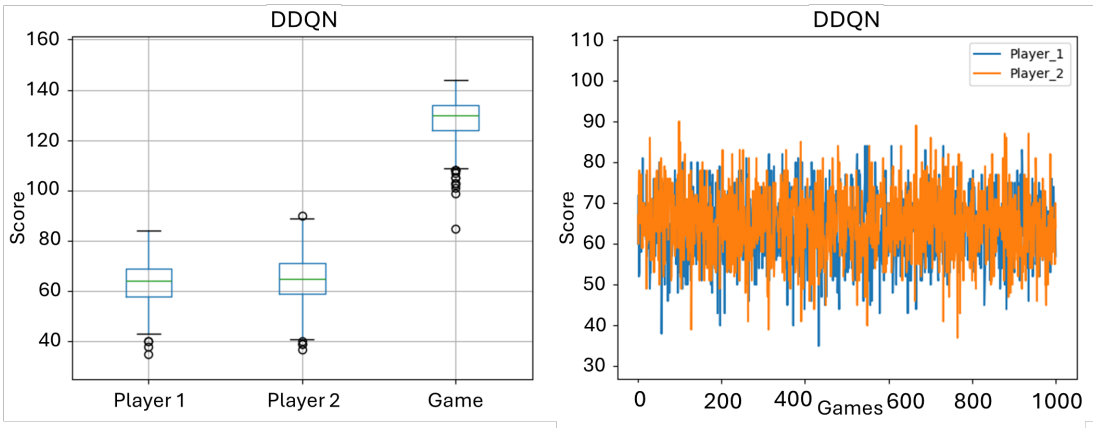


Figure 5.16 DDQN - Quantitative Results after 1000 Games - Increased Action Space

Hyperparameter Tuning

Table 5.3 presents the scores obtained from the last trained step of all the algorithms, when they were trained using the tuned hyperparameters on the game environment with the added actions. Moreover, Figures 5.17, 5.19, 5.21 and 5.23 display how the players' individual and combined scores differed throughout the trained steps. When the hyperparameters were tuned, the PPO and A2C policies improved drastically. Both policies of these algorithms obtained much higher rewards and their training stabilised within fewer number of steps and with much less fluctuations. The PPO policies both started off at around 60 points and they both managed to stabilise to an average of nearly 70 points after less than 500,000 steps. With this amount of trained steps, their combined average also stabilised to a score of 137. This was a great improvement from the previous average scores of 60 and 63, individually, and 123 combined. Similarly, the A2C policies also initially got around 60 points individually and, whilst the player 2 policy had a slight dip in less than 100,000 trained steps, the remainder of the scores were constantly increasing. The player 1 and player 2 policies both converged to an average of around 70 points, compared to the previous average of 61 and 63 respectively. Moreover, previously the average score was 124, whilst with the tuned hyperparameters, the average score was 137. Moreover, for PPO and A2C respectively, the average length of each game decreased from 55 and 56 turns, to 41 and 46 turns. Therefore, since the players were obtaining higher scores in less turns, it shows that the agents were making smarter choices during gameplay.

Meanwhile, the DQN and DDQN policies also obtained higher scores, but the difference was not as remarkable as the difference obtained by the other algorithms. Whilst previously, the DQN policies were obtaining an average score of 61 and 63 individually, and nearly 125 combined, following the hyperparameter tuning, their individual average rewards were nearly 68 and 60 and their average combined score increased to 127. The training process for the DQN algorithm with the tuned hyperparameters was similar to that with the previously used hyperparameters, however, there was less variance in the scores obtained throughout the training with the tuned hyperparameters. Moreover, the DDQN policies obtained an individual average score of 66 and 62, compared to the previously obtained average of 62 and 65 points, with a combined average score of 128 instead of the previous 127. The rewards achieved throughout the training with the tuned hyperparameters were very similar when compared to rewards received prior to the hyperparameter tuning. However, a notable difference was that, initially, the players were obtaining very different scores with player 1 achieving high scores and player 2 obtaining low scores. Despite this, similarly to when this algorithm was trained with the previous hyperparameters, overall the scores obtained by the policies were very similar to each other, with the player 2

policy obtaining a larger range of scores than the player 1 policy. However, it should be noted that the mentioned scores obtained by the DQN and DDQN algorithms were achieved in less trained steps than those mentioned when the algorithms were trained without the tuned hyperparameters. When these algorithms were trained for the same number of steps, as those trained in the increased action space experiment, they started to overfit and the scores started decreasing. This shows that with the tuned hyperparameters, the DQN and DDQN algorithms obtained better scores in less trained and sampled steps.

Figures 5.18, 5.20, 5.22, 5.24 display the scores obtained when the final models were played for 1000 simulated games on the environment with the increased action space. The results presented in these figures correspond to the analysis carried out above, regarding the results obtained from the last training iteration, as well as the change in scores throughout the training. Upon comparing these figures with those presented for the increased action space experiment, it is clear that all the algorithms obtained higher scores with the tuned hyperparameters, especially the PPO and A2C algorithms. Moreover, from these figures, it can also be noted that, for all the algorithms, both policies were achieving similar results to each other, with the greatest difference in policies being observed within the DQN algorithm, where the player 1 policy was better than the player 2 policy. This was also the case for when the previous hyperparameters were used. Therefore, from this experiment it can be concluded that the tuned hyperparameters did indeed improve the performance of all the algorithms with greater sample efficiency.

Table 5.3 Quantitative Results from the Last Training Iteration - Hyperparameter Tuning

Metrics	Algorithms			
	PPO	A2C	DQN	DDQN
Number of Games Trained	46,768	16,851	31,361	57,784
Episode Mean Reward	136.78	137.39	127.33	128.03
Episode Max Reward	155	150	141	144
Episode Min Reward	117	106	112	109
Player 1 Mean Reward	69.28	69.29	67.76	65.87
Player 1 Max Reward	92	88	84	86
Player 1 Min Reward	51	53	49	42
Player 2 Mean Reward	67.5	68.1	59.57	62.16
Player 2 Max Reward	93	84	83	85
Player 2 Min Reward	39	49	36	39
Episode Mean Length	41.32	46.18	57.2	56.15

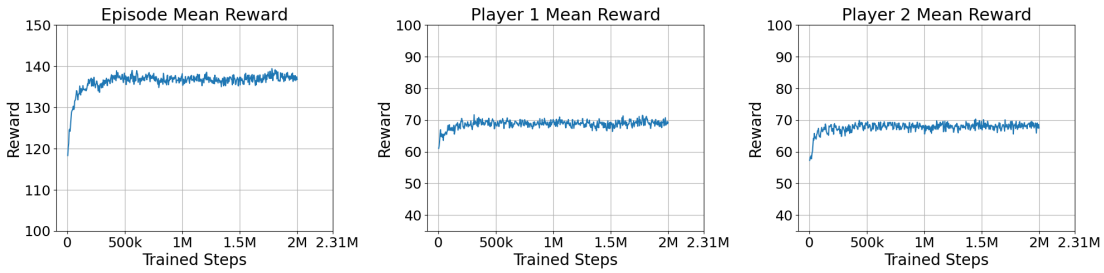


Figure 5.17 PPO Episode, Player 1 and Player 2 Average Scores - Hyperparameter Tuning

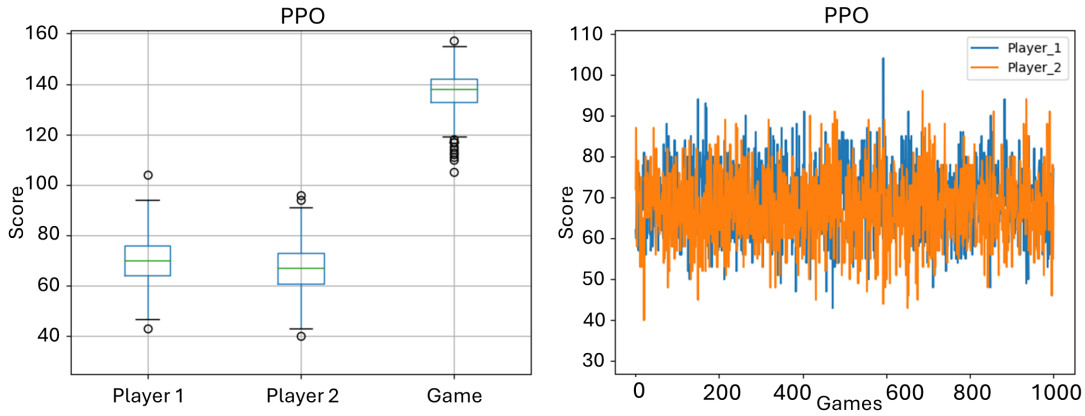


Figure 5.18 PPO - Quantitative Results after 1000 Games - Hyperparameter Tuning

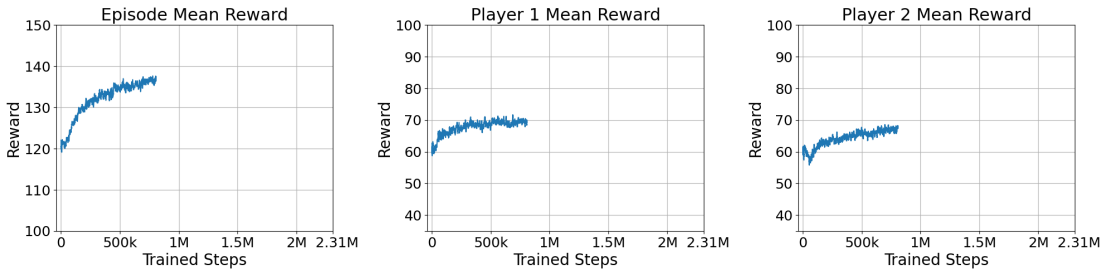


Figure 5.19 A2C Episode, Player 1 and Player 2 Average Scores - Hyperparameter Tuning

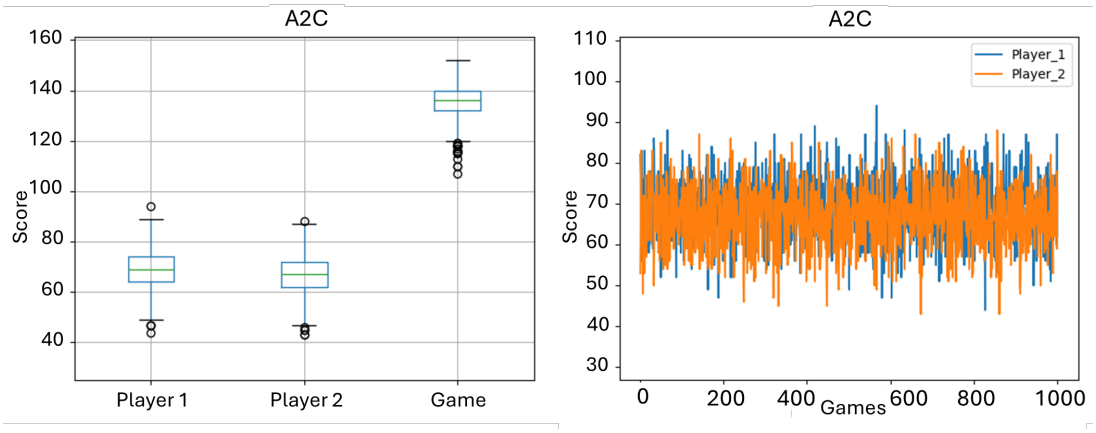


Figure 5.20 A2C - Quantitative Results after 1000 Games - Hyperparameter Tuning

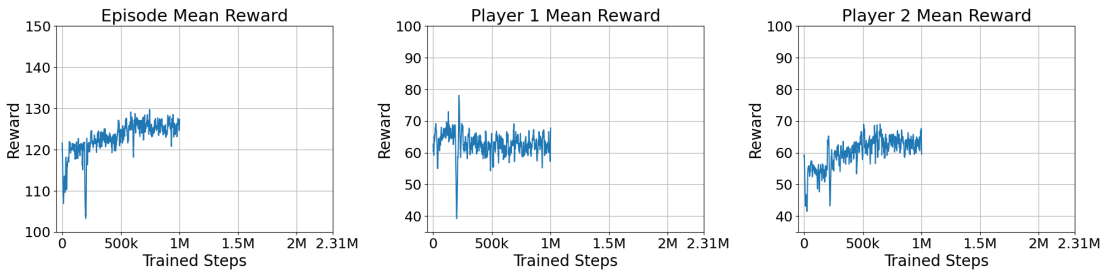


Figure 5.21 DQN Episode, Player 1 and Player 2 Average Scores - Hyperparameter Tuning

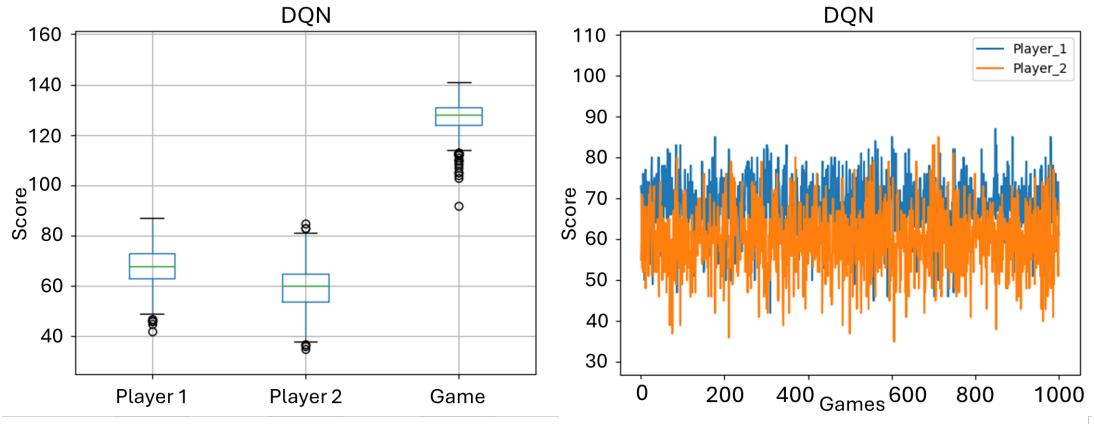


Figure 5.22 DQN - Quantitative Results after 1000 Games - Hyperparameter Tuning

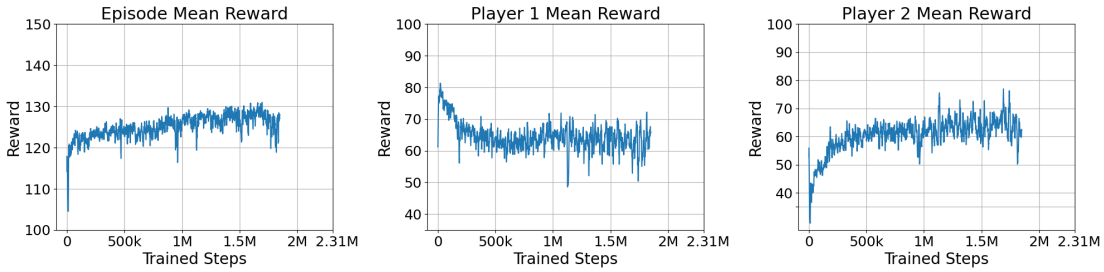


Figure 5.23 DDQN Episode, Player 1 and Player 2 Average Scores - Hyperparameter Tuning

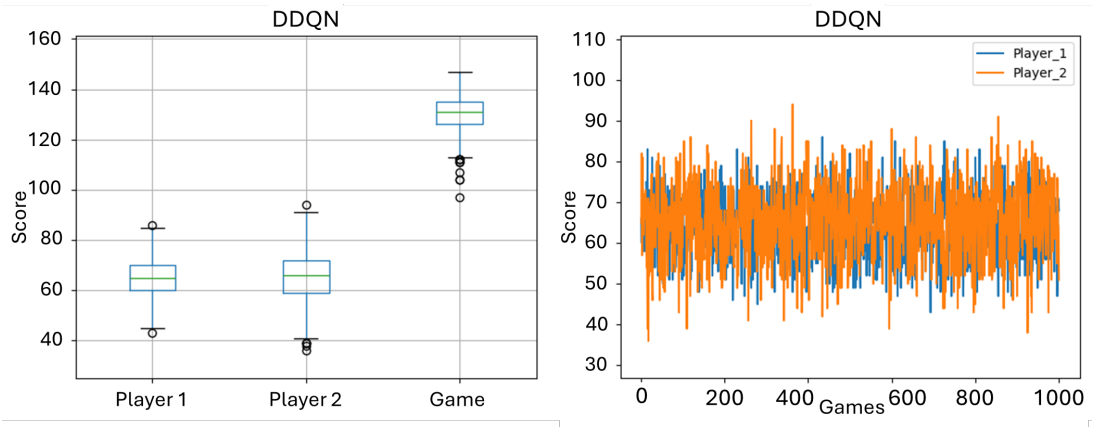


Figure 5.24 DDQN - Quantitative Results after 1000 Games - Hyperparameter Tuning

Increased Agent Observation

The scores obtained, within the final training step of the four algorithms, when the agents were keeping track of the opponent's observable information during gameplay, are displayed in Table 5.4. Moreover, the combined and individual scores of the policies throughout the training are presented in Figures 5.25, 5.27, 5.29 and 5.31, for the PPO, A2C, DQN and DDQN algorithms respectively. The results of this experiment were compared to the results obtained when the models were trained with the increased action space with the tuned hyperparameters. Compared to the results obtained when the training was conducted without the opponent observation, the training progress of all the algorithms was very similar. However, within this implementation, for the PPO and A2C policies, there was a minor improvement where the algorithms achieved slightly higher scores at earlier time steps. Meanwhile, the DQN and DDQN algorithms both took longer to improve during training. Moreover, compared to the models of the tuned hyperparameters experiment, the DQN policies experienced less stability, whilst the DDQN policies displayed greater stability. These findings are also reflected in the results obtained within the final training iteration, where PPO and A2C achieved minimalistic higher average combined and individual scores. Meanwhile, despite being trained for longer, the other algorithms obtained lower combined average return, with an increase in one policy and a decrease in the other policy. Moreover, with this implementation, the average game length for the algorithms was roughly the same as that within the environment without the opponent observations, with the greatest change observed within the DDQN model where the average length was 3 turns shorter.

The results obtained when the policies were simulated on 1000 unique games can be seen in Figures 5.26, 5.28, 5.30, 5.32. These figures display that the individual policies of the algorithms were all obtaining very similar scores to each other, more similar than the individual policies of the tuned hyperparameters experiment. It can be seen that the PPO and A2C algorithms had an improvement in the combined scores obtained when compared to the experiment performed without the opponent's observation. However, given that the individual policies became more balanced, the player 2 policy of both these algorithms, which was the superior policy in the other experiment, obtained slightly less scores while the player 1 policy had an increase in the scores achieved. Meanwhile, the DQN and DDQN algorithms obtained slightly less combined average scores when the opponent's observations were provided. However, for both algorithms, the minimum combined score increased whilst the maximum combined score remained the same, when compared to the previously mentioned experiment. For the DQN policies, the player 1 policy's scores decreased whilst the player 2 policy's scores increased, resulting in the policies within this experiment to

have much less of a reward gap than that observed when the policies were not trained with the opponent's observation. Meanwhile, the DDQN policies both had a decrease in return. Nonetheless, the scores obtained for all the policies within this experiment are still relatively similar to those obtained from the tuned hyperparameters experiment and they also correspond to those achieved by human players.

Table 5.4 Quantitative Results from the Last Training Iteration - Increased Agent Observation

Metrics	Algorithms			
	PPO	A2C	DQN	DDQN
Number of Games Trained	46,378	17,231	33,105	65,052
Episode Mean Reward	137.34	138.76	126.98	127.67
Episode Max Reward	152	152	141	140
Episode Min Reward	119	124	104	102
Player 1 Mean Reward	69.43	70.03	65.68	65.92
Player 1 Max Reward	89	90	85	83
Player 1 Min Reward	43	53	47	45
Player 2 Mean Reward	67.91	68.73	61.3	61.75
Player 2 Max Reward	91	86	89	77
Player 2 Min Reward	49	48	40	44
Episode Mean Length	40.65	46.34	56.48	53.14

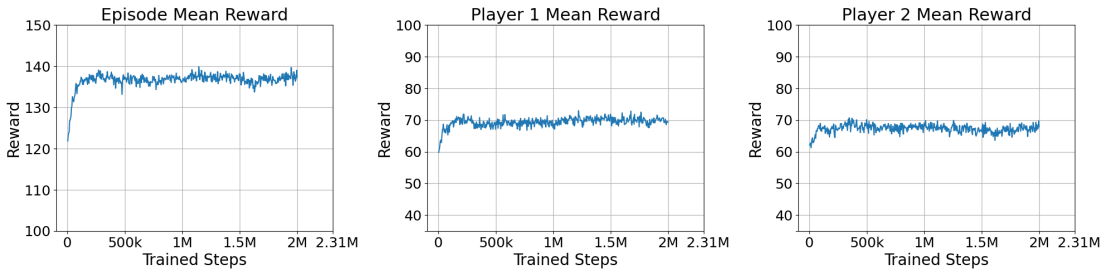


Figure 5.25 PPO Episode, Player 1 and Player 2 Average Scores - Increased Agent Observation

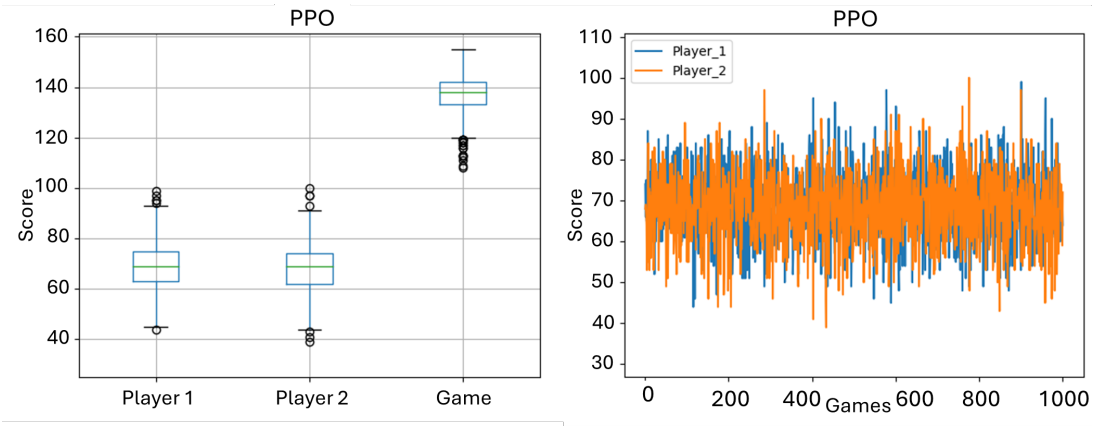


Figure 5.26 PPO - Quantitative Results after 1000 Games - Increased Agent Observation

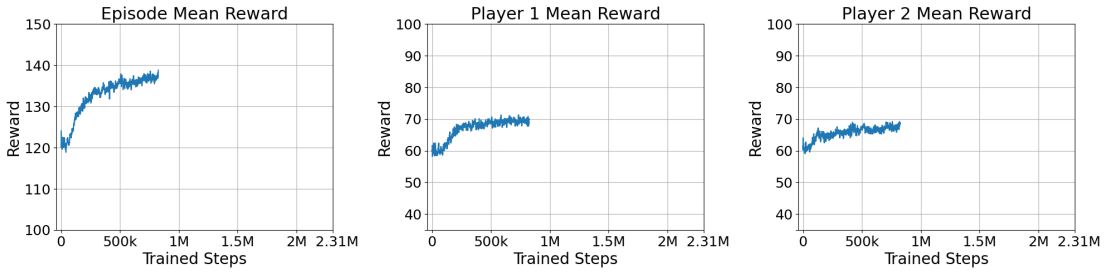


Figure 5.27 A2C Episode, Player 1 and Player 2 Average Scores - Increased Agent Observation

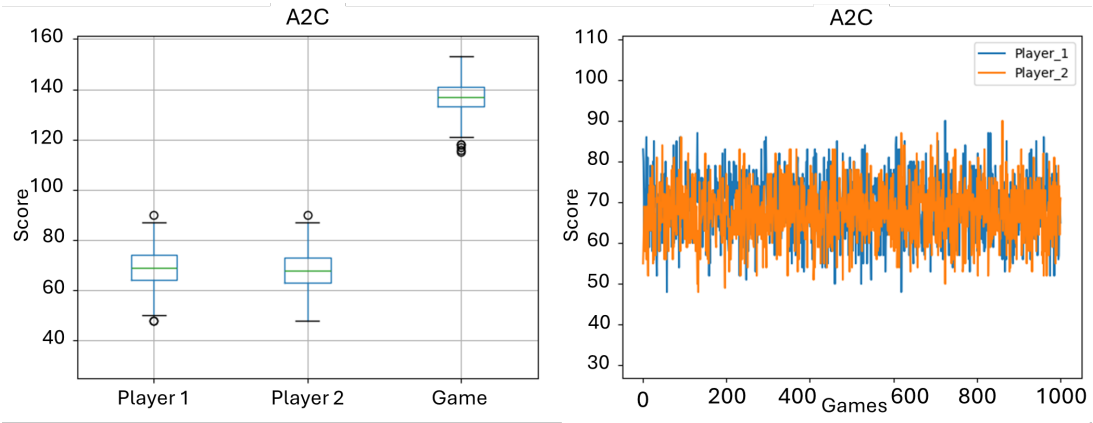


Figure 5.28 A2C - Quantitative Results after 1000 Games - Increased Agent Observation

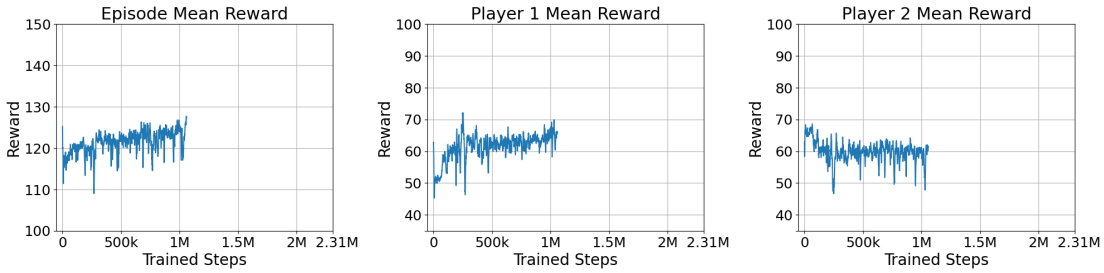


Figure 5.29 DQN Episode, Player 1 and Player 2 Average Scores - Increased Agent Observation

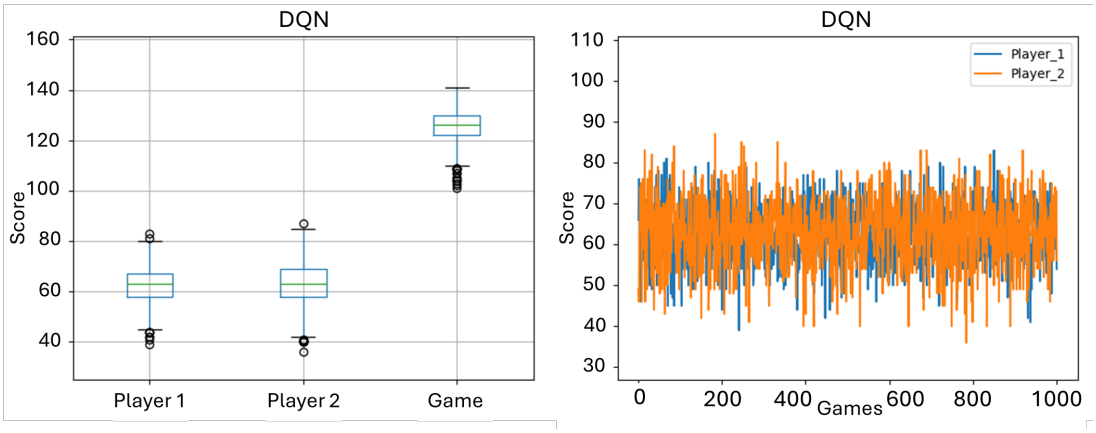


Figure 5.30 DQN - Quantitative Results after 1000 Games - Increased Agent Observation

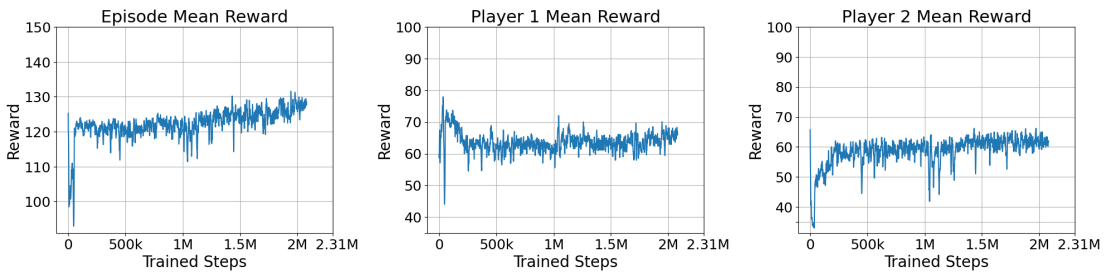


Figure 5.31 DDQN Episode, Player 1 and Player 2 Average Scores - Increased Agent Observation

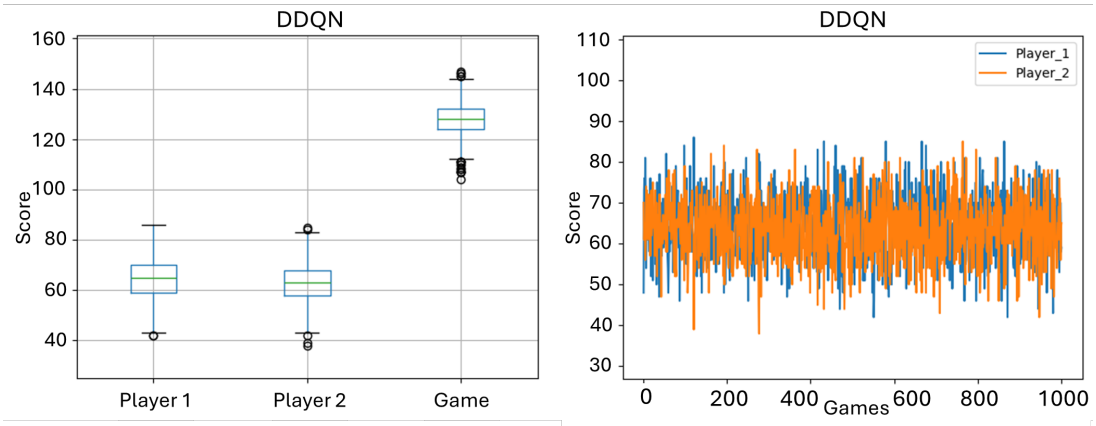


Figure 5.32 DDQN - Quantitative Results after 1000 Games - Increased Agent Observation

Full Opponent Observation

When the full opponent observation was provided to the agents, as can be seen in Table 5.5, which displays the results obtained from the final training iteration of the four algorithms, all the agents were achieving good results compatible with those obtained by human players. Upon comparing these results with the results obtained from the tuned hyperparameters experiment, the PPO and A2C models obtained worse results, whilst the DQN and DDQN models benefited from the additional information. Within this implementation, both PPO policies observed a decrease in return. Meanwhile for the A2C model, the player 1 policy achieved a minor increase in rewards, whilst the player 2 policy had a decrease in rewards. The opposite was observed with the DQN policies, where the rewards of the player 1 policy decreased slightly, whilst the player 2 policy achieved a greater return. Meanwhile, the rewards obtained by both DDQN policies increased. Moreover, all the models were taking slightly less turns to finish the game, with the greatest decrease in the average length noted by 4 turns in the DQN and DDQN models. Furthermore, whilst the PPO and DQN models were trained on more games, the A2C and DDQN were trained for less.

These observations are also reflected in Figures 5.33, 5.35, 5.37 and 5.39, which display the average individual and combined scores obtained by all the algorithms throughout the training. Moreover, from these figures it can be seen that the DQN and DDQN policies had an increase in stability during the training process and they also achieved higher scores at earlier time steps, when compared to the models of the tuned hyperparameters experiment. Meanwhile, Figures 5.34, 5.36, 5.38 and 5.40, display the maximum, minimum and average results obtained from each model following the 1000 simulated games. These results also correspond to the above observation, except for the player 2 policy of the PPO model, which obtained higher rewards, and of the DDQN model, which obtained lower rewards. Moreover,

from these results it can be noted that the PPO and DQN individual policies became more balanced in the scores that they were achieving, whilst the opposite occurred for the A2C and DDQN policies.

Moreover, the results obtained within this implementation were also compared to those obtained when the agent's observation was increased to the observable opponent information. Within this comparison, it was observed that, with the full opponent observation, the PPO and A2C algorithms obtained lower rewards overall, however A2C's player 1 policy obtained slightly better rewards. Meanwhile, the DQN and DDQN models achieved greater combined and individual rewards from this implementation. Furthermore, the A2C, DQN and DDQN algorithms were trained for less games when the full opponent observation was provided, whilst the PPO model was trained for more. Nonetheless, during the training, all the algorithms, particularly the DQN and DDQN algorithms, displayed a slight increase in stability when provided with the full opponent observation. Therefore, overall, the DQN and DDQN algorithms benefitted the most from being provided with the full opponent observation.

Moreover, given that the observed changes in performance were minimal, whilst also taking into consideration the randomness element of the games used within the training sets, it can be concluded that, within the Jaipur environment, all the algorithms managed to perform well and optimise their policies even when characterised by partial observability. This shows that the policies are converging to a dominant strategy of the game which allows them to obtain high scores irrespective of their opponent's hand.

Table 5.5 Quantitative Results from the Last Training Iteration - Full Opponent Observation

Metrics	Algorithms			
	PPO	A2C	DQN	DDQN
Number of Games Trained	51,372	16,705	32,204	53,944
Episode Mean Reward	135.84	136.15	128.92	132.52
Episode Max Reward	152	151	144	144
Episode Min Reward	98	120	109	110
Player 1 Mean Reward	68.93	70.74	66.86	67.35
Player 1 Max Reward	89	90	84	89
Player 1 Min Reward	40	54	40	49
Player 2 Mean Reward	66.91	65.41	62.06	65.17
Player 2 Max Reward	89	83	81	85
Player 2 Min Reward	43	46	47	43
Episode Mean Length	41.96	45.84	53.52	51.69

Figure 5.33 PPO Episode, Player 1 and Player 2 Average Scores - Full Opponent Observation

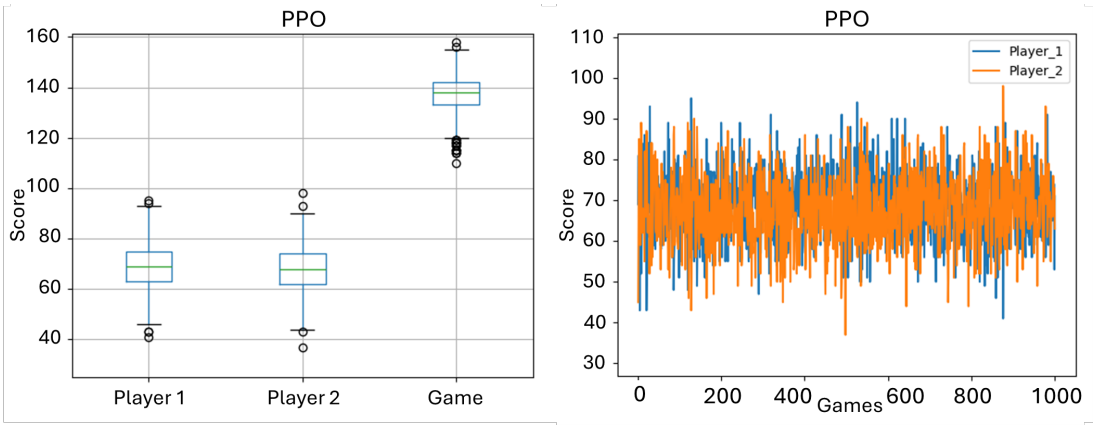


Figure 5.34 PPO - Quantitative Results after 1000 Games - Full Opponent Observation

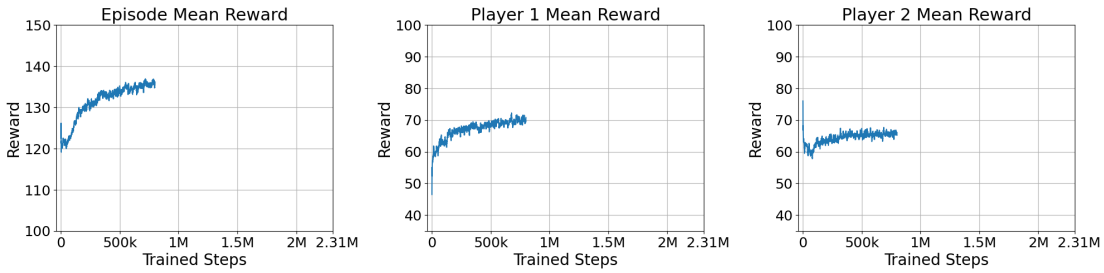


Figure 5.35 A2C Episode, Player 1 and Player 2 Average Scores - Full Opponent Observation

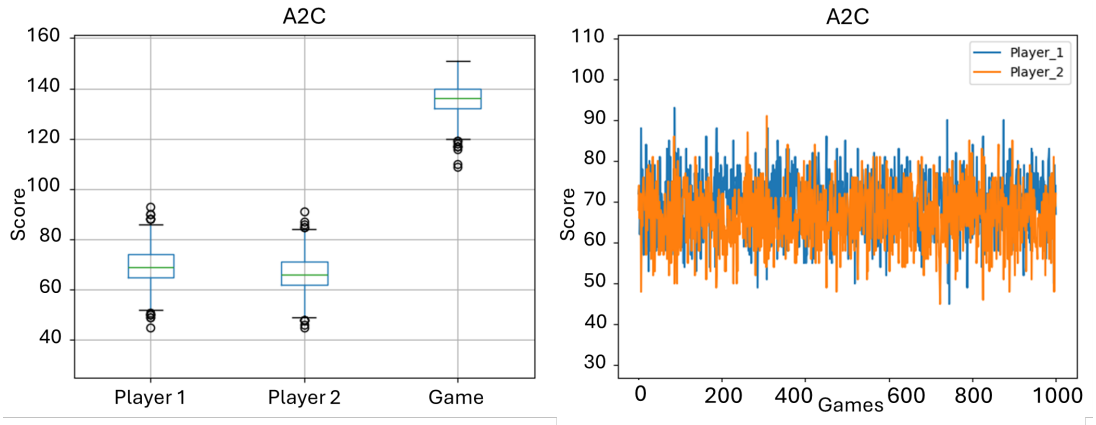


Figure 5.36 A2C - Quantitative Results after 1000 Games - Full Opponent Observation

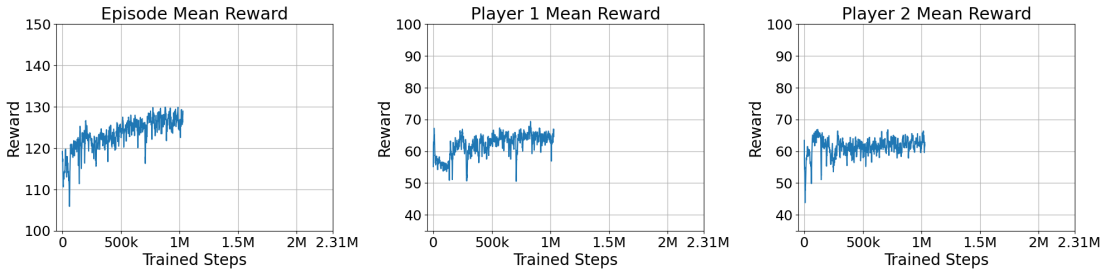


Figure 5.37 DQN Episode, Player 1 and Player 2 Average Scores - Full Opponent Observation

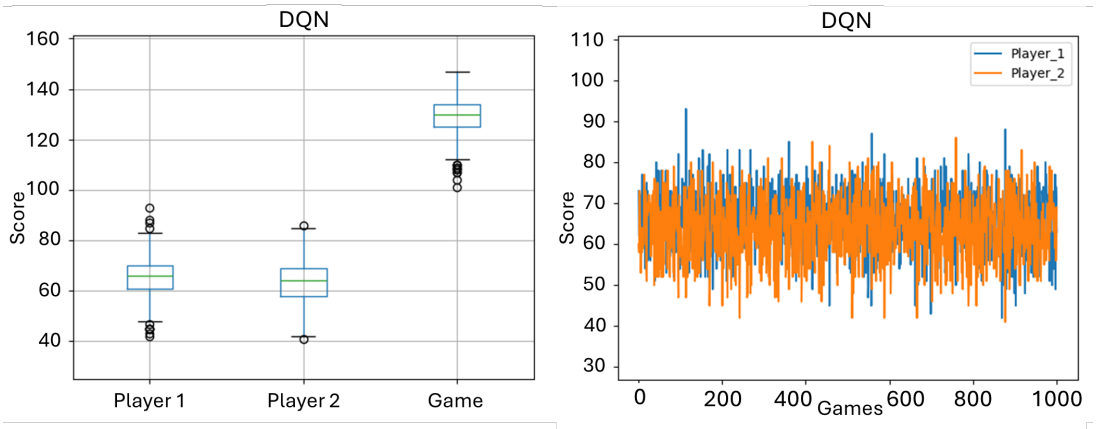


Figure 5.38 DQN - Quantitative Results after 1000 Games - Full Opponent Observation

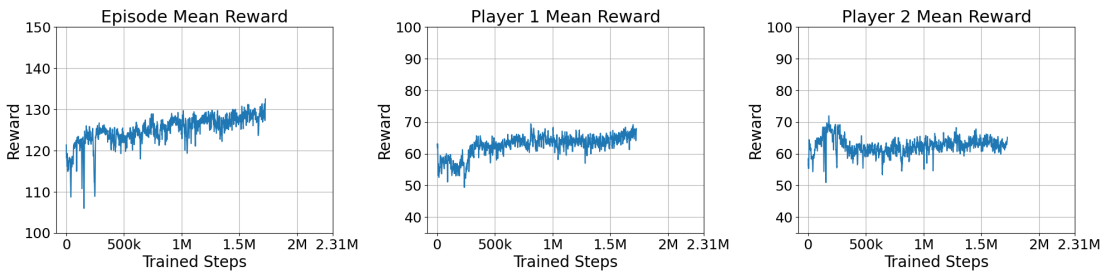


Figure 5.39 DDQN Episode, Player 1 and Player 2 Average Scores - Full Opponent Observation

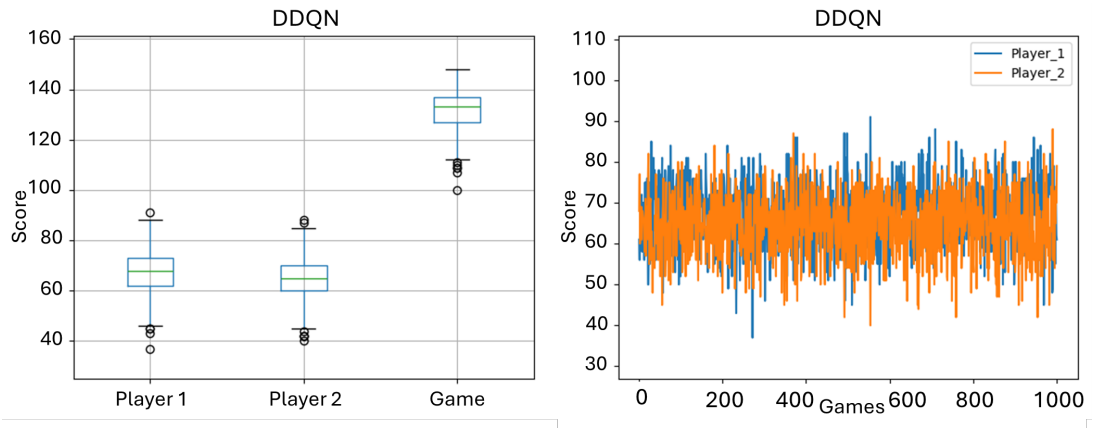


Figure 5.40 DDQN - Quantitative Results after 1000 Games - Full Opponent Observation

Policy Cloning during Training

When the training was interrupted for the superior policy to be copied and used for both players, all the algorithms achieved higher rewards in less amount of trained games. This can be seen by comparing the results displayed in Table 5.6 and Figures 5.41 to 5.48, with the results obtained from the tuned hyperparameters experiment, where the training was carried out on the same environment without the policy cloning technique. Table 5.6 displays the results obtained by all the algorithms from the final training iteration, while Figures 5.41 to 5.48 present the results obtained by all the algorithms during training and following the 1000 simulated games. Moreover, Figures 5.41, 5.43, 5.45 and 5.47 also display the time step, where the policies were switched, in red. Although the final model of the A2C algorithm was trained for 100 additional games, by comparing Figures 5.19 and 5.43, it can still be seen that the policies obtained higher rewards after they were overwritten with the superior policy.

Furthermore, by comparing the scores achieved when the final models were played for 1000 games, it can be seen that all the algorithms obtained a higher combined reward, with the greatest change displayed within the A2C models. The PPO and A2C models were obtaining slightly higher rewards for both individual policies. Meanwhile, for the DQN and DDQN algorithms, the average reward of one policy improved, whilst that of the other decreased slightly. For DQN, this occurred as the reward gap between the policies decreased, whilst in the DDQN model, the reward gap increased with the player 1 policy becoming superior. Therefore, from the observed results, it can be concluded that the use of the policy cloning technique proved to be beneficial.

Table 5.6 Quantitative Results from the Last Training Iteration - Policy Cloning during Training

Metrics	Algorithms			
	PPO	A2C	DQN	DDQN
Number of Games Trained	41,914	16,941	30,055	57,162
Episode Mean Reward	137.9	138.77	127.77	129.39
Episode Max Reward	153	152	142	142
Episode Min Reward	124	118	104	112
Player 1 Mean Reward	70.19	70.33	64.94	68.3
Player 1 Max Reward	90	91	86	84
Player 1 Min Reward	52	49	45	45
Player 2 Mean Reward	67.71	68.44	62.83	61.09
Player 2 Max Reward	92	85	77	82
Player 2 Min Reward	46	49	40	37
Episode Mean Length	42.07	44.42	57.91	54.95

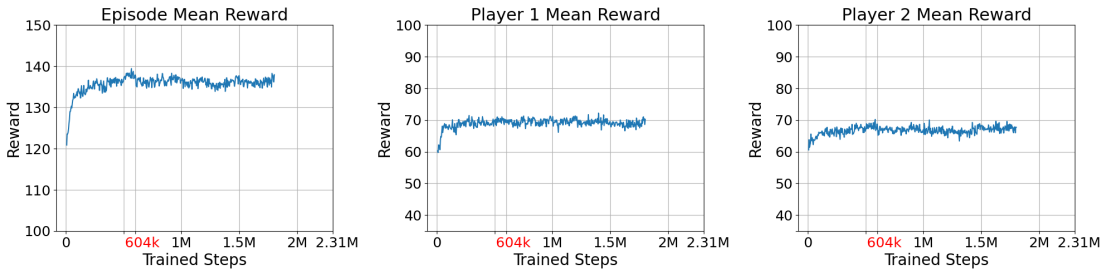


Figure 5.41 PPO Episode, Player 1 and Player 2 Average Scores - Policy Cloning during Training

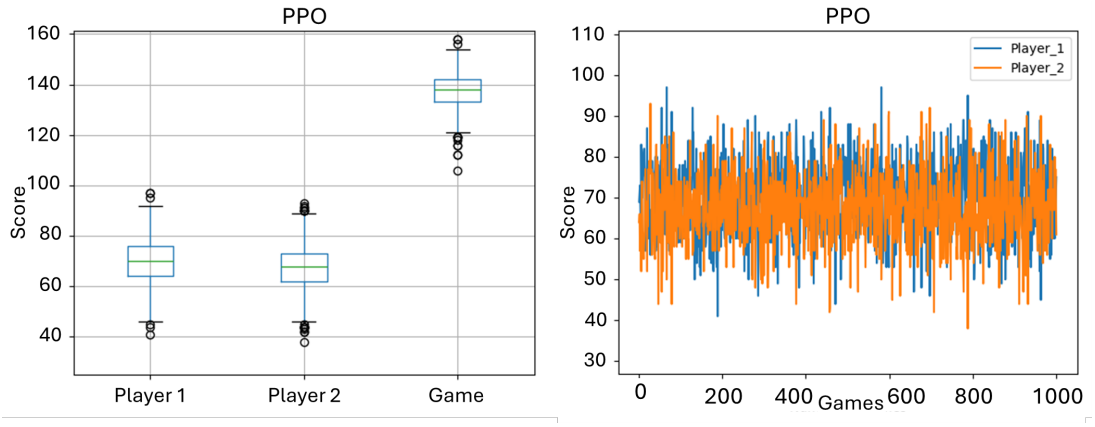


Figure 5.42 PPO - Quantitative Results after 1000 Games - Policy Cloning during Training

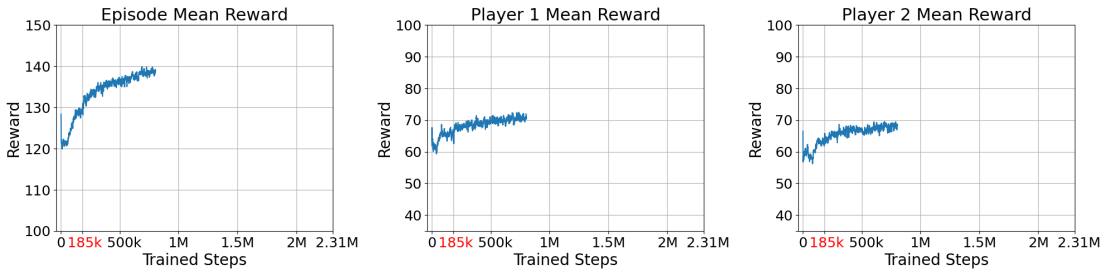


Figure 5.43 A2C Episode, Player 1 and Player 2 Average Scores - Policy Cloning during Training

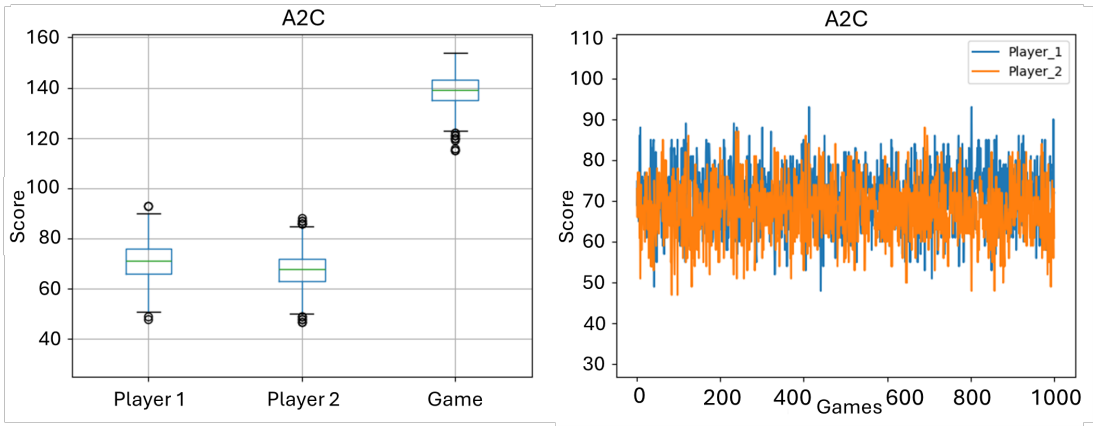


Figure 5.44 A2C - Quantitative Results after 1000 Games - Policy Cloning during Training

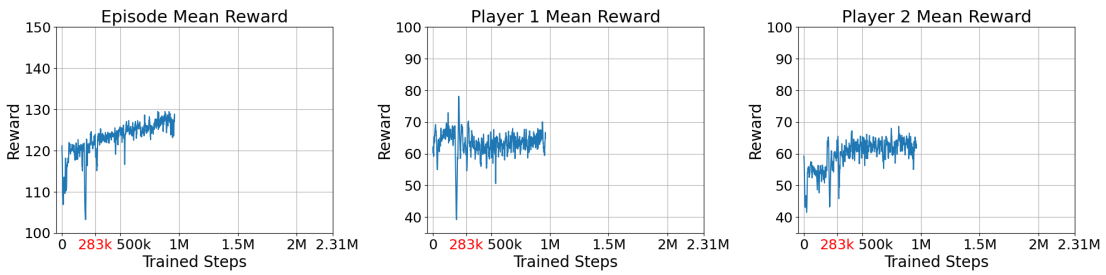


Figure 5.45 DQN Episode, Player 1 and Player 2 Average Scores - Policy Cloning during Training

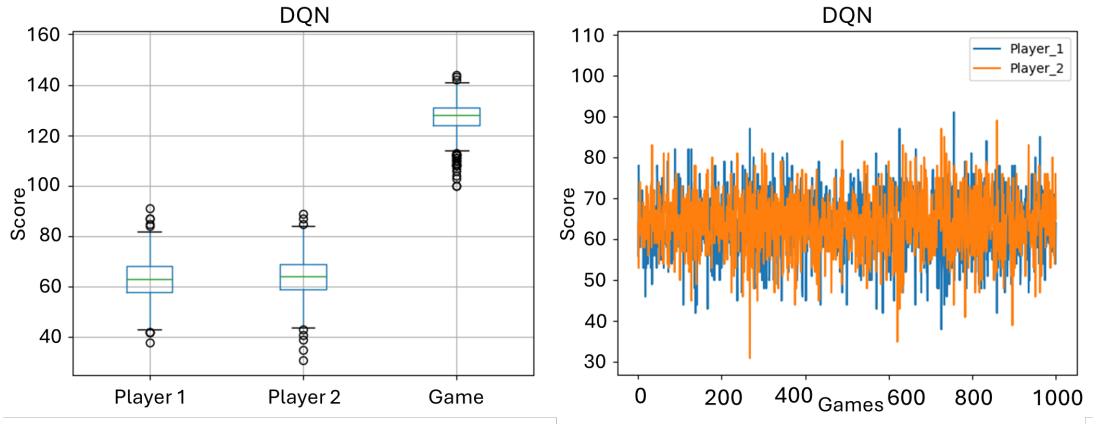


Figure 5.46 DQN - Quantitative Results after 1000 Games - Policy Cloning during Training

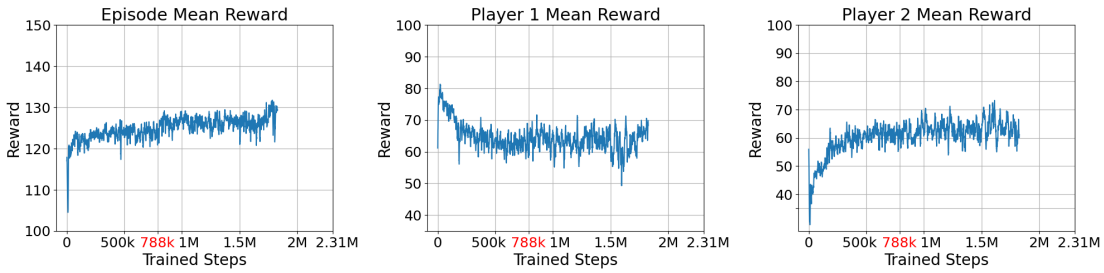


Figure 5.47 DDQN Episode, Player 1 and Player 2 Average Scores - Policy Cloning during Training

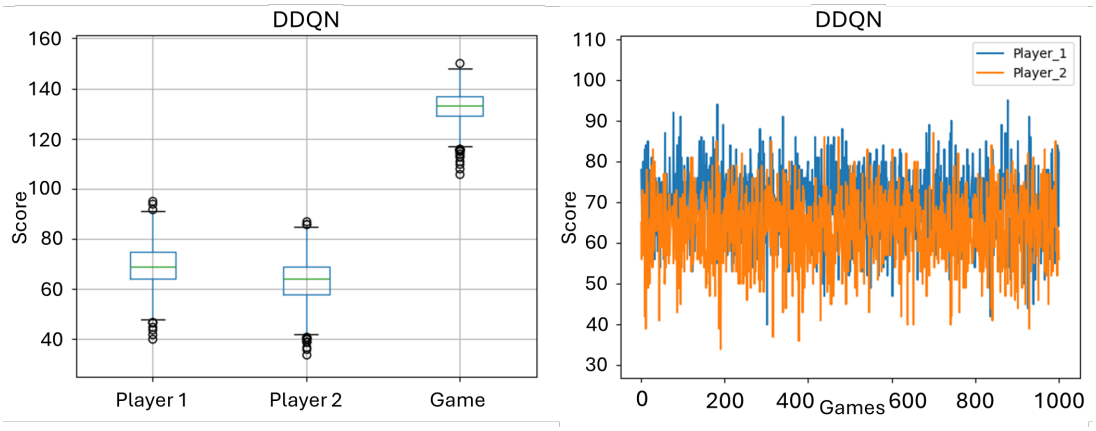


Figure 5.48 DDQN - Quantitative Results after 1000 Games - Policy Cloning during Training

Action Embedding

The scores achieved within the last training step of all the algorithms, with the action embedding technique, are displayed in Table 5.7. Meanwhile, Figures 5.49, 5.51, 5.53, and 5.55, present how the average scores of the policies, individually and in each game, fluctuated throughout the training. These results display that for the PPO algorithm, this technique resulted in very unbalanced individual policies as the player 1 policy was significantly stronger than the player 2 policy, with an average score of 76 compared to player 2's 53 points. Similarly, the A2C player 2 policy was stronger than the player 1 policy, with average scores of 69 and 52 respectively. Meanwhile, whilst the DQN and DDQN models both had a clear superior policy, the difference in scores between the individual policies were not that significant when compared to those noted within the PPO and A2C models. Moreover, during PPO's training, the player 1 policy discovered a better policy after 1 million steps as it started obtaining higher rewards. Nonetheless, it was interesting to observe that the player 2 policy also managed to increase its rewards whilst trying to compete against the player 1 policy. Subsequently, this resulted in a great increase in the combined average scores of the players. Meanwhile, for the A2C model, both policies were obtaining similar rewards until around 600,000 trained steps, where the player 2 policy's score increased greatly, which resulted in player 1's scores to decrease. At around 900,000 steps, the player 1 policy managed to obtain higher rewards, which caused the player 2 policy's scores to decrease, however the player 2 policy succeeded in gaining a superior strategy once again. Meanwhile, the DQN and DDQN models presented a lot of variance in the scores obtained throughout the training, however both of these algorithms' individual policies were obtaining relatively similar scores.

Following the 1000 played games, for which the results are displayed in Figures 5.50, 5.52, 5.54, and 5.56, the above observation remained consistent for the PPO and A2C algorithms. Both these algorithms presented a very strong policy along with a weaker policy, with the average scores obtained being nearly identical to those discussed above. However, for the DQN and DDQN models, the individual policies were obtaining scores similar to each other, with the average reward of all individual policies being greater than 60, compatible with the results obtained by human players. The PPO and A2C models also obtained scores compatible with those discussed in the forums, as despite the weaker policy of both models obtaining an average of less than 60, given that the competing policy was very strong, the rewards achieved by the weaker policies are still considered to be good. Moreover, the difference in scores observed between the results from the final training step and following the 1000 games played, of the DQN and DDQN policies, further displays the importance of applying the trained models on simulated unique games to gather an accurate analysis

of each policy's performance.

Furthermore, given that the models within this experiment were trained with the previously used hyperparameters, prior to hyperparameter tuning, the comparison was performed with the results obtained from the increased action space experiment where the models were trained with action masking only. With the action embedding technique, the PPO and DQN models were trained for a few less games, while the DDQN model was trained for 2 more games and the A2C model was trained for 10,536 more games. The PPO and A2C models took longer to train, with PPO requiring almost double the training time taken for the action masking models, showing that additional computational complexities were brought about from this technique. Meanwhile, the opposite was observed for the DQN and DDQN models, as the models took less time to sample and train steps and they also obtained a greater ratio of trained to sampled steps. The architecture of PPO and A2C could have contributed to the decrease in efficiency since they rely on on-policy learning, where they frequently sample new actions. The greatest decrease in efficiency was noted within PPO due to this algorithm's frequent policy updates, whilst performing clipping to ensure stability, in comparison to A2C, which limits the number of policy updates performed during training to one update per batch. Meanwhile, the DQN and DDQN algorithms employ off-policy learning, where they make use of a replay buffer to store and reuse past experiences. Therefore, this element contributed to the DQN and DDQN algorithms' ability to remain efficient despite the added complexities.

Moreover, when comparing the results from the final training iteration of this experiment to those obtained with the increased action space models, it was observed that the PPO model achieved higher average, minimum and maximum combined scores, as well as higher player 1 scores, whilst the player 2 scores decreased. Meanwhile, A2C and DDQN achieved lower combined and player 1 scores, with higher player 2 scores, and the DQN model obtained lower combined and player 2 scores, with higher player 1 scores. Most of these observations were consistent with the results obtained when the policies were applied to the unique games, except that the scores of the DQN player 1 policy decreased whilst those of the player 2 policy increased, and both the DDQN individual policies' scores decreased. Moreover, from these results, it was noted that the action embedding PPO and A2C models contained a much greater gap between the rewards obtained by their individual policies, when compared to the respective action masking models. Meanwhile, the individual policies of the DQN and DDQN models became more balanced. Furthermore, some slight changes in the average game length were also noticed with the PPO and DQN models playing slightly less turns than the respective action masking models, whilst the A2C and DDQN models resulted in slightly longer games. Therefore, overall, the action masking technique proved to be more beneficial across all algorithms, as despite

obtaining marginally lower rewards with the PPO algorithm, it was still much more computationally efficient than the action embedding technique.

Table 5.7 Quantitative Results from the Last Training Iteration - Action Embedding

Metrics	Algorithms			
	PPO	A2C	DQN	DDQN
Number of Games Trained	36,452	27,044	27,584	64,261
Episode Mean Reward	129.09	120.17	123.96	124.92
Episode Max Reward	144	135	135	135
Episode Min Reward	108	96	103	106
Player 1 Mean Reward	75.61	51.65	67.47	59.88
Player 1 Max Reward	97	83	87	80
Player 1 Min Reward	52	35	52	41
Player 2 Mean Reward	53.48	68.52	56.49	65.04
Player 2 Max Reward	76	89	73	88
Player 2 Min Reward	34	42	33	45
Episode Mean Length	53.16	60.12	57.02	55.55

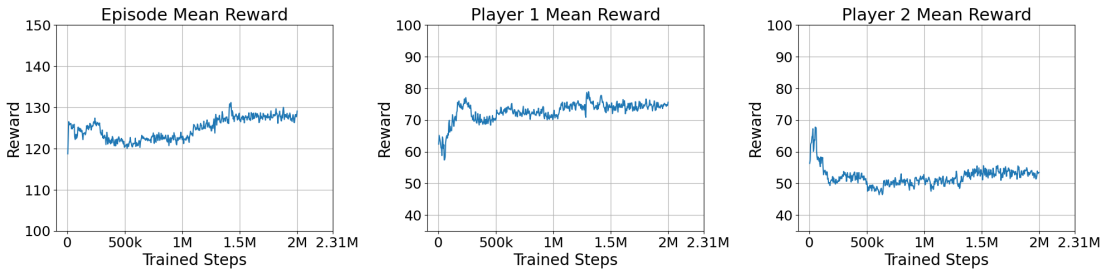


Figure 5.49 PPO Episode, Player 1 and Player 2 Average Scores - Action Embedding

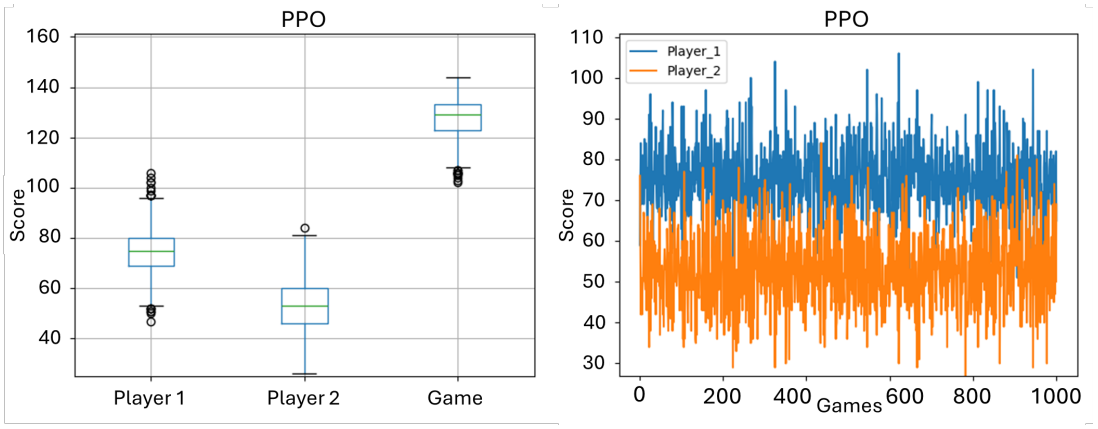


Figure 5.50 PPO - Quantitative Results after 1000 Games - Action Embedding

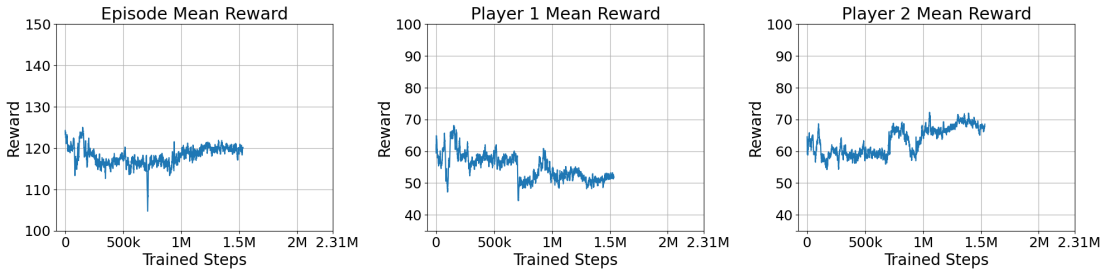


Figure 5.51 A2C Episode, Player 1 and Player 2 Average Scores - Action Embedding

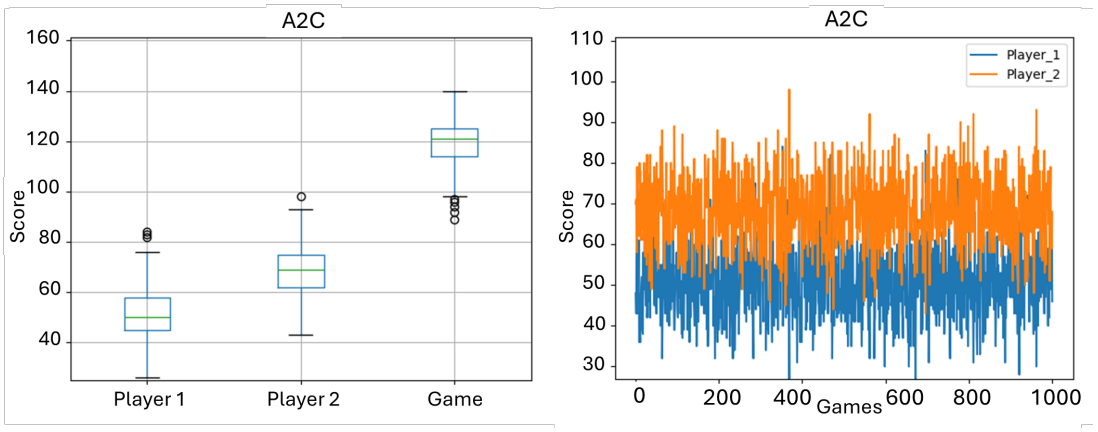


Figure 5.52 A2C - Quantitative Results after 1000 Games - Action Embedding

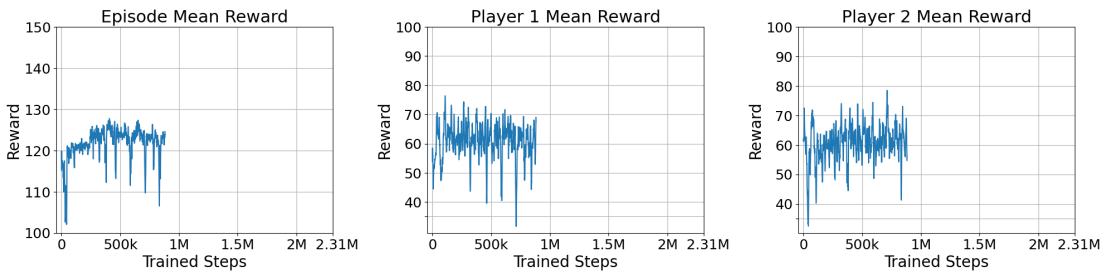


Figure 5.53 DQN Episode, Player 1 and Player 2 Average Scores - Action Embedding

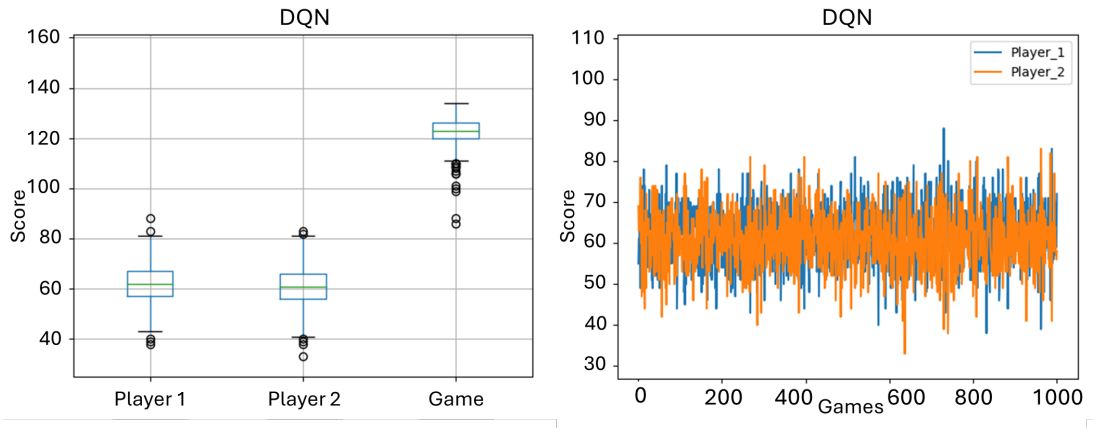


Figure 5.54 DQN - Quantitative Results after 1000 Games - Action Embedding

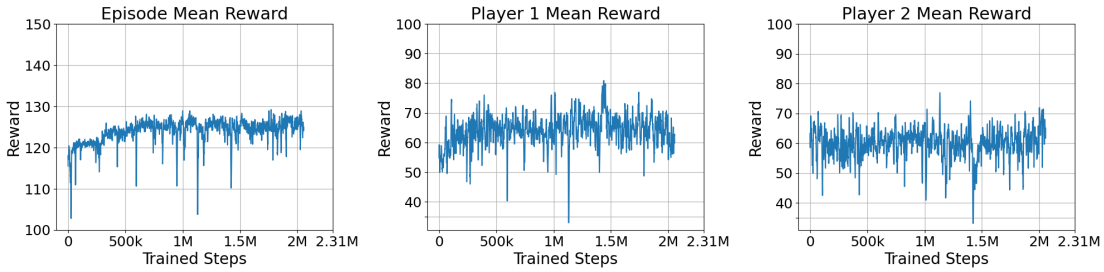


Figure 5.55 DDQN Episode, Player 1 and Player 2 Average Scores - Action Embedding

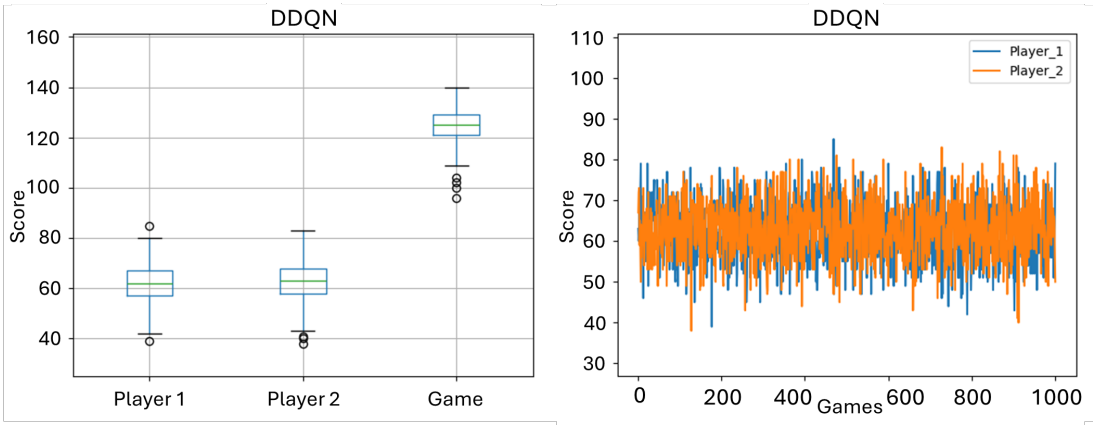


Figure 5.56 DDQN - Quantitative Results after 1000 Games - Action Embedding

Hierarchical RL with Centralised Critic

The results obtained from the last training iteration of all four models, when these were trained with the hierarchical RL with centralised critic technique, are presented in Table 5.8. From these results, it can be noted that all the algorithms obtained scores which match those achieved by human players. Moreover, it was also noted that all four models developed a stronger player 2 policy. The PPO and A2C models displayed a great gap in performance between their individual policies, whilst the DQN and DDQN models developed policies similar to each other. Moreover, Figures 5.57, 5.59, 5.61 and 5.63 display the change in the average individual and combined scores of the models during training. For the PPO algorithm, it is evident that the player 2 policy was superior throughout the entirety of the training. Up until around 100,000 steps, this policy was achieving very high rewards above 70 points. However, then the player 1 policy improved, which resulted in the player 2 policy's scores to decrease to above 65 points, which is still considered high. Moreover, the reward gap between both policies kept decreasing whilst the combined average score was increasing. Meanwhile, within the A2C model, the player 1 policy initially had a very substantial advantage over the other policy. Despite this, at around 600,000 steps the player 2 policy discovered a superior strategy, resulting in it achieving higher rewards than the player 1 policy, with less of a gap between both policies. Furthermore, both the DQN and DDQN models presented policies which were obtaining a larger range of scores throughout the training than the other models. Moreover, in both these models, the policies were very similar to each other, with the player 1 policy overall achieving a slightly higher range of rewards than the player 2 policy. The combined average rewards of both DQN and DDQN stabilised to around 120 almost instantaneously. Moreover, from the Figures 5.58, 5.60, 5.62 and 5.64, which display the rewards obtained when the policies were tested on 1000 random games, it can be seen that the achieved results of the PPO, A2C and DDQN algorithms correspond to those discussed above. These three algorithms all had a stronger player 2 policy, with a large gap in the rewards obtained by the individual PPO and A2C policies, whilst the policies of the DDQN model obtained scores similar to each other. However, despite the rewards from the final training iteration showing that the player 2 policy was stronger for DQN, the results observed from the games displayed that, the player 1 policy was superior with the performance of both policies being relatively similar. This was also observed from the training process of this model.

The results obtained within this implementation were compared to the results obtained from the increased action space experiment, where the models were trained with action masking only, as well as the action embedding experiment. This comparison was performed as the models within these three experiments were all

trained with the initial hyperparameters and therefore, it provides great insight on the performance of each algorithm with each technique.

When compared to the action masking technique, the PPO and A2C models of the HRL implementation achieved a minor decrease in the combined reward, whilst the policies became less balanced. There was a noticeable decrease in the player 1 policy's return and an increase in the scores achieved by the player 2 policy. Moreover, for the DQN and DDQN algorithms, both individual policies were returning lower rewards, which also caused a lower combined score. The reward gap between the individual policies remained very similar within both techniques.

Meanwhile, when the results were compared to those obtained by the action embedding technique, the PPO and DQN models presented lower combined and player 1 scores with higher player 2 scores. However, from the results obtained following the games played, the DQN had higher player 1 scores with lower player 2 and combined scores, than the respective action embedding model. Moreover, from the final training iterations, the A2C hierarchical model displayed higher player 1 and combined scores, with very similar player 2 scores, where the average reward had 1 point less, whilst the minimum and maximum rewards had 1 point more each. However, by comparing the results obtained from the simulated games, the HRL A2C player 2 policy displayed a lower average and minimum return, with the minimum combined score also being less than that obtained by the action masking A2C model. Furthermore, the DDQN HRL model returned lower individual and combined scores. On the other hand, the PPO, DQN and DDQN models had greater stability when trained with the HRL technique rather than the action masking and action embedding techniques. This was also the case for the A2C model, apart from the considerable shift that occurred within the scores obtained by the policies when the player 2 policy discovered a superior strategy.

Furthermore, given that this technique performs two separate steps in each player turn, one to select the high-level action and another to select the low level action, the number of in-game steps were double those performed by the models trained with the other techniques. Despite this, all the action masking and action embedding models had less than half the number of steps, except for the A2C action embedding model which had more than half. Nevertheless, when considering only half the amount obtained by each HRL model, the difference within the number of steps between techniques was not substantial. However, due to the double step for each player turn, the HRL models were trained for much less number of games and took much longer time to train, compared to the other techniques. Table 5.9 displays the percentage of the games trained with the HRL technique relative to the respective action masking and action embedding models. Similarly, the HRL models sampled much more steps than the respective action masking and action embedding models, except for the A2C action embedding model. The greatest difference in sampled steps

was observed within the DQN and DDQN algorithms, with an additional 253,000 and 758,000 from the respective action masking models, as well an additional 587,000 and 909,000 steps from the respective action embedding models. Despite this, the number of steps trained by the DQN and DDQN algorithms with the HRL technique were less than those trained with the other techniques, with the percentages of trained steps, from the action masking and action embedding models, corresponding to the respective percentages of games trained displayed in Table 5.9. Furthermore, when comparing the training time of the models, relative to the number of steps sampled and trained, the HRL models took significantly longer training times for all the models, expect for the PPO action embedding model.

Table 5.8 Quantitative Results from the Last Training Iteration - Hierarchical RL with Centralised Critic

Metrics	Algorithms			
	PPO	A2C	DQN	DDQN
Number of Games Trained	19,892	10,133	17,707	36,208
Episode Mean Reward	121.77	122.19	119.9	120.46
Episode Max Reward	132	133	127	131
Episode Min Reward	100	104	94	102
Player 1 Mean Reward	55.29	54.06	59.02	59.5
Player 1 Max Reward	76	81	78	78
Player 1 Min Reward	31	34	39	36
Player 2 Mean Reward	66.48	68.13	60.88	60.96
Player 2 Max Reward	86	90	80	77
Player 2 Min Reward	39	43	40	33
Episode Mean Length	111.36	112.5	118.22	120.1

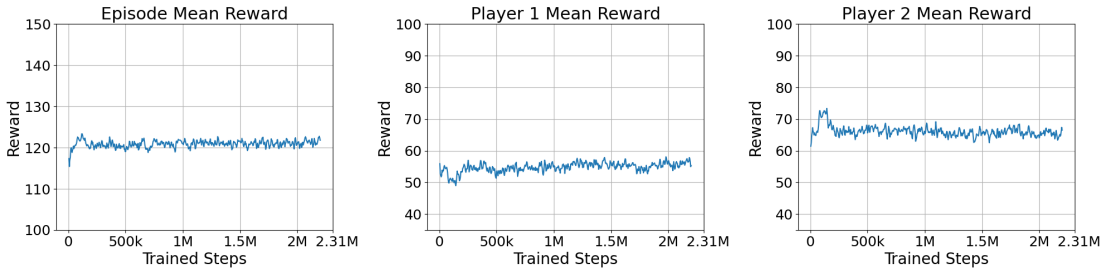


Figure 5.57 PPO Episode, Player 1 and Player 2 Average Scores - Hierarchical RL with Centralised Critic

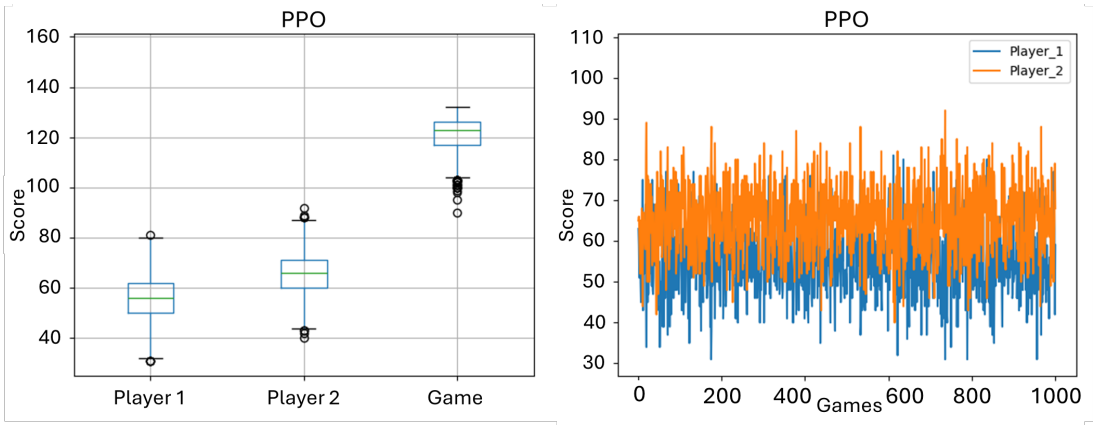


Figure 5.58 PPO - Quantitative Results after 1000 Games - Hierarchical RL with Centralised Critic

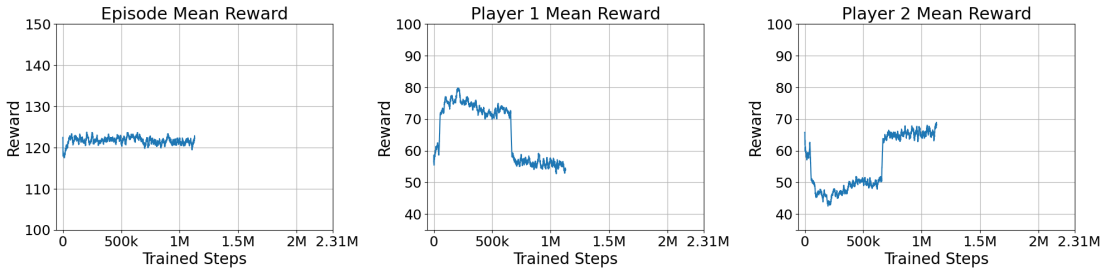


Figure 5.59 A2C Episode, Player 1 and Player 2 Average Scores - Hierarchical RL with Centralised Critic

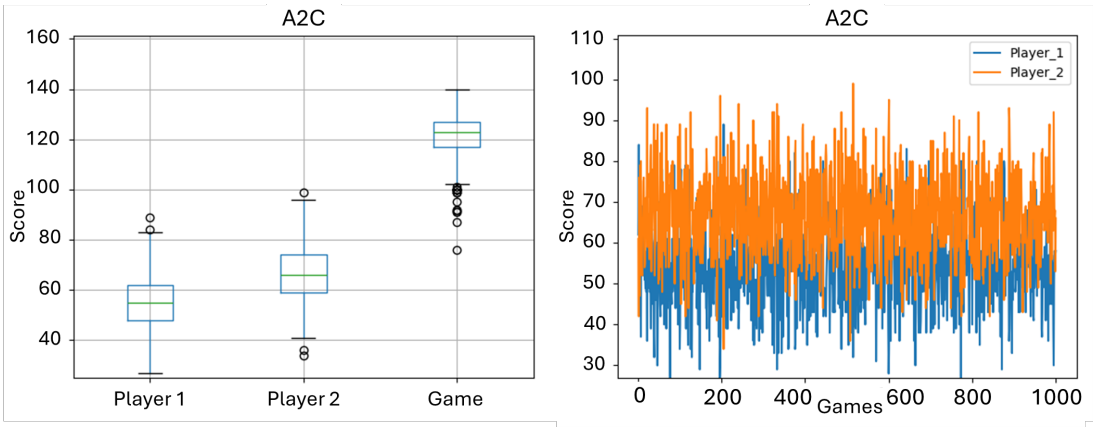


Figure 5.60 A2C - Quantitative Results after 1000 Games - Hierarchical RL with Centralised Critic

Table 5.9 Percentage of the Games Trained by the Hierarchical Models from the Action Masking and Embedding Models

Algorithm	Action Masking	Action Embedding
PPO	54%	55%
A2C	61%	37%
DQN	55%	64%
DDQN	56%	56%

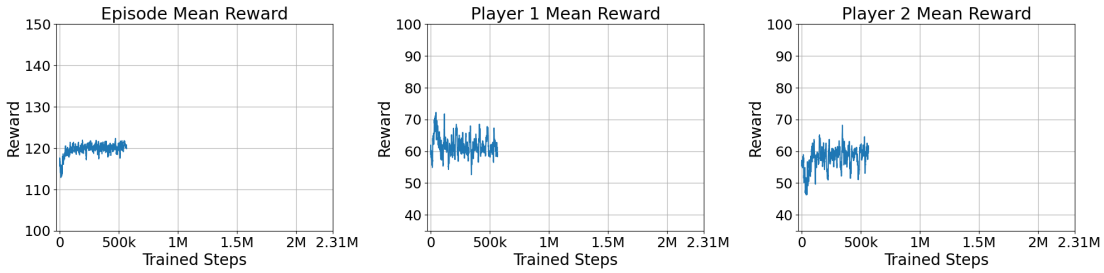


Figure 5.61 DQN Episode, Player 1 and Player 2 Average Scores - Hierarchical RL with Centralised Critic

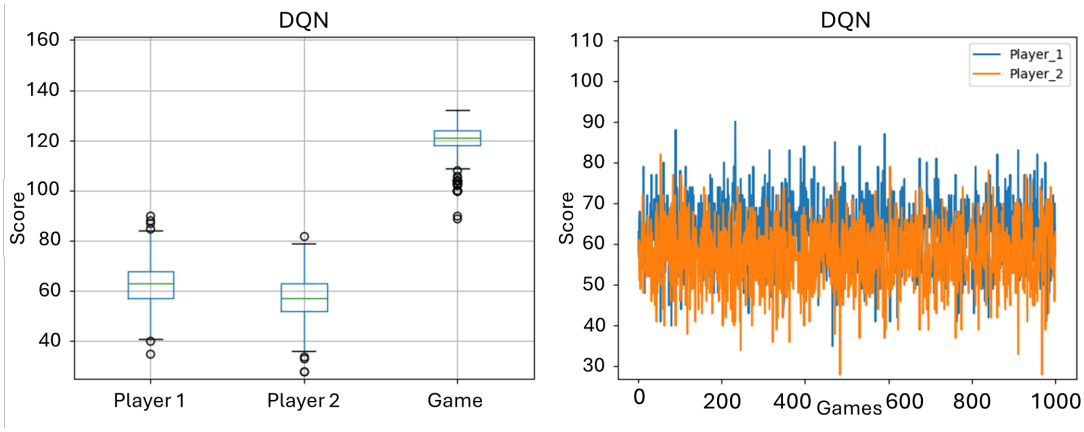


Figure 5.62 DQN - Quantitative Results after 1000 Games - Hierarchical RL with Centralised Critic

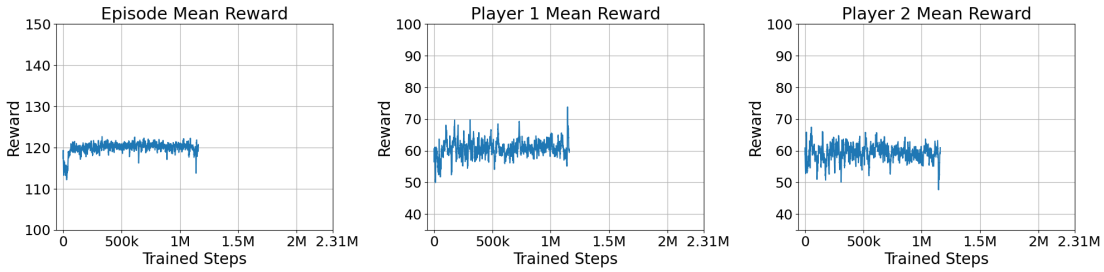


Figure 5.63 DDQN Episode, Player 1 and Player 2 Average Scores - Hierarchical RL with Centralised Critic

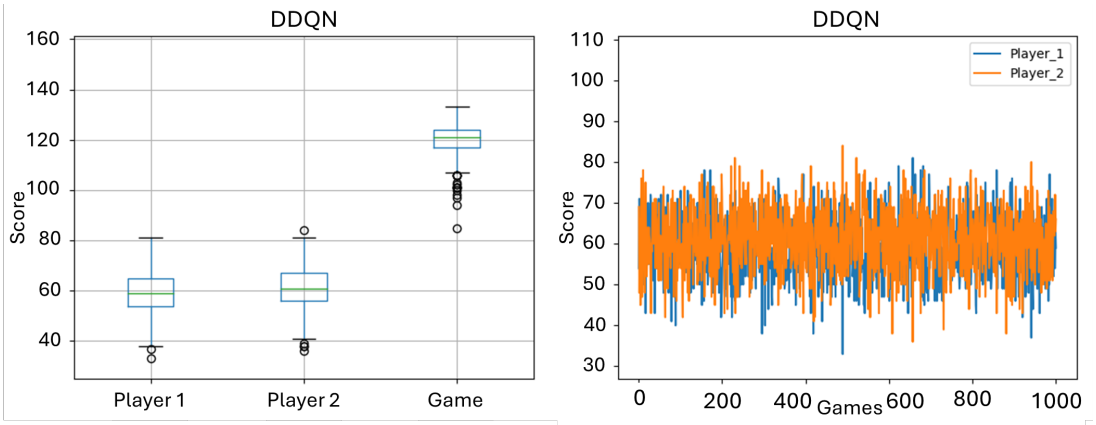


Figure 5.64 DDQN - Quantitative Results after 1000 Games - Hierarchical RL with Centralised Critic

5.2.2 Action Selection Analysis

In addition to the quantitative evaluation, the performance of each policy was also assessed by analysing the agent's action selection. This was performed by evaluating whether the selected actions seemed to be ideal when taking into consideration the state of the game. Furthermore, to gain a deeper understanding of the strategies that human players adopt during gameplay, and be able to compare with the actions selected by the agents, the forums [5], [6] and [7] were discovered which contain discussions between avid players about their ideal strategies. Such strategies include:

- taking Camel cards when the opponent's hand is full, since if this is the case, the opponent was most likely wanting to sell in their round and therefore will be less likely to take any good cards should any become present in the marketplace
- selling low amounts of cards at the start of the game to get the first few tokens which contain a lot of points
- not keeping a lot of cards near the end of the game, since if the game ends, these cards will be wasted opportunities to get points
- paying attention to the opponent's actions and even taking unnecessary cards that the opponent might need to sell multiple cards at once for the bonus token points
- keeping low-values cards to trick the opponent into thinking that you want to sell but instead using the cards as an exchange for high-value cards

[5] https://www.reddit.com/r/boardgames/comments/10fbd9j/jaipur_any_strategy_or_all_luck_of_the_draw/
[6] https://www.reddit.com/r/boardgames/comments/2cb85d/jaipur_strategies/
[7] https://www.reddit.com/r/boardgames/comments/fcjk9c/is_there_any_strategy_to_jaipur/

- selling cards when the marketplace contains a lot of Camel cards to force the opponent to take the Camel cards to gain an opportunity to take any good-value cards from the replenished marketplace
- trying to end the game quickly when the opponent has less points and trying to drag out the game when the opponent has more points
- keeping track of how many cards remain with the use of the tokens since for each Goods type, there is one more card than the number of tokens
- not waiting to sell large amounts of high-value cards to not risk the opponent from taking the higher value tokens
- taking all but one tokens from a high-value Goods type token set for the opponent to waste a turn to sell two cards whilst only receiving the points for one

The evaluation was carried out by simulating 10 random games for each model and analysing the actions performed by both agents from the beginning till the end of each game. During these simulated games, the current player's gameplay observation, according to the experiment conducted, as well as the player's selected action, were displayed for each player turn. This analysis provided an interesting observation of the potential strategies that each policy learned to adopt and how these strategies differ when the policies encountered different scenarios.

The greatest difference within this evaluation was that the PPO and A2C algorithms were much more likely to trade multiple cards in a turn than the DQN and DDQN algorithms. Meanwhile the DQN and DDQN algorithms were opting to take individual cards from the marketplace rather than trading out other Goods cards or Camel cards. Despite this, all the agents, across all algorithms and techniques, were observed trading up to 5 cards at once.

Another observation was that most of the agents were often taking Camel cards throughout the game as it allowed them to be able to get sets of cards without trading out a large number of the Goods cards in their hand. However, the interest in Camel cards was especially evident nearing the end of the game, which shows that the players were aiming to get more Camel cards than their opponent to get the Camel token. Contributing further to this observation was that when trading out Camel cards, the players would almost always either keep at least half of their Camel cards, or keep the same number of, or more, Camel cards than the opponent can have in total, when taking into consideration the Camel cards in the opponent's herd and in the marketplace. This was also especially observed near the end of the game and less frequently observed at the start of the games, where the agents were often prioritising getting Goods cards rather than keeping Camel cards. Moreover, when the opponent

had most of the Camel cards available in the game within their herd, especially if the game was nearing the end, the agents were more likely to trade more of their Camel cards, given that the likelihood of them being able to get a larger number of Camel cards was very low. In this case, the agents were observed getting and selling as many cards as possible prior to the game ending to try to collect more points, given that they will lose the Camel token. However, throughout the game, the agents were observed prioritising taking a number of high value cards instead of having more Camel cards.

All the agents, especially those of the PPO and A2C algorithms, were often prioritising taking the high-value cards immediately as soon as they appear within the marketplace. Moreover, all the agents were also noted selling the cards which would return the highest valued tokens at the start of the game, given that the token sets are sorted in descending order, and therefore the first few tokens are worth a lot of points.

Apart from this, the majority of the PPO, A2C and DDQN agents were noted selling up to 3 high value cards and 5 low value cards, particularly of type Leather. This is an interesting and smart decision given that there are only 6 high value cards within the game and it is unlikely for a player to manage to obtain more than 3 at once, especially if the player aims to take the first two tokens. Meanwhile, there are a lot more low value cards, especially of the Leather type, and therefore, the agent has a much greater chance of being able to collect a large number of such cards. Moreover, since the Leather tokens are low in value, the player can risk waiting to collect more cards as the Bonus tokens are worth more than the Leather tokens.

This observation differed within the PPO algorithm of the increased action space environment, as these agents were only noticed selling up to 3 low value cards. These agents were frequently selling individual Leather cards, to delay performing another action, especially instead of taking a large number of Camel cards, in order to prevent from repopulating the marketplace and potentially providing the opponent with a good card. Moreover, these agents were also noted selling 2 high value cards even when they had 3 in hand. This was potentially performed to get the highest valued tokens, of the respective set, as well as to have a higher chance of being able to get more tokens, since 2 cards need to be sold at once for the high value card types. This strategy was also being performed by the A2C algorithm of the same environment. However, this algorithm was not applying this strategy to Silver cards, since the Silver tokens are all of the same value. In fact, this algorithm was noted collecting and selling 4 Silver cards at once. Moreover, both the A2C, DQN and DDQN algorithms, of the increased action space environment, were observed selling up to 4 low value cards.

Similarly to the above, the policies of the DQN algorithm were also observed frequently selling only 2 high value cards, across all implementations except for the initial implementation. Within the increased agent observation and full opponent observation environment, such an action was noticed even when the agent had 3 or 4

cards of the same type at once. However, within the initial implementation, this observation differed drastically as the DQN agent was observed collecting and selling 7 Leather cards at once.

From this action selection analysis, all the policies, across all the different experiments, were observed performing smart decisions that can be linked to strategies which human players choose to adopt during gameplay. For each algorithm of each of the conducted experiments, Appendix F displays a number of interesting actions that the agents were observed performing.

5.3 Summary

This chapter delved into how the implementation of each experiment was tested, as well as provided a detailed explanation of the evaluation process. Moreover, the quantitative results were also presented, along with an accompanying analysis for each experiment. The presented results displayed that all the techniques and algorithms developed policies which obtained great rewards corresponding to those obtained by human players. Moreover, the results also displayed that the hyperparameter tuning as well as policy cloning technique were effective. Furthermore, it was concluded that the algorithms were capable of successfully learning an optimal policy, which allowed them to obtain high scores whilst playing the game of Jaipur with partial observability, as when the observable and full opponent observations were provided, only minor changes were noted within the performance of the algorithms. Finally, between the action masking, action embedding and hierarchical RL with centralised critic techniques, the action masking technique resulted in the best performance across all algorithms. The A2C, DQN and DDQN algorithms obtained slightly lower rewards with the action embedding technique. Meanwhile, the rewards obtained by the PPO algorithm with this technique were the highest among all three techniques, however this came at the detriment of the longest training times for this algorithm, almost double the time taken by the action masking technique. Moreover, the computational complexities for the A2C algorithms were also increased, however the DQN and DDQN algorithms benefitted from the action embedding technique, as they were taking less time to sample and train steps. Meanwhile, the hierarchical technique provided greater stability for the algorithms at the cost of lower rewards and much longer training times.

6 Conclusion

This final chapter will revisit the aims and objectives and highlight the achievements made throughout this study. Moreover, a critique of this work will be provided, together with potential opportunities for future work.

6.1 Revisiting the Aims and Objectives

The aim of this work was to successfully and efficiently develop intelligent AI agents on POSGs. Specifically, this study consisted of the implementation of various RL techniques and algorithms on the game Jaipur, to explore how the challenges that are brought about by the characteristics of such games can be mitigated. To ensure that this aim was addressed, the objectives mentioned in Section 1.3, will be revisited.

Objective 1: Develop the Initial Environment with Action Masking

The first objective consisted of the implementation of the Jaipur game environment along with a RL environment with the appropriate action and observation spaces, as well as the integration of a RL library for the application of different RL algorithms with the action masking technique. For this objective to be completed, the Jaipur game environment was developed in Python and included all the necessary functions, which were thoroughly tested, to ensure that the implementation contained accurate game logic which complied with all the rules of Jaipur. Afterwards, the PettingZoo library was used for the RL environment to be developed. To properly integrate the Jaipur environment with the RL environment, the necessary action and observation spaces were designed to accurately represent the state of the game to the agent. This consisted of an action space which included all the possible actions that a player can choose from during gameplay. Moreover, the observation space was designed to include all the player's gameplay information such as their own cards, their number of points, their own and their opponent's number of Camel cards, the marketplace cards and the remaining tokens. Furthermore, the RL environment was designed to consist of two agents, with their own action and observation spaces, to develop independent policies. This was performed to allow the agents to interact with the environment separately in turns and to accurately reflect the competitive element of Jaipur. In order to be able to apply and test different RL algorithms on the RL environment, Ray's RLlib library was integrated and the PPO, A2C, DQN and DDQN algorithms were applied. However, given the large number of possible actions where most of them are invalid in a given state, the action masking technique was applied by making use of RLlib's action masking model. Apart from this, both the game and the RL environments were modified for the action mask to be created in each player's turn,

based on which actions were valid or not, and for it to be included within the player's observation space. This technique proved to be very beneficial as by masking out the invalid actions, the time taken by the agents to learn was reduced greatly.

Objective 2: Experiment with the Game Features and Hyper-Parameter Tuning

The second objective focused on conducting several experiments in which certain game features and hyperparameters were altered to analyse their effect on the training and performance of the policies. The first experiment consisted of making use of the initial implementation and increasing the action space of the game. This was achieved by further splitting the actions into more detailed actions to allow the agents to have the utmost control over how many cards they take, trade or sell. Following the addition of these actions, the game environment was updated along with the action and observation spaces. This implementation was also crucial for the remaining experiments to be conducted on and tested against. The second experiment consisted of performing hyperparameter tuning to try to optimise each algorithm's performance. This was achieved by making use of Ray Tune's ready-made PopulationBasedTraining schedular to adjust the hyperparameters during training. The hyperparameters which resulted in the best trained model were used for most of the remainder of the experiments. The third experiment consisted of allowing each agent to keep track of the opponent's observable information. This was accomplished by updating the game environment to allow the player to keep track of the which cards the opponent takes from the marketplace and sells, as well as an estimation of the opponent's number of points. Moreover, the observation space was also updated for the added information to be provided to the agent. Whilst this experiment focused on providing the agent with further information about the environment, the fourth experiment was conducted to provide the agent with the full opponent observation. These two experiments, along with the experiment that only contained what the player can observe in each state, allowed for an analysis to be carried out on how partial observability affected the RL process on the Jaipur environment. To implement the fourth experiment, the observation space and the necessary functions within the RL environments were updated to keep track of all the opponent information.

Objective 3: Experiment with Different Techniques

Finally, the third objective concentrated on experimenting with different techniques which can handle the challenges brought about by the characteristics in POSGs, mainly large discrete action spaces. Apart from the action masking technique, three other techniques were applied to the implementation, namely the policy cloning, action embedding and HRL with a centralised critic techniques. The policy cloning technique was applied by stopping the models after various amounts of training steps, comparing the performance of the policies to identify the best performing policy,

creating a copy of it for each policy and finishing the training. Meanwhile, the action embedding technique was applied using RLlib's ParametricActionsModelThatLearnsEmbeddings model. When action embedding was applied without the use of action masking, the agents were unable to learn and therefore, the mentioned model was modified to make use of the action mask. Finally, the HRL with centralised critic technique was applied. The actions were all split into four main action types which were set as the high level agents, whilst the more detailed actions were set as the low level agents. Moreover, the use of a centralised critic was crucial, given that all the actions affect each other within the game. However, given that one particular action type contained the majority of the low level actions, the use of action masking was still necessary. With this in mind, to simplify the implementation, instead of having separate policies, one to select the high level action and four to select the low level action, for each player, with a centralised critic, only one policy was used for each player and the action mask was not only used to mask out the invalid actions, but also the actions of the other agents.

6.2 Critique and Limitations

While the study provided great insights on the application of RL techniques on POSGs, some constraints and limitations were encountered throughout the implementation.

Given the significantly long training times taken by each model to be trained, as well as due to the lack of more powerful hardware, restricted hyperparameter tuning was performed for each algorithm and the architecture of the policy network of each algorithm was left as the default configuration. Moreover, the hyperparameter tuning performed was also limited to one particular implementation, which served as a basis for the other experiments. These may have impacted the optimal performance achieved by the algorithms with the different techniques applied. Similarly, given the increased computational requirements of the HRL with centralised critic technique, the models using this technique were not trained for as many iterations as the models with the other techniques. This may have reduced the performance of the models and provided slight unfairness in the quantitative comparison of the different techniques.

Furthermore, due to the large number of experiments, whilst all the policies were individually evaluated quantitatively based on the scores achieved, the trained policies of the different implementations were not played against each other. Moreover, due to the same reasoning, the action selecting analysis performed in this work was limited to a high-level analysis of all the policies, without providing an in-depth qualitative evaluation of each agent's reasoning across different game states. These two limitations could have offered deeper insights into the strategic differences

between the trained agents, stemming from the different techniques applied. Another limitation of this study is that despite Jaipur's interesting and challenging characteristics, the experiments carried out were all limited to this game, and therefore, the findings lack generalisability beyond the game Jaipur.

6.3 Future Work

Whilst the work carried out achieves the overarching goals of this study, several other promising avenues can be explored to expand and deepen the research of applying AI on POSGs. The main improvement of this study could be the application of the techniques and algorithms explored in this work on other POSGs, to analyse how well these findings generalise across different environments. Moreover, another interesting addition could be the application of other techniques, such as imitation learning, in which the agents learn from actions selected by experts, to compare the results with the techniques explored in this work. Similarly, the implementation of algorithms that make use of RNNs can also be explored to analyse their performance on Jaipur.

Moreover, considering the limitations of this study, this work could potentially be improved by performing further hyperparameter turning on all the different implementations separately and experimenting with different policy network configurations for each algorithm. Apart from these, it would be beneficial to play the policies of the different algorithms and techniques against each other, for a deeper analysis on how they differ in performance. Moreover, another idea would be to add a Graphical User Interface to the game, upload it online, and allow human players to play against different policies.

6.4 Final Remarks

This work aimed to investigate how AI agents can be developed to effectively and efficiently operate in POSGs. This was accomplished by implementing the game Jaipur for it to be used as the experimental environment, on which multiple algorithms and techniques were applied and evaluated. Such algorithms included PPO, A2C, DQN and DDQN, whilst the techniques applied consisted of action masking, action embedding, HRL with centralised critic and policy cloning. Moreover, within this work, hyperparameter tuning was also performed and a set of two additional experiments were carried out, where the agents were provided with further state information, to analyse the effect of partial observability on the RL process.

The insights gained from this study display that the action masking technique is not only very beneficial but necessary for the agents to learn efficiently in

environments characterised by a large discrete action space where most actions are invalid in each state. Nonetheless, both the action embedding and the HRL with centralised critic techniques provided good results when applied with the action masking technique, however these techniques were more computationally heavy across some algorithms. Specifically, the action embedding technique provided the longest training times for the PPO algorithm, where the increase was drastic compared to the other techniques. Nonetheless, the results obtained by this model were higher than those obtained by PPO with the other techniques. The action embedding technique also caused an increase in the training time of the A2C algorithm, however, it presented shorter training times for the DQN and DDQN algorithms. Meanwhile, the HRL with centralised critic technique provided the longest training times across the A2C, DQN and DDQN algorithms, with the PPO algorithm's training time still being much longer than the time taken for the algorithm to be trained with the action masking technique. Moreover, the policy cloning technique reduced the overall training time taken for the policies to reach stable performance and the hyperparameter tuning increased the performance of the algorithms. By increasing the agents' partial observations to the observable and full opponent information, it was noted that the agents were able to converge to a dominant strategy which allowed them to efficiently play the game and obtain high scores, irrespective of their knowledge on their opponent's hand, as the difference in performance across all three experiments was marginal.

The methodology and results presented in this work provide a solid foundation for future research, not only in the field of AI on POSGs, but also in the field of applying AI in real-world settings which share similar features and levels of complexity, uncertainty, and observability.

References

[1] C. Cutajar and J. Bajada, "Mastering the card game of jaipur through zero-knowledge self-play reinforcement learning and action masks," in *AIxIA 2023 – Advances in Artificial Intelligence*, R. Basili, D. Lembo, C. Limongelli, and A. Orlandini, Eds., Cham: Springer Nature Switzerland, 2023, pp. 231–244, ISBN: 978-3-031-47546-7. DOI: 10.1007/978-3-031-47546-7_16. [Online]. Available: https://link.springer.com/10.1007/978-3-031-47546-7_16.

[2] Y. Bengio, I. Goodfellow, and A. Courville, *Deep Learning*. MIT Press, Oct. 2016. [Online]. Available: http://www.deeplearningbook.org.

[3] Z.-H. Zhou and S. Liu, *Machine learning*, 1st Edition 2021. Singapore Springer, 2021.

[4] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd. MIT Press, 2020, p. 526, ISBN: 9780262039246.

[5] D. Silver et al., "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, Dec. 2017.

[6] D. Silver et al., "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018. DOI: 10.1126/science.aar6404.

[7] Y. Liu, J. Zheng, and F. Chang, "Learning and planning in partially observable environments without prior domain knowledge," *International Journal of Approximate Reasoning*, vol. 142, pp. 147–160, Mar. 2022, ISSN: 0888613X. DOI: 10.1016/j.ijar.2021.12.004.

[8] D. Zha et al., "Douzero: Mastering doudizhu with self-play deep reinforcement learning," in *International Conference on Machine Learning*, PMLR, 2021, pp. 12 333–12 344.

[9] S. Liu, J. Cao, Y. Wang, W. Chen, and Y. Liu, "Self-play reinforcement learning with comprehensive critic in computer games," *Neurocomputing*, vol. 449, pp. 207–213, Aug. 2021, ISSN: 18728286. DOI: 10.1016/j.neucom.2021.04.006.

[10] N. Justesen, L. M. Uth, C. Jakobsen, P. D. Moore, J. Togelius, and S. Risi, "Blood bowl: A new board game challenge and competition for ai," in *2019 IEEE Conference on Games (CoG)*, 2019, pp. 1–8. DOI: 10.1109/CIG.2019.8848063.

[11] Z. Yao et al., "Towards modern card games with large-scale action spaces through action representation," in *2022 IEEE Conference on Games (CoG)*, IEEE, 2022, pp. 576–579. DOI: 10.1109/CoG51982.2022.9893589.

[12] K. Fujita, “Alphadda: Strategies for adjusting the playing strength of a fully trained alphazero system to a suitable human training partner,” *PeerJ Computer Science*, vol. 8, e1123, 2022.

[13] Q.-Y. Yin et al., “Ai in human-computer gaming: Techniques, challenges and opportunities,” *Machine Intelligence Research*, pp. 1–19, 2023.

[14] G. Dulac-Arnold et al., “Deep reinforcement learning in large discrete action spaces,” *arXiv preprint arXiv:1512.07679*, Dec. 2015. [Online]. Available: http://arxiv.org/abs/1512.07679.

[15] V. Mnih et al., “Playing atari with deep reinforcement learning,” *arXiv preprint arXiv:1312.5602*, Dec. 2013. [Online]. Available: http://arxiv.org/abs/1312.5602.

[16] Y. Lecun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, pp. 436–444, 7553 May 2015, ISSN: 14764687. DOI: 10.1038/nature14539.

[17] A. Plaat, *Deep Reinforcement Learning*. Springer Nature Singapore, 2022. DOI: 10.1007/978-981-19-0638-1.

[18] D. Karunakaran, S. Worrall, and E. Nebot, “Efficient statistical validation with edge cases to evaluate highly automated vehicles,” in *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, IEEE, 2020, pp. 1–8.

[19] K. Bardis, “Reinforcement in cooperative games deep learning approaches. ”[Online]. Available: https://dspace.lib.ntua.gr/xmlui/bitstream/handle/123456789/55942/Dissertation_deliverable.pdf?sequence=1&isAllowed=y.

[20] J. Hu and M. P. Wellman, “Multiagent reinforcement learning: Theoretical framework and an algorithm. ”[Online]. Available: http://people.ee.duke.edu/~lcarin/emag/seminar_presentations/Multiagent_Learning_Hu_ICML98.pdf.

[21] J. J. Tai, J. Wong, M. Innocente, N. Horri, J. Brusey, and S. K. Phang, “Pyflyt – uav simulation environments for reinforcement learning research,” Apr. 2023. [Online]. Available: http://arxiv.org/abs/2304.01305.

[22] G. Efstathiadis, P. Emedom-Nnamdi, J.-P. Onnela, J. Lu, and A. Kolbeinsson, “Stasis: Reinforcement learning simulators for human-centric real-world environments. ”[Online]. Available: https://openreview.net/pdf?id=LsEd-S3ofyW.

[23] J. D. Merrick et al., “Corl: Environment creation and management focused on system integration,” Mar. 2023. DOI: 10.48550/arXiv.2303.02182.

[24] J. K. Terry et al., “Pettingzoo: Gym for multi-agent reinforcement learning,” Sep. 2020. [Online]. Available: http://arxiv.org/abs/2009.14471.

[25] K. Horák and B. Bošanský, “Solving partially observable stochastic games with public observations,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, pp. 2029–2036, Jul. 2019. DOI: 10.1609/aaai.v33i01.33012029. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/4032.

[26] C. D. Bergman and K. Hakhamaneshi, “Hands-on reinforcement learning for recommender systems-from bandits to slateq to offline rl with ray rllib,” Association for Computing Machinery, Inc, Sep. 2022, pp. 700–701, ISBN: 9781450392785. DOI: 10.1145/3523227.3547370.

[27] E. Liang et al., “Rllib: Abstractions for distributed reinforcement learning,” *International Conference on Machine Learning*, pp. 3053–3062, Jul. 2018.

[28] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, “Stable-baselines3: Reliable reinforcement learning implementations,” *The Journal of Machine Learning Research*, vol. 22, no. 1, Jan. 2021, ISSN: 1532-4435.

[29] E. Liang et al., “Ray rllib: A composable and scalable reinforcement learning library,” in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS 2017, 2017.

[30] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, “Tune: A research platform for distributed model selection and training,” *arXiv preprint arXiv.1807.05118*, Jul. 2018. DOI: 10.48550/arXiv.1807.05118.

[31] M. Jaderberg et al., “Population based training of neural networks,” *arXiv preprint arXiv:1711.09846*, Nov. 2017.

[32] F. Ming, F. Gao, K. Liu, and C. Zhao, “Cooperative modular reinforcement learning for large discrete action space problem,” *Neural Networks*, vol. 161, pp. 281–296, Apr. 2023, ISSN: 18792782. DOI: 10.1016/j.neunet.2023.01.046.

[33] A. Kanervisto, C. Scheller, and V. Hautamäki, “Action space shaping in deep reinforcement learning,” in *2020 IEEE Conference on Games (CoG)*, IEEE, 2020, pp. 479–486. DOI: 10.1109/CoG47356.2020.9231687.

[34] T. Zahavy, M. Haroush, N. Merlis, D. J. Mankowitz, and S. Mannor, “Learn what not to learn: Action elimination with deep reinforcement learning,” in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS’18, Montréal, Canada: Curran Associates Inc., 2018, pp. 3566–3577.

[35] C. Y. Tang, C. H. Liu, W. K. Chen, and S. D. You, "Implementing action mask in proximal policy optimization (PPO) algorithm," *ICT Express*, vol. 6, pp. 200–203, 3 Sep. 2020, ISSN: 24059595. DOI: 10.1016/j.icte.2020.05.003.

[36] J. Liu, P. Hou, L. Mu, Y. Yu, and C. Huang, "Elements of effective deep reinforcement learning towards tactical driving decision making," *arXiv preprint arXiv:1802.00332*, Feb. 2018.

[37] D. Ye et al., "Mastering complex control in MOBA games with deep reinforcement learning," in *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI-20)*, 2020, pp. 6672–6679.

[38] Z. Zhang et al., "Hierarchical reinforcement learning for multi-agent moba game," *arXiv preprint arXiv:1901.08004*, 2019. [Online]. Available: https://arxiv.org/abs/1901.08004.

[39] Y. Guo, Q. Zhang, J. Wang, and S. Liu, "Hierarchical reinforcement learning-based policy switching towards multi-scenarios autonomous driving," in *2021 International Joint Conference on Neural Networks (IJCNN)*, 2021, pp. 1–8. DOI: 10.1109/IJCNN52387.2021.9534349.

[40] S. Pateria, B. Subagdja, A.-h. Tan, and C. Quek, "Hierarchical reinforcement learning: A comprehensive survey," *ACM Computing Surveys*, vol. 54, no. 5, Jun. 2021, ISSN: 0360-0300. DOI: 10.1145/3453160. [Online]. Available: https://doi.org/10.1145/3453160.

[41] M. Hutsebaut-Buysse, K. Mets, and S. Latré, "Hierarchical reinforcement learning: A survey and open research challenges," *Machine Learning and Knowledge Extraction*, vol. 4, no. 1, pp. 172–221, 2022, ISSN: 2504-4990. DOI: 10.3390/make4010009. [Online]. Available: https://www.mdpi.com/2504-4990/4/1/9.

[42] E. Gilmour, N. Plotkin, and L. N. Smith, "An approach to partial observability in games: Learning to both act and observe," in *2021 IEEE Conference on Games (CoG)*, 2021, pp. 01–05. DOI: 10.1109/CoG52621.2021.9619004.

[43] S. Huang, A. Kanervisto, A. Raffin, W. Wang, S. Ontañón, and R. F. J. Dossa, "A2c is a special case of ppo," *ArXiv*, May 2022. DOI: 10.48550/arXiv.2205.09123.

[44] C. Yu et al., "The surprising effectiveness of PPO in cooperative multi-agent games," in *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2022. [Online]. Available: https://openreview.net/forum?id=YVXaxB6L2Pl.

[45] V. Mnih et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, 7540 Feb. 2015, ISSN: 14764687. DOI: 10.1038/nature14236.

[46] I. Ghory, “Reinforcement learning in board games,” Department of Computer Science, University of Bristol, Technical Report 105, 2004. [Online]. Available: https://www.researchgate.net/publication/2911513_Reinforcement_Learning_in_Board_Games.

[47] W. Konen, “Reinforcement learning for board games: The temporal difference algorithm,” Research Center CIOP (Computational Intelligence, Optimization and Data Mining), TH Köln–Cologne University of Applied Sciences, Technical Report, 2015. DOI: 10.13140/RG.2.1.1965.2329.

[48] M. A. Wiering, J. P. Patist, and H. Mannen, “Learning to play board games using temporal difference methods,” Utrecht University, Technical Report UU–CS–2005–048, 2005.

[49] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, “Dueling network architectures for deep reinforcement learning,” Nov. 2015. [Online]. Available: http://arxiv.org/abs/1511.06581.

[50] H. van Hasselt, A. Guez, and D. Silver, “Deep reinforcement learning with double q-learning,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, no. 1, Mar. 2016. DOI: 10.1609/aaai.v30i1.10295. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/10295.

[51] C. J. Watkins and P. Dayan, “Q-learning,” *Machine Learning*, vol. 8, pp. 279–292, 1992.

[52] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction Second edition, in progress*. 2015. [Online]. Available: https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf.

[53] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” Jul. 2017. [Online]. Available: http://arxiv.org/abs/1707.06347.

[54] Y. Wu, E. Mansimov, R. B. Grosse, S. Liao, and J. Ba, “Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation,” *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[55] M. Sewak, *Deep Reinforcement Learning : Frontiers of Artificial Intelligence*. Springer Singapore, 2019. [Online]. Available: https://doi.org/10.1007/978-981-13-8285-7.

[56] V. Mnih et al., “Asynchronous methods for deep reinforcement learning,” in *Proceedings of The 33rd International Conference on Machine Learning*, M. F. Balcan and K. Q. Weinberger, Eds., ser. Proceedings of Machine Learning Research, vol. 48, New York, New York, USA: PMLR, Jun. 2016, pp. 1928–1937. [Online]. Available: https://proceedings.mlr.press/v48/mniha16.html.

[57] C. D'Eramo, D. Tateo, A. Bonarini, M. Restelli, and J. Peters, "Mushroomrl: Simplifying reinforcement learning research," *Journal of Machine Learning Research*, vol. 22, no. 1, Jan. 2021, ISSN: 1532-4435.

Appendix A Initial Hyper-Parameters

A.1 PPO

The models using the PPO algorithm were trained with the hyperparameter values displayed in Table A.1.

Table A.1 PPO Hyperparameters

Hyperparameter	Value
lr_schedule	None
use_critic	True
use_gae	True
lambda	1.0
kl_coeff	0.0
sgd_minibatch_size	128
num_sgd_iter	30
shuffle_sequences	True
vf_loss_coeff	1.0
entropy_coeff	0.0
entropy_coeff_schedule	None
clip_param	0.3
vf_clip_param	10.0
grad_clip	None
kl_target	0.01
vf_share_layers	-1

A.2 A2C

The models using the A2C algorithm were trained with the hyperparameter values displayed in Table A.2.

Table A.2 A2C Hyperparameters

Hyperparameter	Value
use_critic	True
use_gae	True
grad_clip	30.0
lr_schedule	None
vf_loss_coeff	0.5
entropy_coeff	0.01
entropy_coeff_schedule	None
microbatch_size	None
lambda	1.0

A.3 DQN

The models using the DQN algorithm were trained with the hyperparameter values displayed in Table A.3.

Table A.3 DQN Hyperparameters

Hyperparameter	Value
lr_schedule	None
adam_epsilon	1e-08
grad_clip	40
tau	1.0
num_atoms	1
v_min	-10.0
v_max	10.0
noisy	False
sigma	0.5
dueling	False
hiddens	[]
double_q	False
n_step	1
before_learn_on_batch	None
training_intensity	None
td_error_loss_fn	'huber'
categorical_distribution_temperature	1.0
replay_buffer_config 'type'	MultiAgentPrioritizedReplayBuffer
replay_buffer_config 'prioritized_replay'	-1
replay_buffer_config 'capacity'	1000
replay_buffer_config 'prioritized_replay_alpha'	0.6
replay_buffer_config 'prioritized_replay_beta'	0.4
replay_buffer_config 'prioritized_replay_eps'	1e-06
replay_buffer_config 'replay_sequence_length'	1
replay_buffer_config 'worker_side_prioritization'	False

A.4 DDQN

The models using the DDQN algorithm were trained with the hyperparameter values displayed in Table A.4.

Table A.4 DDQN Hyperparameters

Hyperparameter	Value
lr_schedule	None
adam_epsilon	1e-08
grad_clip	40
tau	1.0
num_atoms	1
v_min	-10.0
v_max	10.0
noisy	False
sigma	0.5
dueling	False
hiddens	[]
double_q	True
n_step	1
before_learn_on_batch	None
training_intensity	None
td_error_loss_fn	'huber'
categorical_distribution_temperature	1.0
replay_buffer_config 'type'	MultiAgentPrioritizedReplayBuffer
replay_buffer_config 'prioritized_replay'	-1
replay_buffer_config 'capacity'	1000
replay_buffer_config 'prioritized_replay_alpha'	0.6
replay_buffer_config 'prioritized_replay_beta'	0.4
replay_buffer_config 'prioritized_replay_eps'	1e-06
replay_buffer_config 'replay_sequence_length'	1
replay_buffer_config 'worker_side_prioritization'	False

Appendix B Jaipur Environment Test Cases

Table B.1 Player Class Test Cases

Function	Input	Expected Output	Output
`__init__`	Player ID	Creates a player object and assigns its ID with the given value	The player object was created correctly
`reset`	N/A	Creates and initialises the player object's remaining variables accordingly	The variables were created and initialised

Table B.2 Card Class Test Cases

Function	Input	Expected Output	Output
`__init__`	N/A	Creates the game cards accordingly	The cards were created correctly
`shuffle`	N/A	Reorganises the order of the game cards randomly	The cards were shuffled
`deal`	N/A	Removes and returns the first card from the list to reflect taking the card from the deck	The first card from the list was removed and returned
`deal_camel`	N/A	Removes and returns a Camel card from the list	A Camel card was removed from the list and returned

Table B.3 Token Class Test Cases

Function	Input	Expected Output	Output
`__init__`	N/A	Creates the game tokens and sorts them according to the game rules	The tokens were created correctly

Table B.4 Jaipur Class Test Cases

Function	Input	Expected Output	Output
`__init__`	N/A	Calls the respective functions for the two player objects and trade dictionary to be created	The functions were called and the objects were created correctly
`create_trade_dict`	N/A	Creates the trade dictionary with all the possible trading actions	The dictionary was created correctly
`round`	N/A	Sets up the initial game environment correctly according to the game rules	The initial game environment was set up according to the rules
`take_1_good`	Selected Goods card currently in the marketplace and player's hand is not full	Removes the card from the marketplace, assigns it to the player's hand and refills the marketplace with a card from the deck	The card was taken out of the marketplace and assigned to the player's hand, whilst the marketplace was repopulated with a card from the deck
`take_1_good`	Selected Goods card not currently in the marketplace	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment
`take_1_good`	Camel card	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment
`take_1_good`	Player's hand is full	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment

Table B.5 Jaipur Class Test Cases

Function	Input	Expected Action	Action
`take_goods`	Trading correct combination of cards present in the player's hand/herd and marketplace, without exceeding the hand size	Trades the cards from the player's hand/herd to the marketplace	The cards were traded accordingly
`take_goods`	Trading correct combination of cards but exceeding the hand size	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment
`take_goods`	Selected cards not present in the player's hand/herd or marketplace	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment
`take_goods`	Trading cards of the same type between the player's hand and marketplace	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment
`take_goods`	Trading different number of cards between the player's hand/herd and marketplace	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment
`take_goods`	Trading for Camel card/s from the marketplace	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment

Table B.6 Jaipur Class Test Cases

Function	Input	Expected Action	Action
`take_camels`	At least one Camel card in the marketplace	Removes all the Camel cards from the marketplace, assigns them to the player's herd and refills the marketplace with cards from the deck	All the Camel cards were taken out of the marketplace and assigned to the player's herd, whilst the marketplace was repopulated with cards from the deck
`take_camels`	No Camel cards in the marketplace	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment
`sell_goods`	Selling correct number of Goods cards of the same type found in the player's hand	Calls the necessary functions to remove the cards from the player's hand, remove and obtain the corresponding tokens, including any bonus tokens when selling 3 or more cards, and assigns the points to the player	Called the necessary functions to remove the cards as well as return the correct tokens and assigned the player with the correct number of points
`sell_goods`	Selling 1 Diamond, Gold or Silver cards	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment
`sell_goods`	Selling different types of Goods cards	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment

Table B.7 Jaipur Class Test Cases

Function	Input	Expected Action	Action
`sell_goods`	Selling Goods cards not present in the player's hand	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment
`sell_goods`	Selling Camel cards	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment
`sell_card`	Goods card present in the player's hand	Removes the specified card from the player's hand and appends it to the discard pile	Removed the card from the player's hand and appended it to the discard pile
`sell_card`	Goods card not present in the player's hand	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment
`sell_cards`	Player has 1 or more cards of the specified low-value Goods type in their hand	Returns all the cards found in the player's hand of the specified Goods type	Returned all the cards found in the player's hand of the specified Goods type
`sell_cards`	Player does not have any cards of the specified low-value Goods type in their hand	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment
`sell_min2_cards`	Player has 2 or more cards of the specified high-value Goods type in their hand	Returns all the cards found in the player's hand of the specified Goods type	Returned all the cards found in the player's hand of the specified Goods type

Table B.8 Jaipur Class Test Cases

Function	Input	Expected Action	Action
`sell_min2_cards`	Player has less than 2 cards of the specified high-value Goods type in their hand	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment
`ret_token_amount`	Goods type	Returns the correct amount of tokens left for the provided card type	Returned the correct number of tokens left for the provided card type
`ret_token_amount`	Camel	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment
`ret_token_amount`	Incorrect card type	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment
`pop_token`	Goods type	Removes the next token of the provided card type and returns its value	Removed the next token of the provided card type and returned its value
`pop_token`	Camel	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment
`pop_token`	Incorrect card type	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment

Table B.9 Jaipur Class Test Cases

Function	Input	Expected Action	Action
`pop_bonus_token`	Integer greater or equal to 3	Removes the next token from the bonus token set corresponding to the given number and returns its value	Removed the next token from the correct bonus token set and returned its value
`pop_bonus_token`	Integer smaller than 3	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment
`check_token`	Goods type with empty token set	Appends the card type to the list containing the empty token sets	Appended the card type to the list
`check_token`	Goods type with remaining tokens	No action is performed	No action was performed
`check_token`	Camel	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment
`check_token`	Incorrect card type	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment
`options`	Integer within the range of actions	Calls the respective function to perform the requested action	Called the correct function
`options`	Integer outside of the range of actions	No action is performed	No action was performed

Table B.10 Jaipur Class Test Cases

Function	Input	Expected Action	Action
`get_masked_options`	Correct player ID	Creates the action mask by checking if each action is valid or invalid and sets the elements corresponding to the valid actions to 1 and those corresponding to the invalid actions to 0	Created the action mask which correctly reflected whether each action was valid or invalid
`get_masked_options`	Incorrect player ID	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment
`finished`	5 cards present in the marketplace and less than 3 Goods token sets have finished	Returns `False` to signify that the game has not ended	Returned `False`
`finished`	3 or more Goods token sets have finished	Returns `True` to signify that the game has ended	Returned `True`
`finished`	Less than 5 cards present in the marketplace	Returns `True` to signify that the game has ended	Returned `True`
`final`	Players have different number of Camel cards	Assigns the Camel token to the player with the most Camel cards and returns this player's ID	Assigned the Camel token to the player with the most Camel cards and returned the correct player ID
`final`	Players have same number of Camel cards	Does not assign the Camel token to any player and returns an empty string	Did not assign the Camel token to any player and returned an empty string

Appendix C Reinforcement Learning Environment Test Cases

Table C.1 Reinforcement Learning Environment Class Test Cases

Function	Input	Expected Action	Action
`env`	N/A	Wraps and returns the RL environment	Returned the wrapped RL environment
`__init__`	N/A	Creates and initialises the necessary variables to be used within the environment	Created and initialised the necessary variables correctly
`reset`	N/A	Resets the necessary variables	Resetted the necessary variables correctly
`observation_space`	Correct player ID	Returns the observation space of the specified agent	Returned the observation space of the specified agent
`observation_space`	Incorrect player ID	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment
`action_space`	Correct player ID	Returns the action space of the specified agent	Returned the action space of the specified agent
`action_space`	Incorrect player ID	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment

Table C.2 Reinforcement Learning Environment Class Test Cases

Function	Input	Expected Action	Action
`get_observation`	Correct player ID	Compiles and returns the specified player's gameplay observation	Returned the correct gameplay observation in the correct order
`get_observation`	Incorrect player ID	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment
`observe`	Correct player ID	Calls the necessary functions to obtain the specified player's gameplay observation and action mask and returns them in the correct structure	Returned the correct gameplay observation and action mask in the correct structure
`step`	Integer within the range of actions that reflects a valid action	Calls the necessary functions to apply the action within the game environment and updates the necessary variables	Called the necessary functions that applied the action to the game environment and the necessary variables were updated correctly
`step`	Integer within the range of actions that reflects an invalid action	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment
`step`	Integer not within the range of actions	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment

Table C.3 Reinforcement Learning Environment Class Test Cases

Function	Input	Expected Action	Action
`step`	Agents performed 1000 steps within the environment	Truncates all agents	Truncated both agents
`step`	Terminated or truncated agent/s	Breaks out of the function without performing any changes to the environment	Broke out of the function and no modifications were applied to the environment
`render`	N/A	No action is performed	No action was performed
`close`	N/A	No action is performed	No action was performed

Appendix D Updated Jaipur Environment Test Cases

Table D.1 Updated Jaipur Class Test Cases

Function	Input	Expected Action	Action
`sell_num_cards`	Valid Goods cards present in the player's hand	Collects and returns the cards selected to be sold	Collected and returned the cards selected to be sold
`sell_num_cards`	Cards not present in the player's hand	Returns an error message without performing any changes to the environment	An error message was displayed and no modifications were applied to the environment

Appendix E Graphical Representation of Results

E.1 Initial Implementation with Action Masking - PPO

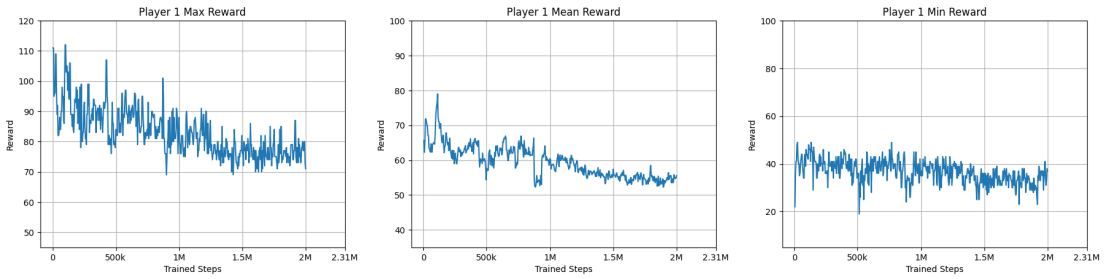


Figure E.1 PPO Player 1 Max, Mean and Min Scores

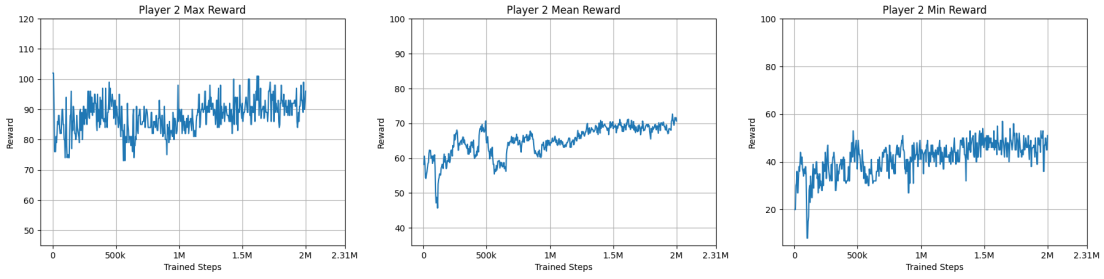


Figure E.2 PPO Player 2 Max, Mean and Min Scores

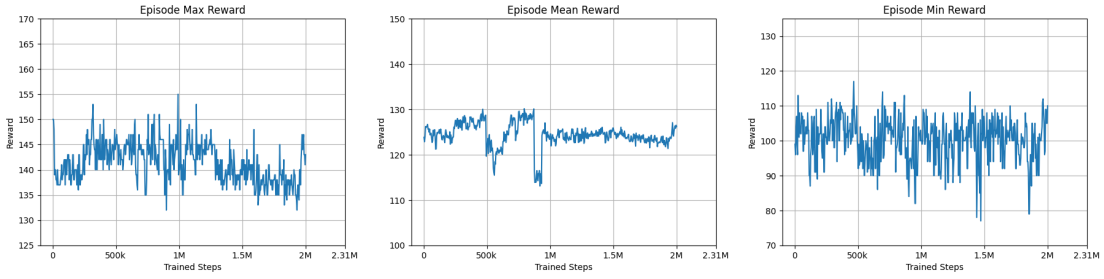


Figure E.3 PPO Episode Max, Mean and Min Scores

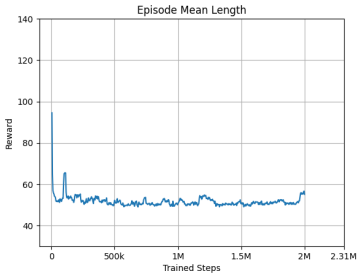


Figure E.4 PPO Episode Average Lengths

E.2 Initial Implementation with Action Masking - A2C

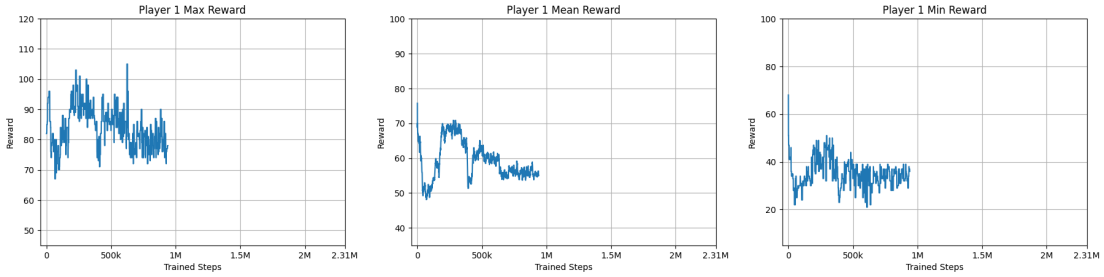


Figure E.5 A2C Player 1 Max, Mean and Min Scores

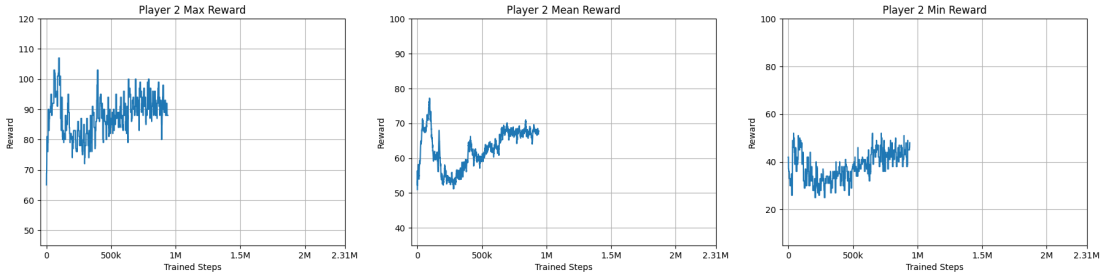


Figure E.6 A2C Player 2 Max, Mean and Min Scores

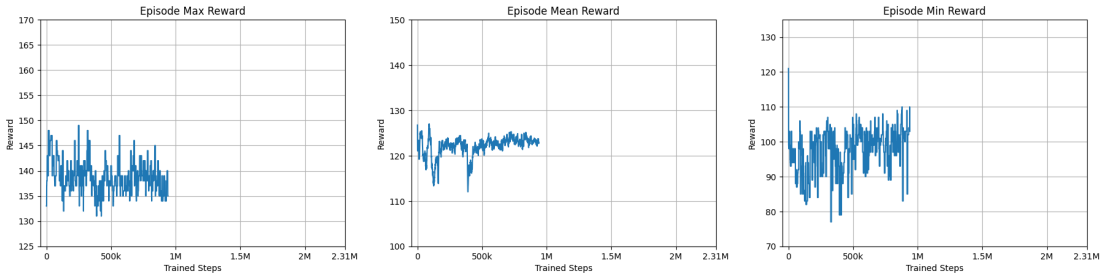


Figure E.7 A2C Episode Max, Mean and Min Scores

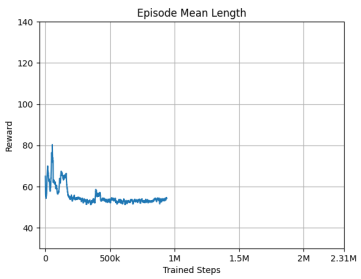


Figure E.8 A2C Episode Average Lengths

E.3 Initial Implementation with Action Masking - DQN

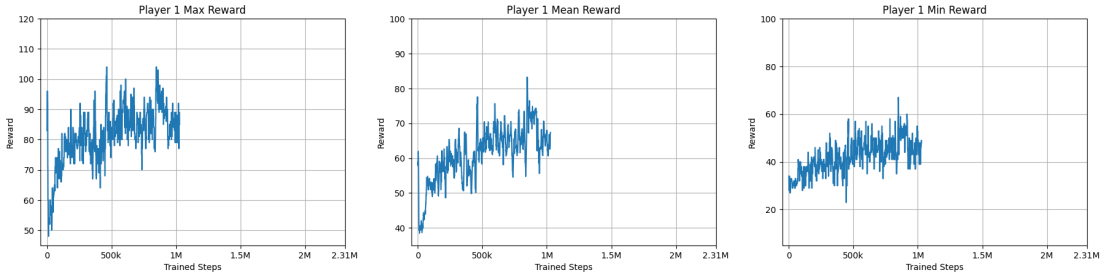


Figure E.9 DQN Player 1 Max, Mean and Min Scores

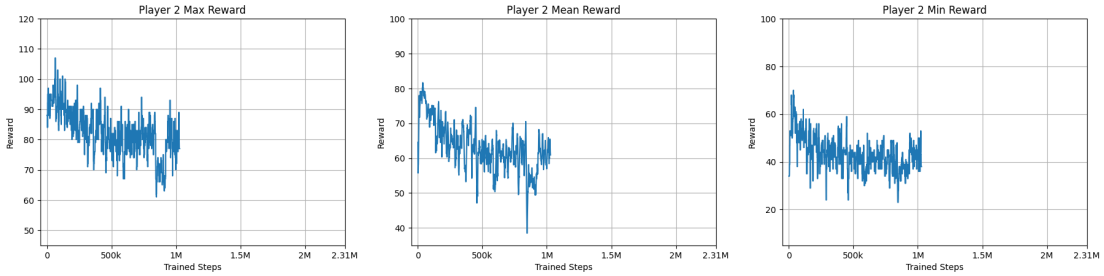


Figure E.10 DQN Player 2 Max, Mean and Min Scores

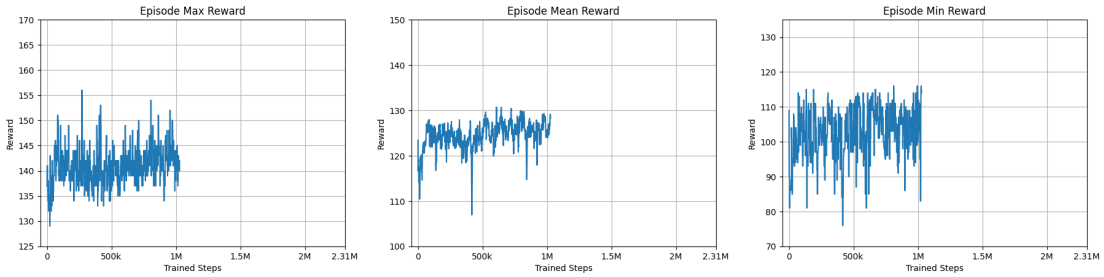


Figure E.11 DQN Episode Max, Mean and Min Scores

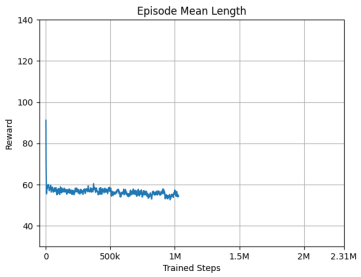


Figure E.12 DQN Episode Average Lengths

E.4 Initial Implementation with Action Masking - DDQN

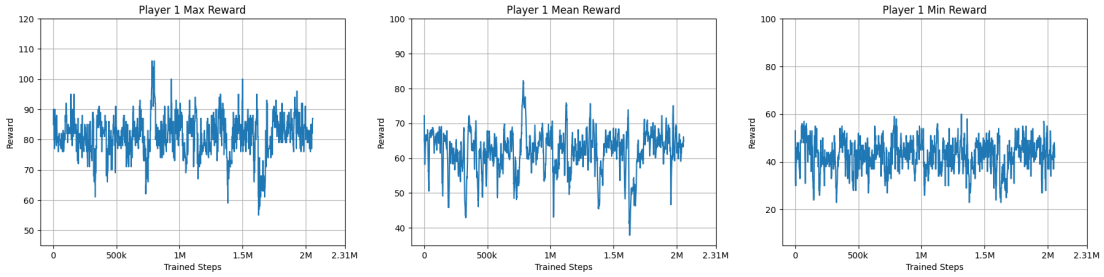


Figure E.13 DDQN Player 1 Max, Mean and Min Scores

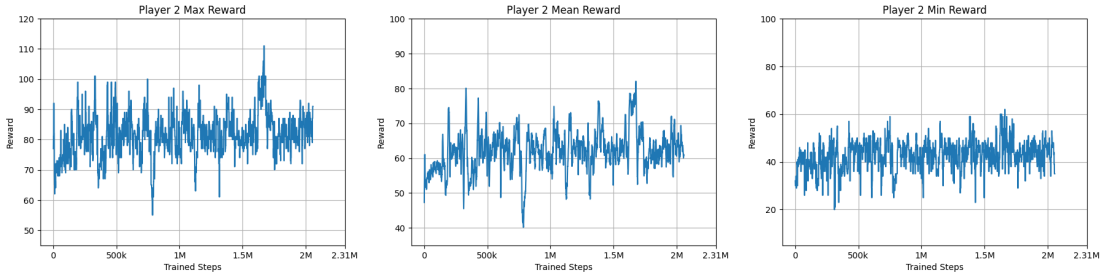


Figure E.14 DDQN Player 2 Max, Mean and Min Scores

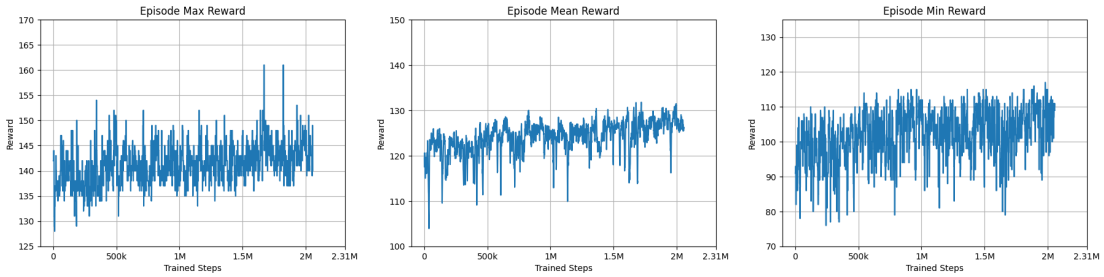


Figure E.15 DDQN Episode Max, Mean and Min Scores

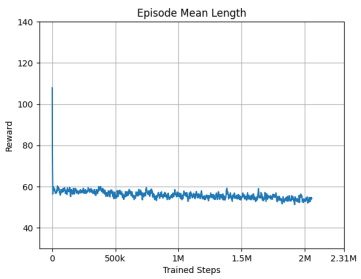


Figure E.16 DDQN Episode Average Lengths

E.5 Increased Action Space - PPO

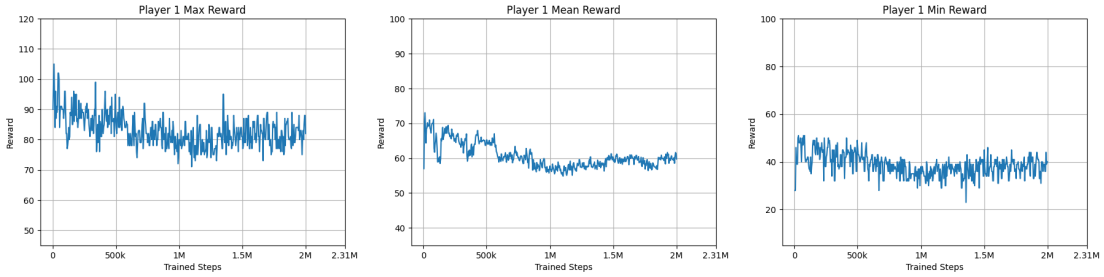


Figure E.17 PPO Player 1 Max, Mean and Min Scores

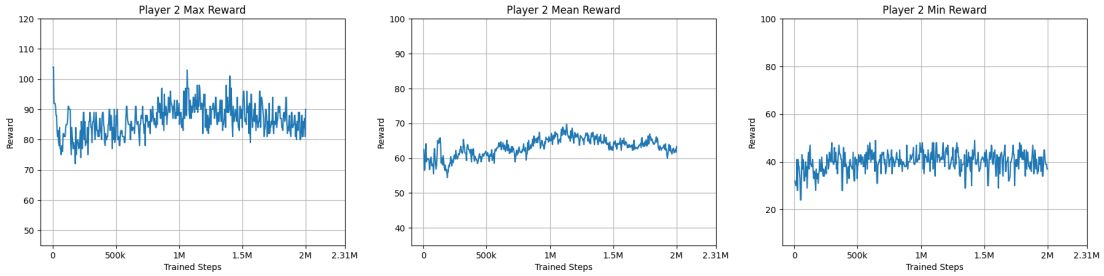


Figure E.18 PPO Player 2 Max, Mean and Min Scores

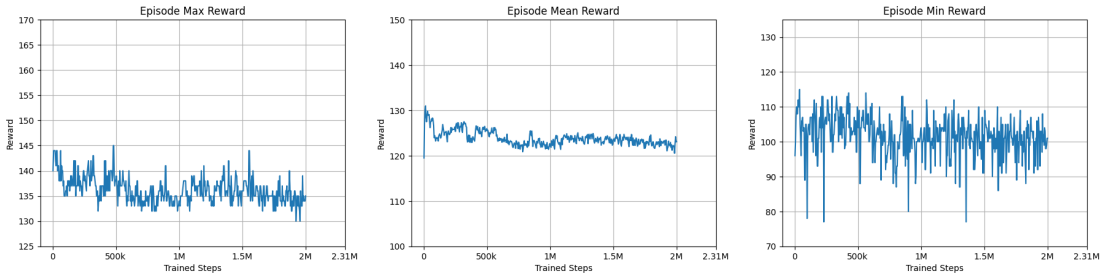


Figure E.19 PPO Episode Max, Mean and Min Scores

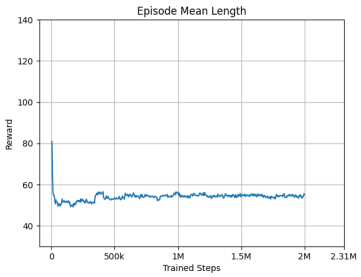


Figure E.20 PPO Episode Average Lengths

E.6 Increased Action Space - A2C

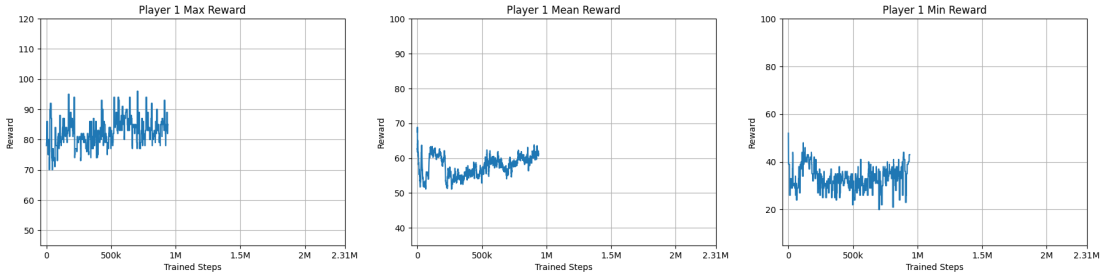


Figure E.21 A2C Player 1 Max, Mean and Min Scores

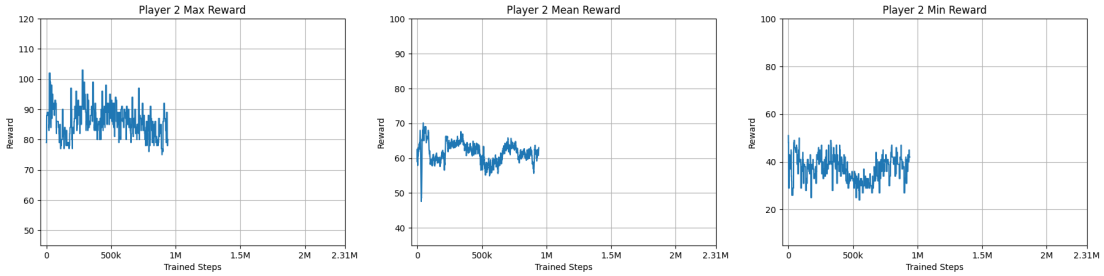


Figure E.22 A2C Player 2 Max, Mean and Min Scores

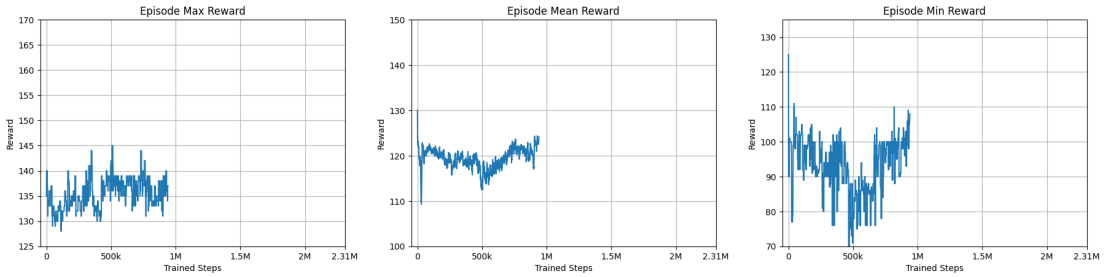


Figure E.23 A2C Episode Max, Mean and Min Scores

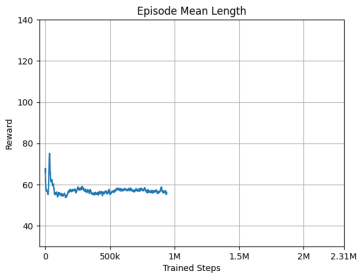


Figure E.24 A2C Episode Average Lengths

E.7 Increased Action Space - DQN

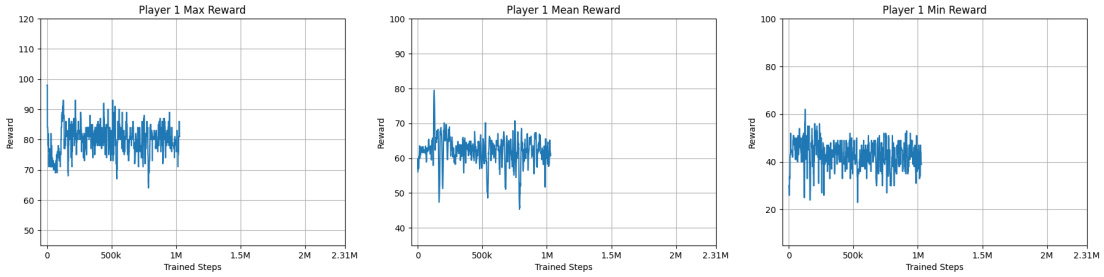


Figure E.25 DQN Player 1 Max, Mean and Min Scores

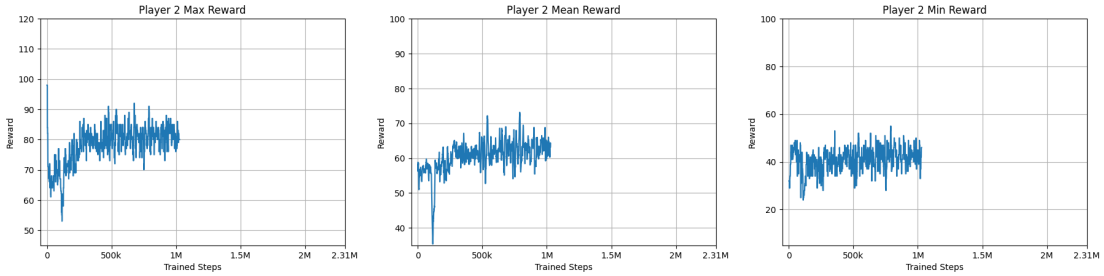


Figure E.26 DQN Player 2 Max, Mean and Min Scores

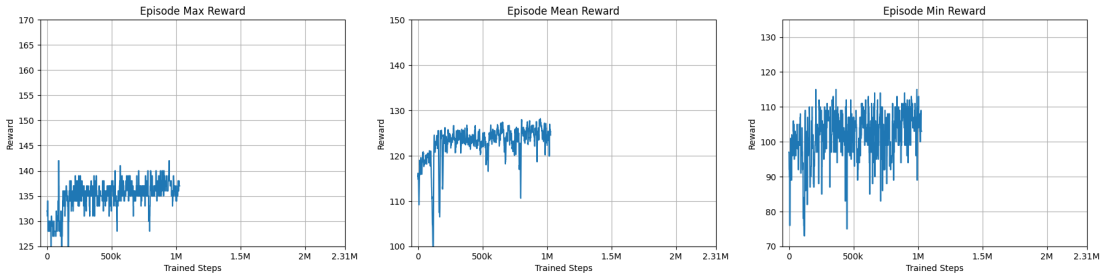


Figure E.27 DQN Episode Max, Mean and Min Scores

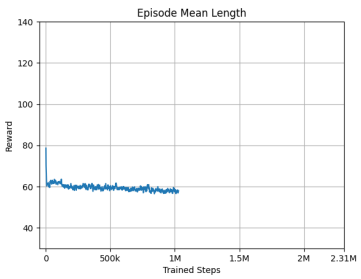


Figure E.28 DQN Episode Average Lengths

E.8 Increased Action Space - DDQN

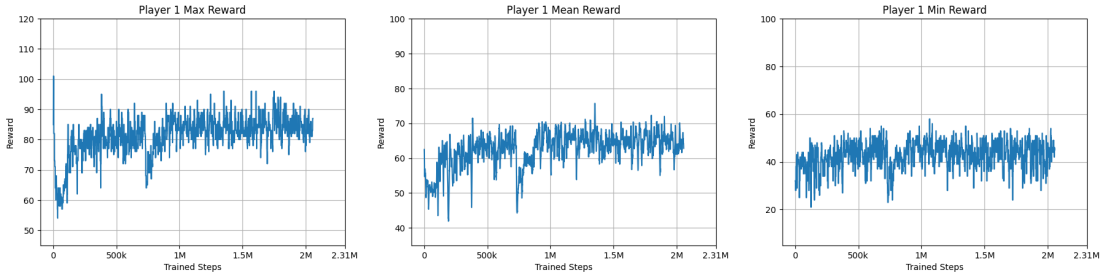


Figure E.29 DDQN Player 1 Max, Mean and Min Scores

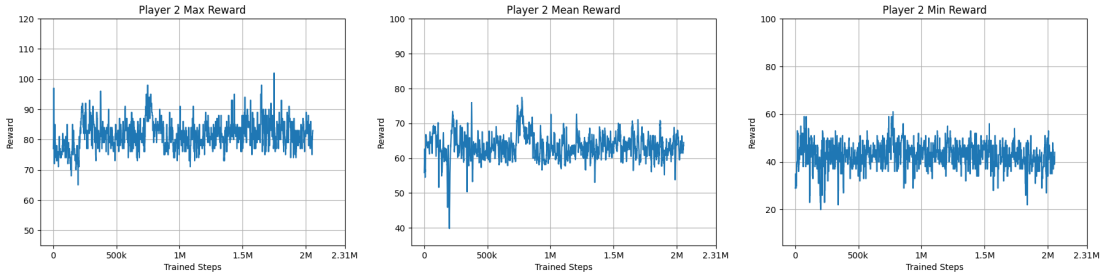


Figure E.30 DDQN Player 2 Max, Mean and Min Scores

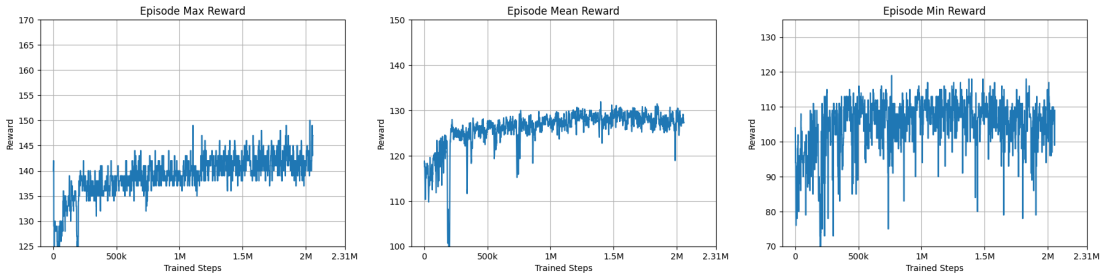


Figure E.31 DDQN Episode Max, Mean and Min Scores

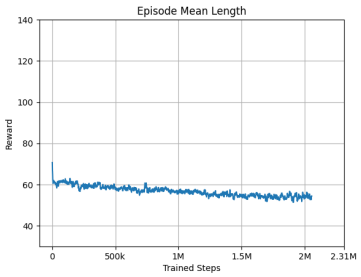


Figure E.32 DDQN Episode Average Lengths

E.9 Hyperparameter Tuning - PPO

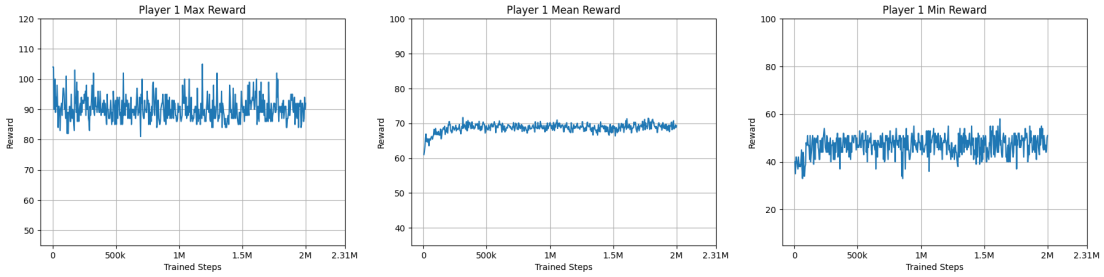


Figure E.33 PPO Player 1 Max, Mean and Min Scores

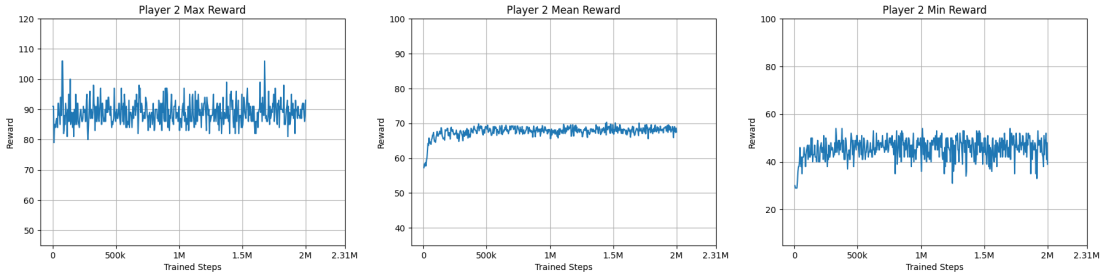


Figure E.34 PPO Player 2 Max, Mean and Min Scores

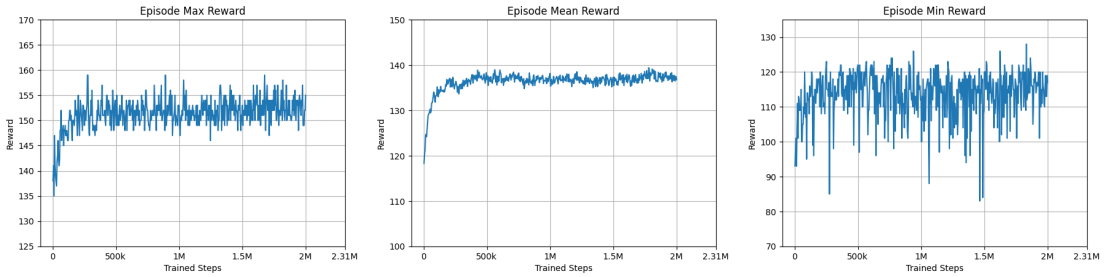


Figure E.35 PPO Episode Max, Mean and Min Scores

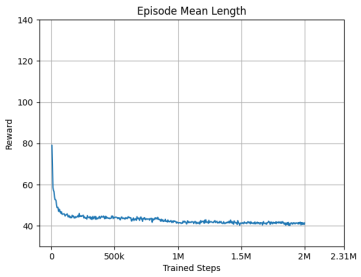


Figure E.36 PPO Episode Average Lengths

E.10 Hyperparameter Tuning - A2C

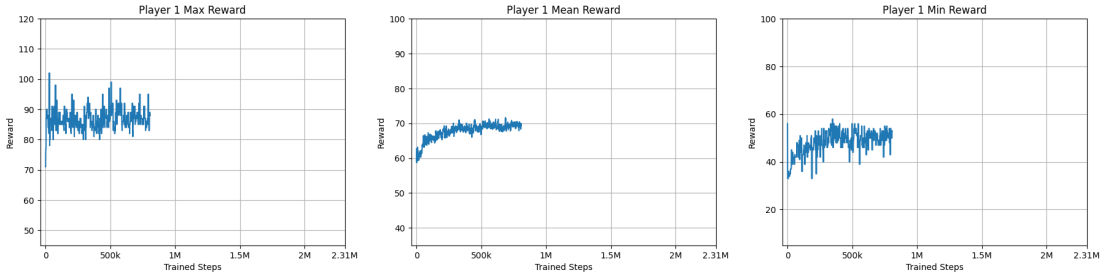


Figure E.37 A2C Player 1 Max, Mean and Min Scores

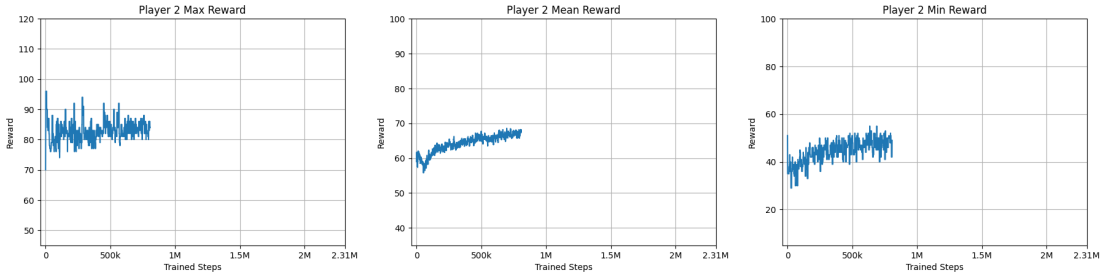


Figure E.38 A2C Player 2 Max, Mean and Min Scores

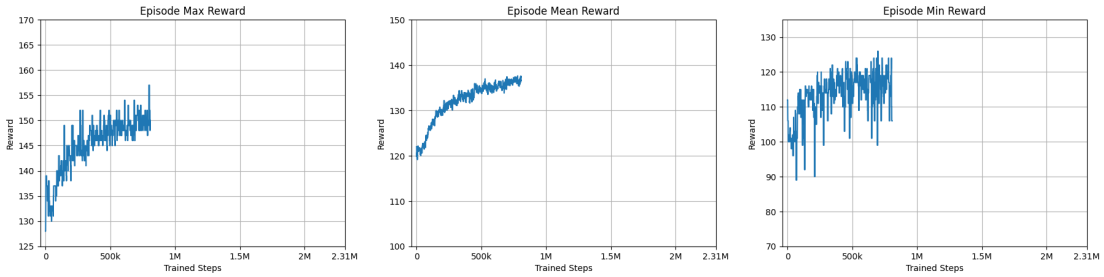


Figure E.39 A2C Episode Max, Mean and Min Scores

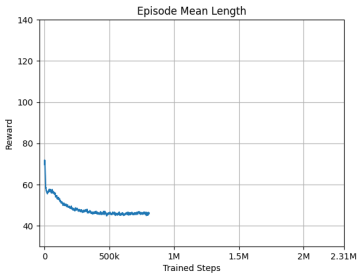


Figure E.40 A2C Episode Average Lengths

E.11 Hyperparameter Tuning - DQN

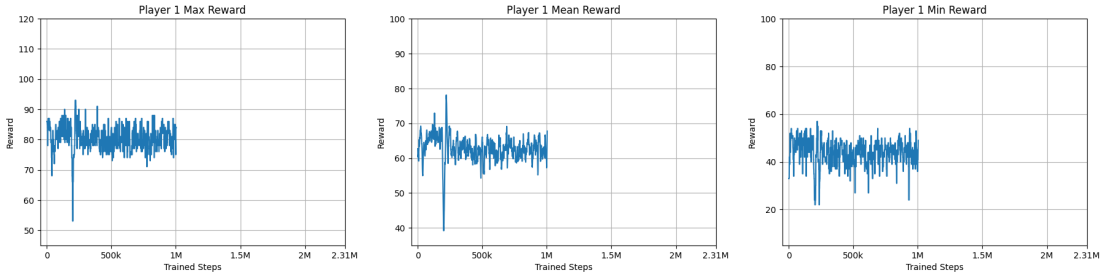


Figure E.41 DQN Player 1 Max, Mean and Min Scores

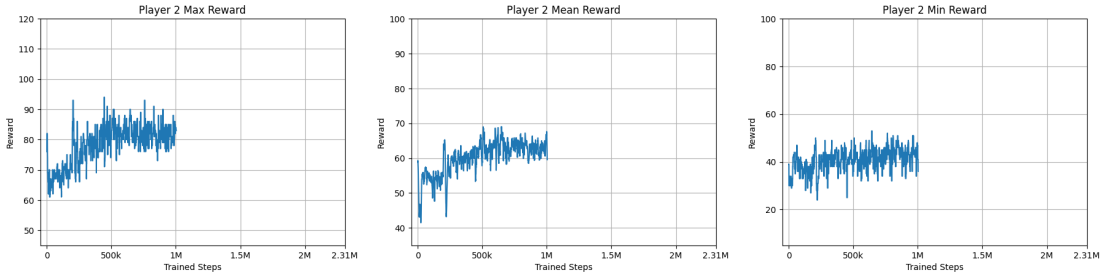


Figure E.42 DQN Player 2 Max, Mean and Min Scores

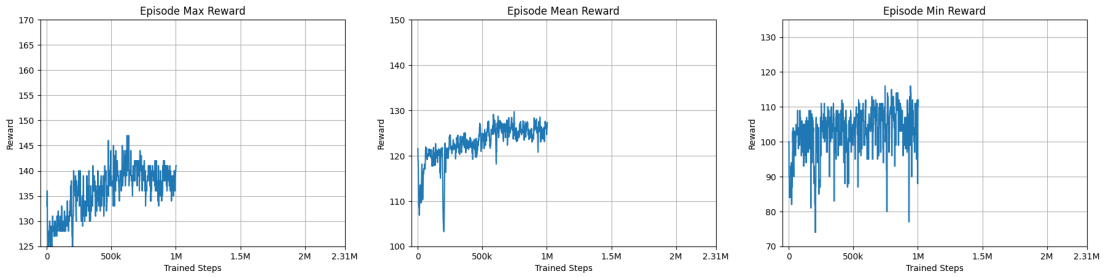


Figure E.43 DQN Episode Max, Mean and Min Scores

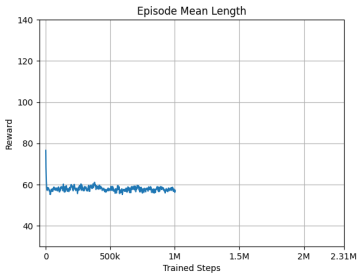


Figure E.44 DQN Episode Average Lengths

E.12 Hyperparameter Tuning - DDQN

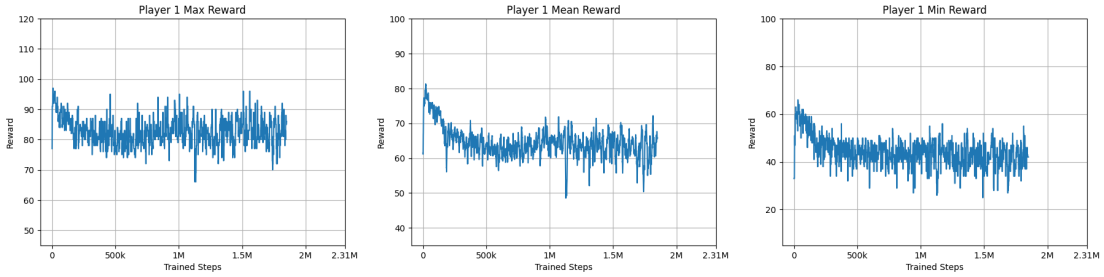


Figure E.45 DDQN Player 1 Max, Mean and Min Scores

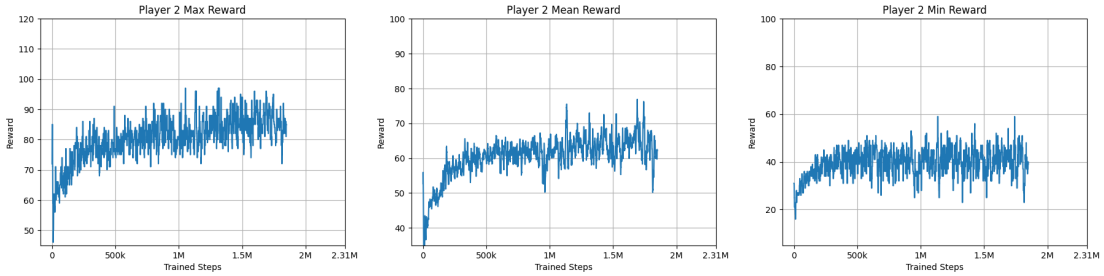


Figure E.46 DDQN Player 2 Max, Mean and Min Scores

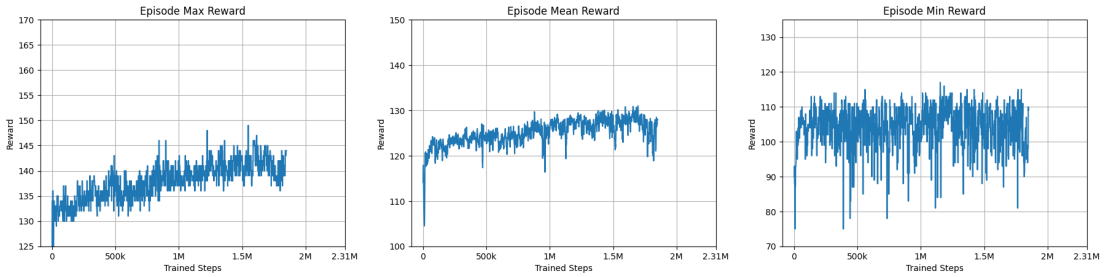


Figure E.47 DDQN Episode Max, Mean and Min Scores

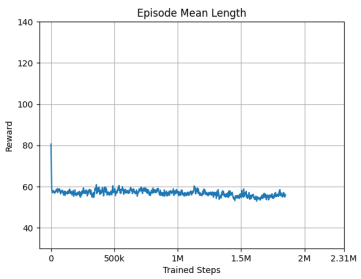


Figure E.48 DDQN Episode Average Lengths

E.13 Increased Agent Observation - PPO

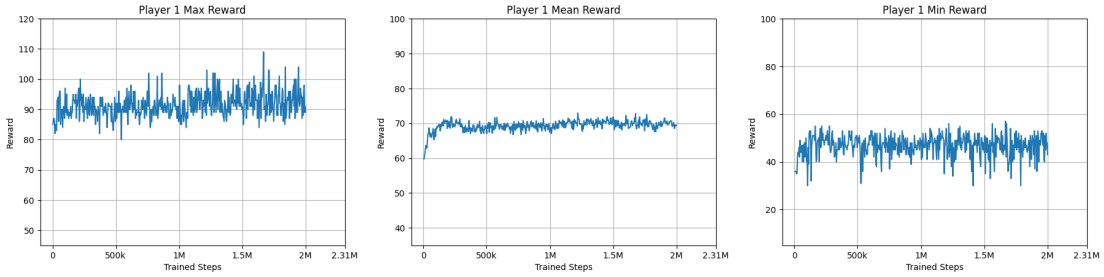


Figure E.49 PPO Player 1 Max, Mean and Min Scores

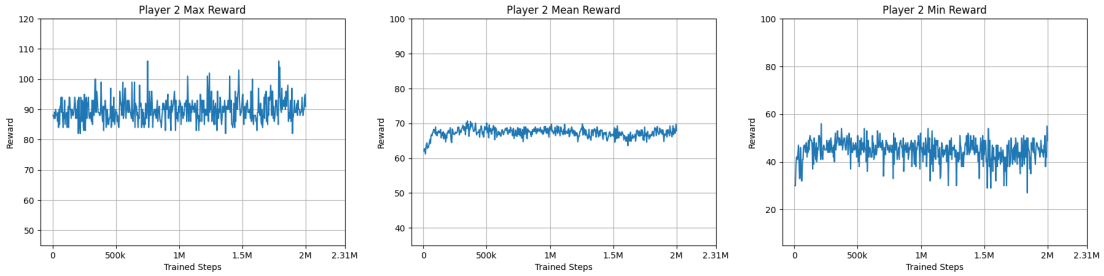


Figure E.50 PPO Player 2 Max, Mean and Min Scores

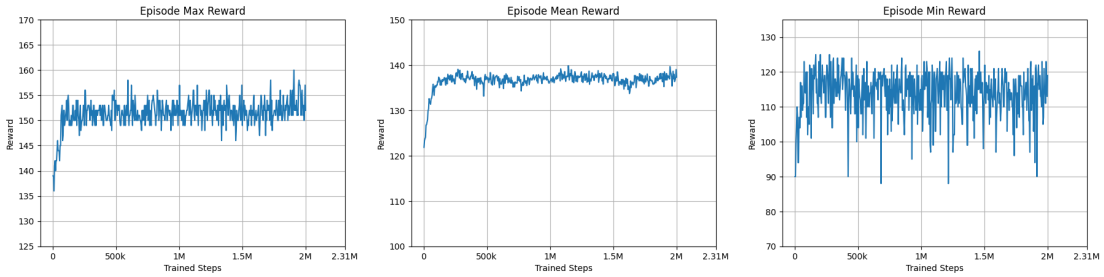


Figure E.51 PPO Episode Max, Mean and Min Scores

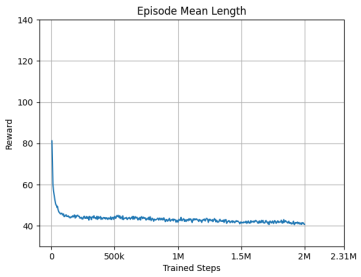


Figure E.52 PPO Episode Average Lengths

E.14 Increased Agent Observation - A2C

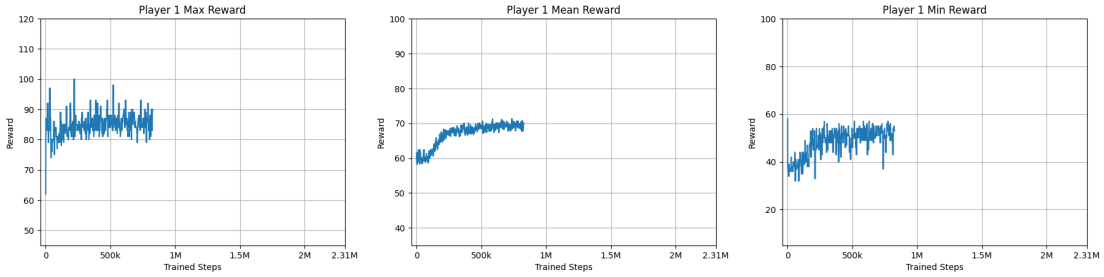


Figure E.53 A2C Player 1 Max, Mean and Min Scores

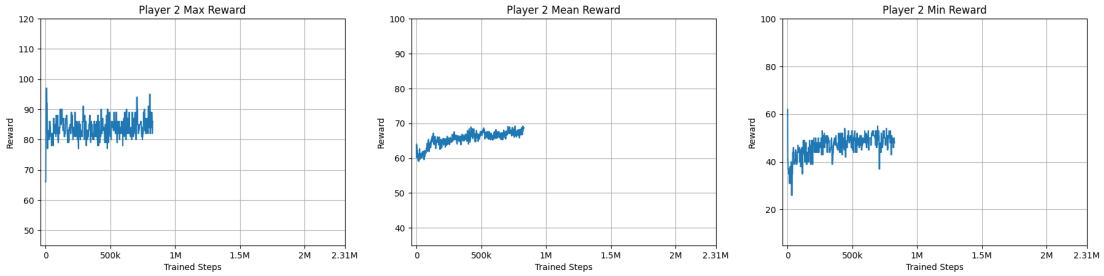


Figure E.54 A2C Player 2 Max, Mean and Min Scores

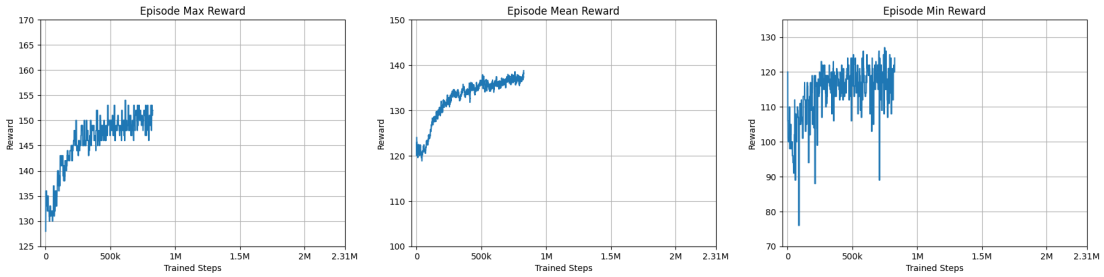


Figure E.55 A2C Episode Max, Mean and Min Scores

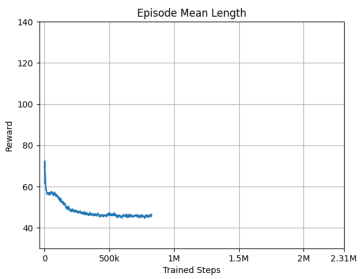


Figure E.56 A2C Episode Average Lengths

E.15 Increased Agent Observation - DQN

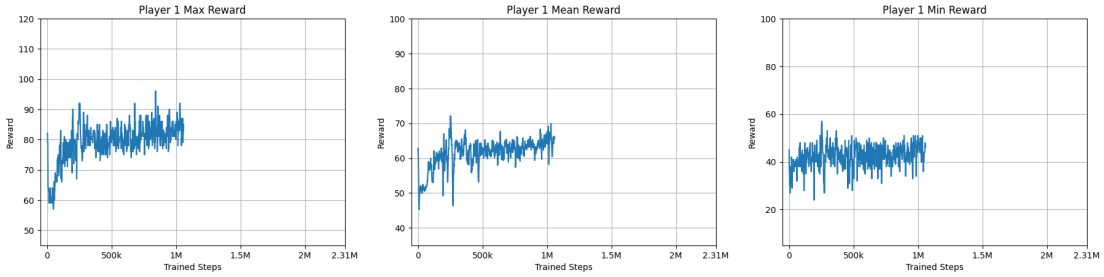


Figure E.57 DQN Player 1 Max, Mean and Min Scores

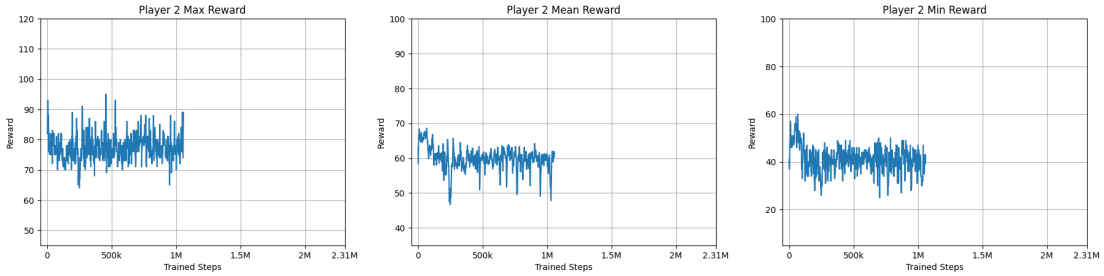


Figure E.58 DQN Player 2 Max, Mean and Min Scores

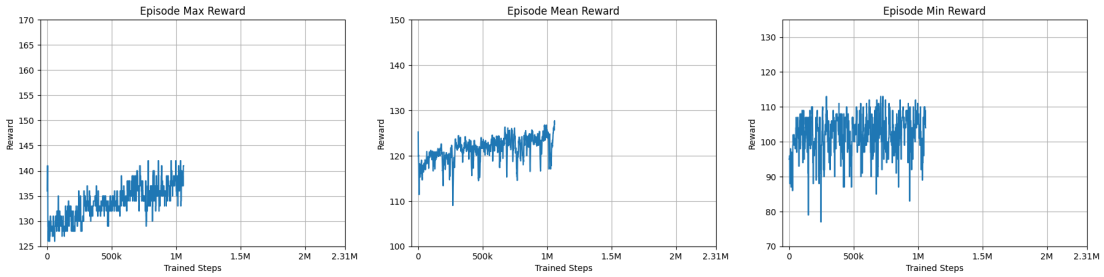


Figure E.59 DQN Episode Max, Mean and Min Scores

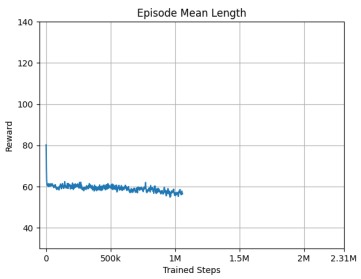


Figure E.60 DQN Episode Average Lengths

E.16 Increased Agent Observation - DDQN

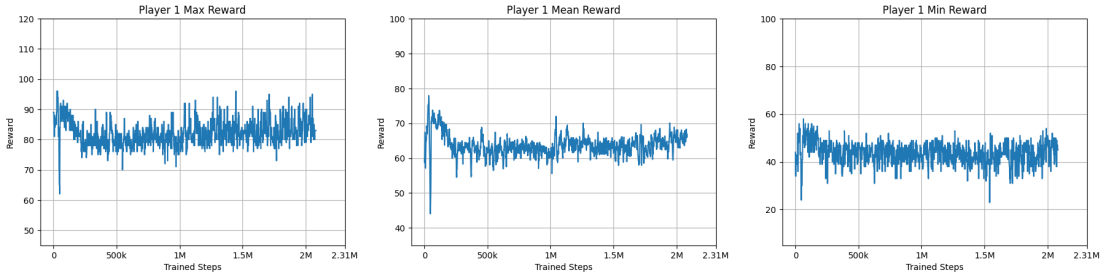


Figure E.61 DDQN Player 1 Max, Mean and Min Scores

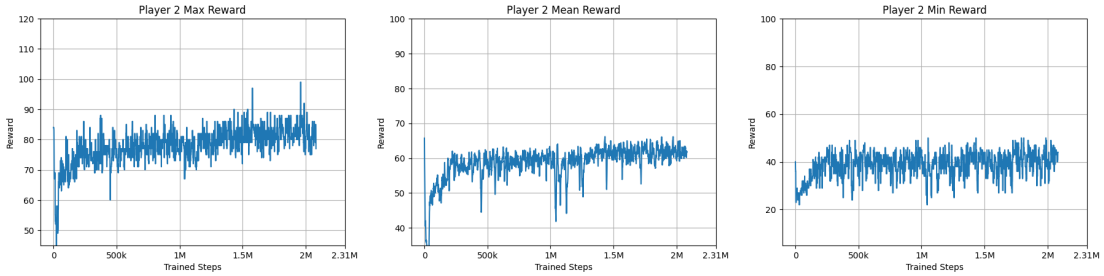


Figure E.62 DDQN Player 2 Max, Mean and Min Scores

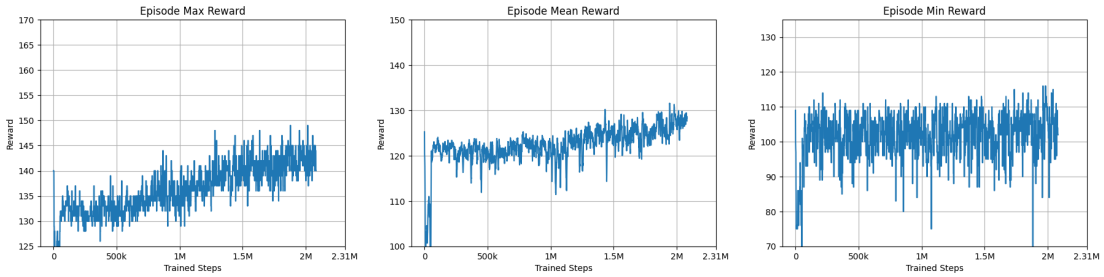


Figure E.63 DDQN Episode Max, Mean and Min Scores

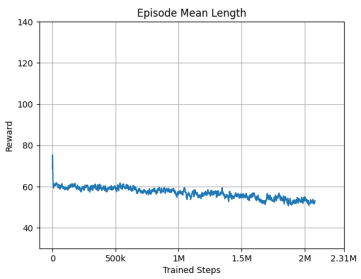


Figure E.64 DDQN Episode Average Lengths

E.17 Full Opponent Observation - PPO

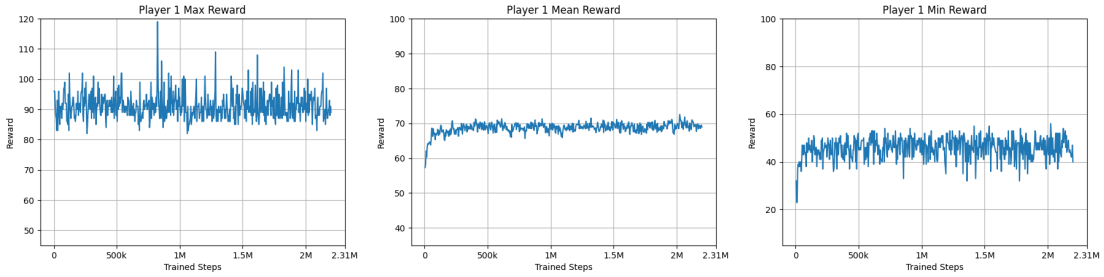


Figure E.65 PPO Player 1 Max, Mean and Min Scores

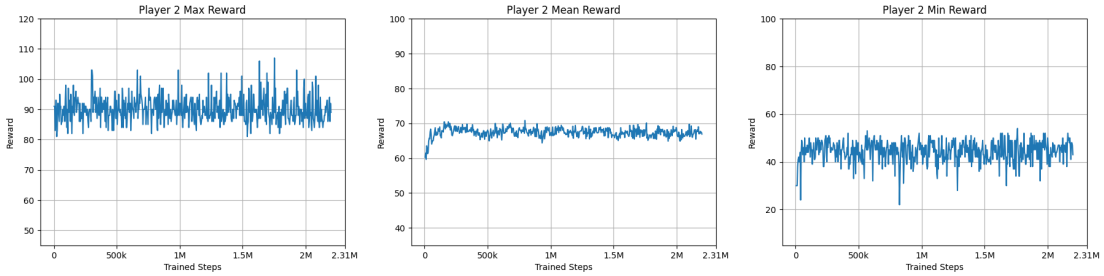


Figure E.66 PPO Player 2 Max, Mean and Min Scores

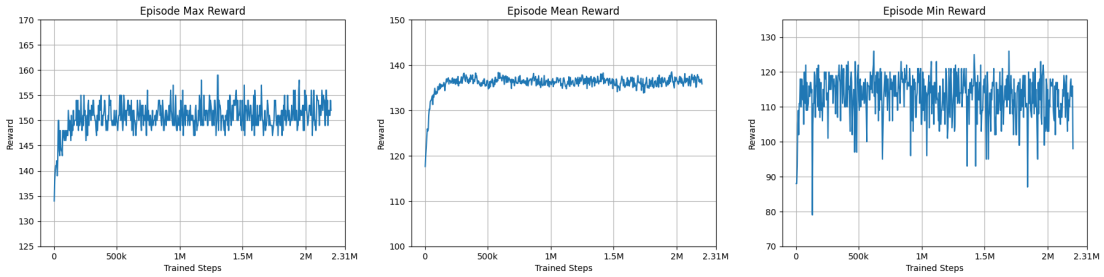


Figure E.67 PPO Episode Max, Mean and Min Scores

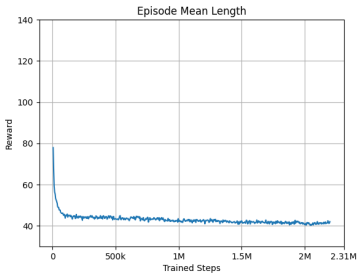


Figure E.68 PPO Episode Average Lengths

E.18 Full Opponent Observation - A2C

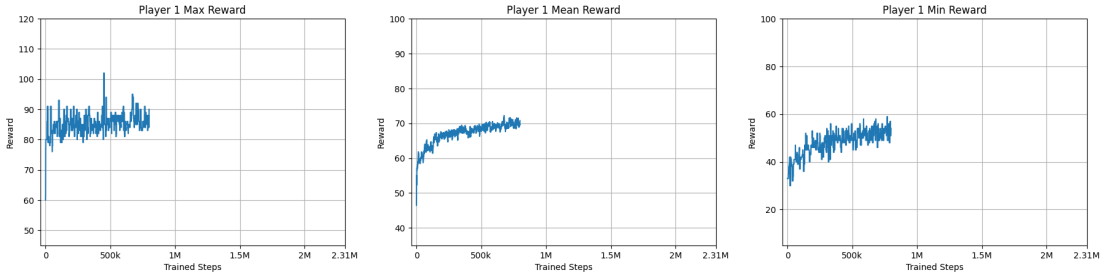


Figure E.69 A2C Player 1 Max, Mean and Min Scores

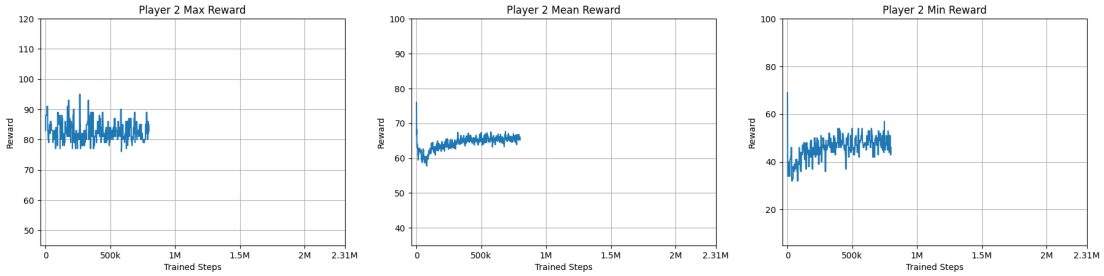


Figure E.70 A2C Player 2 Max, Mean and Min Scores

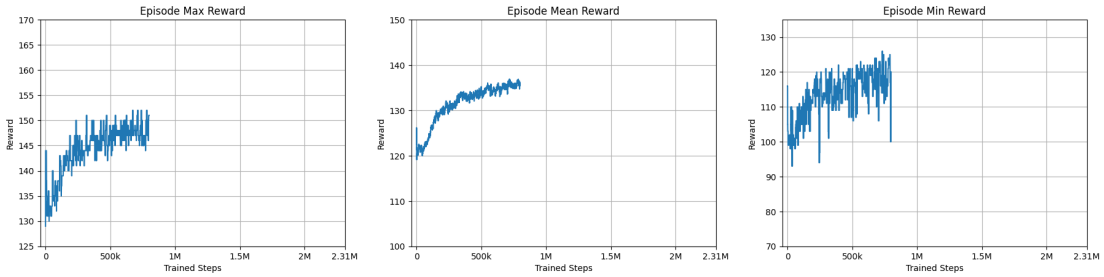


Figure E.71 A2C Episode Max, Mean and Min Scores

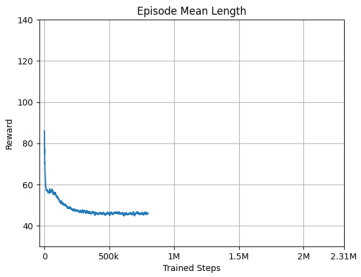


Figure E.72 A2C Episode Average Lengths

E.19 Full Opponent Observation - DQN

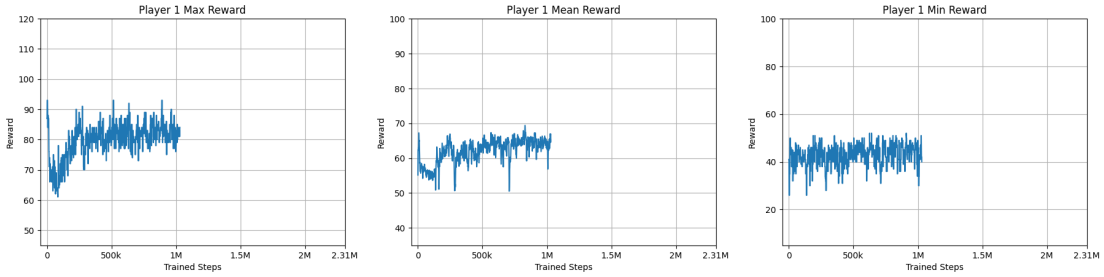


Figure E.73 DQN Player 1 Max, Mean and Min Scores

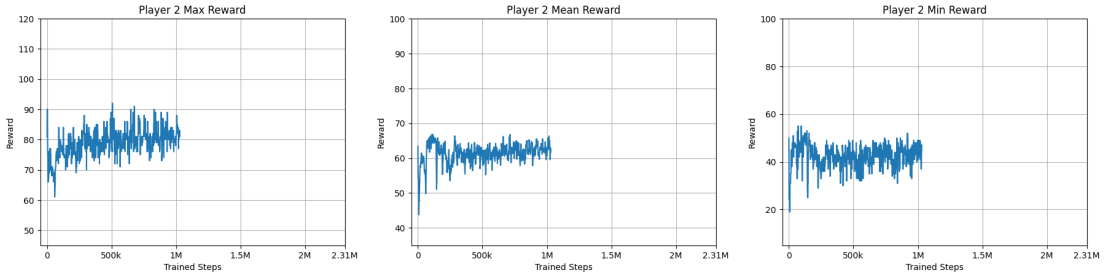


Figure E.74 DQN Player 2 Max, Mean and Min Scores

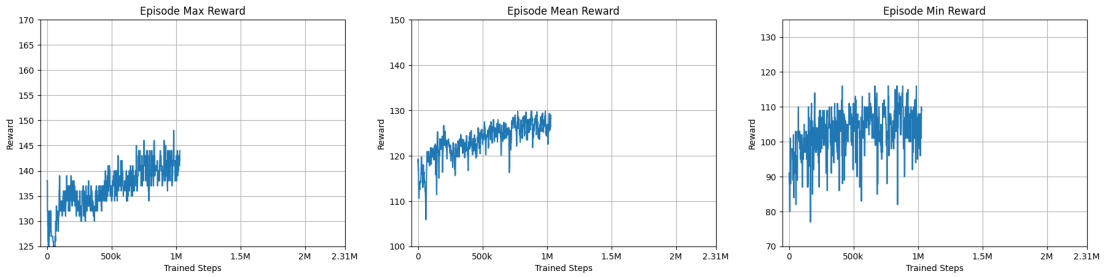


Figure E.75 DQN Episode Max, Mean and Min Scores

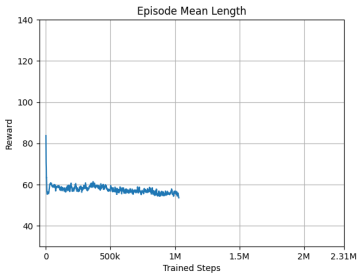


Figure E.76 DQN Episode Average Lengths

E.20 Full Opponent Observation - DDQN

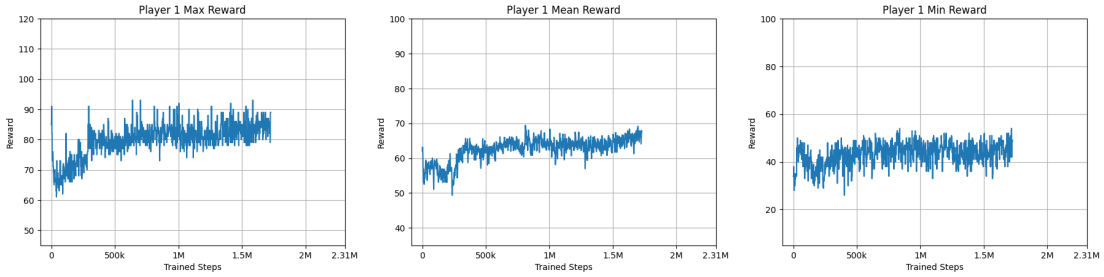


Figure E.77 DDQN Player 1 Max, Mean and Min Scores

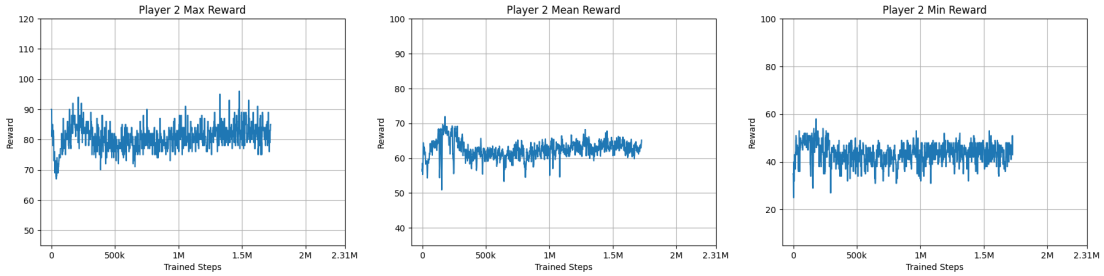


Figure E.78 DDQN Player 2 Max, Mean and Min Scores

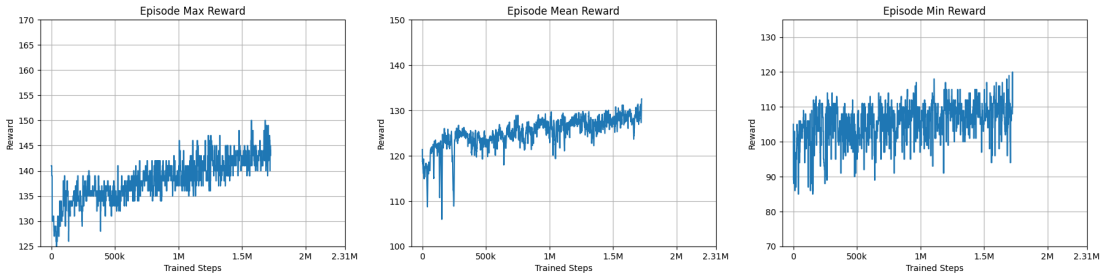


Figure E.79 DDQN Episode Max, Mean and Min Scores

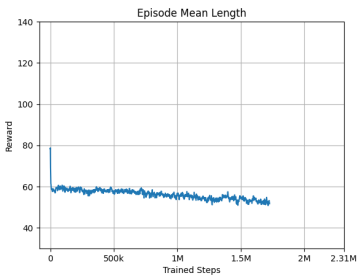


Figure E.80 DDQN Episode Average Lengths

E.21 Policy Cloning during Training - PPO

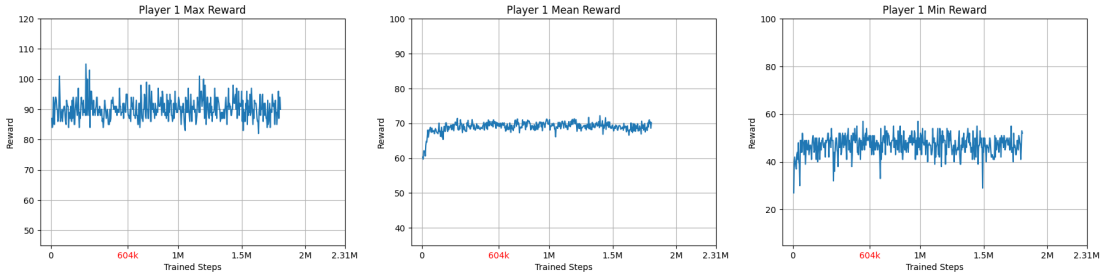


Figure E.81 PPO Player 1 Max, Mean and Min Scores

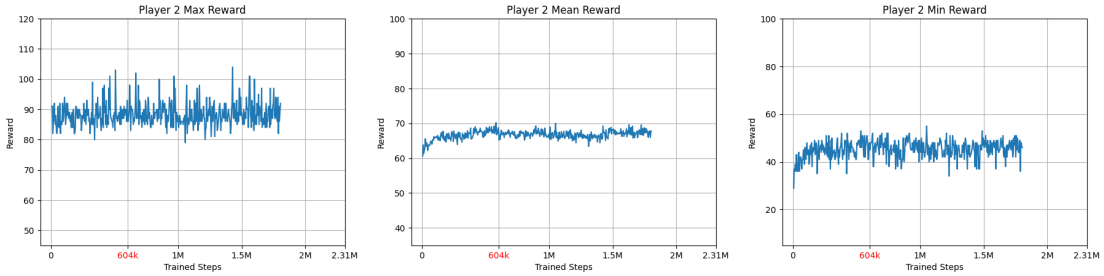


Figure E.82 PPO Player 2 Max, Mean and Min Scores

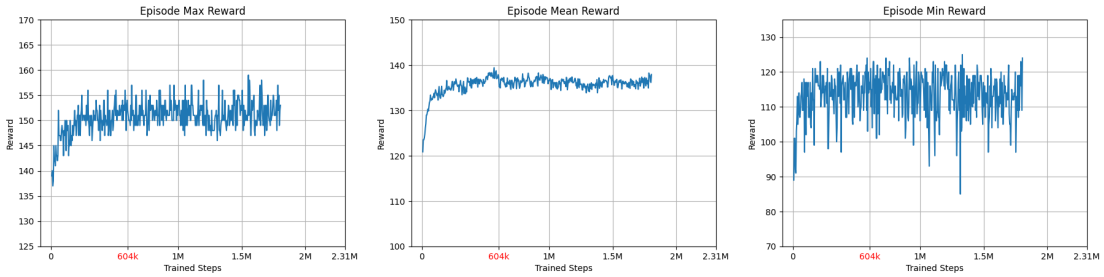


Figure E.83 PPO Episode Max, Mean and Min Scores

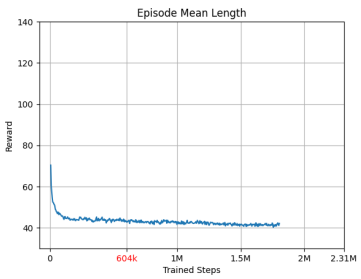


Figure E.84 PPO Episode Average Lengths

E.22 Policy Cloning during Training - A2C

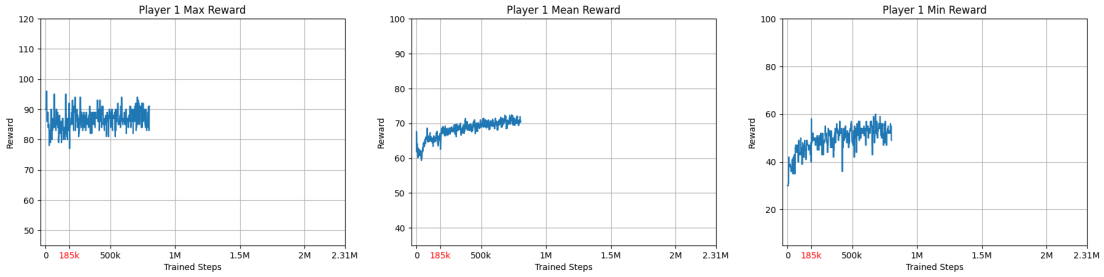


Figure E.85 A2C Player 1 Max, Mean and Min Scores

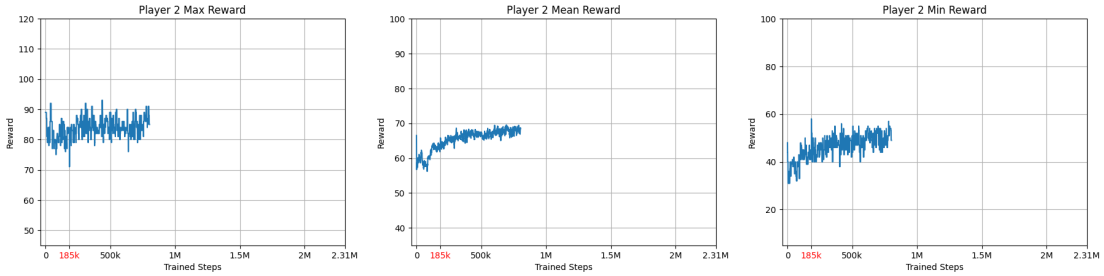


Figure E.86 A2C Player 2 Max, Mean and Min Scores

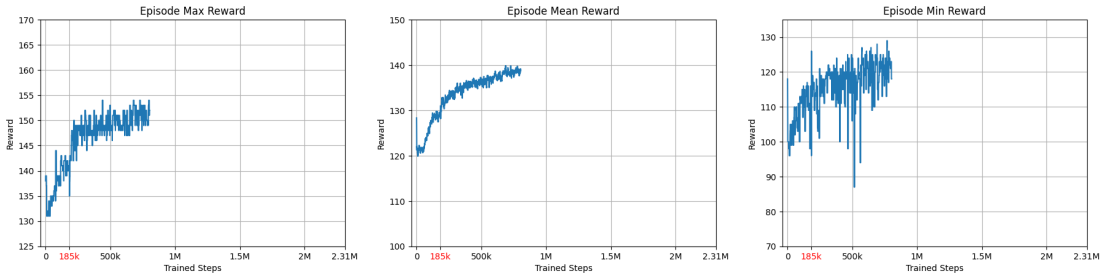


Figure E.87 A2C Episode Max, Mean and Min Scores

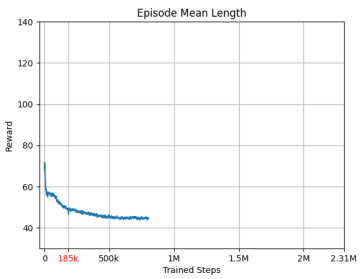


Figure E.88 A2C Episode Average Lengths

E.23 Policy Cloning during Training - DQN

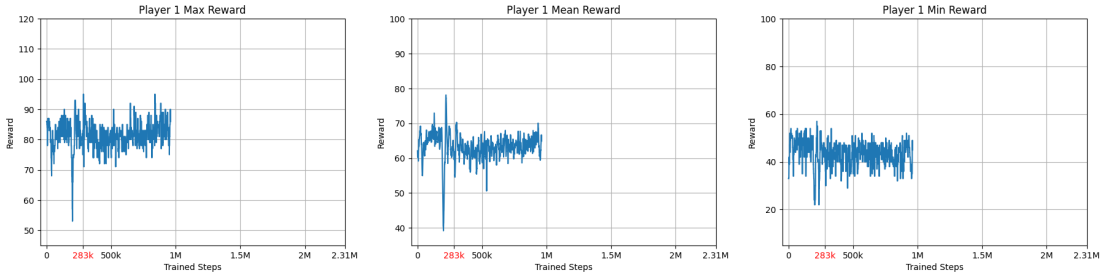


Figure E.89 DQN Player 1 Max, Mean and Min Scores

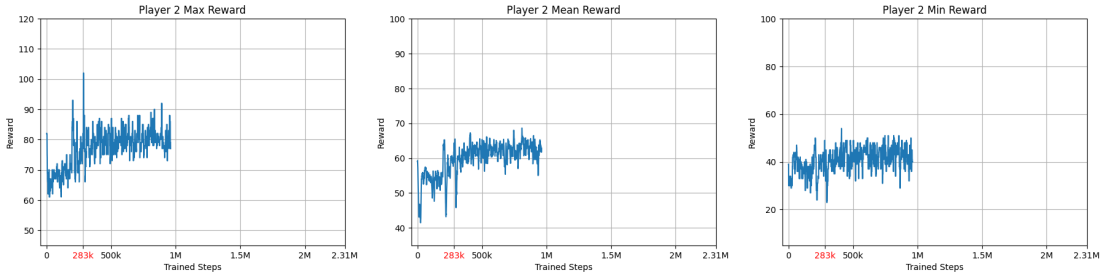


Figure E.90 DQN Player 2 Max, Mean and Min Scores

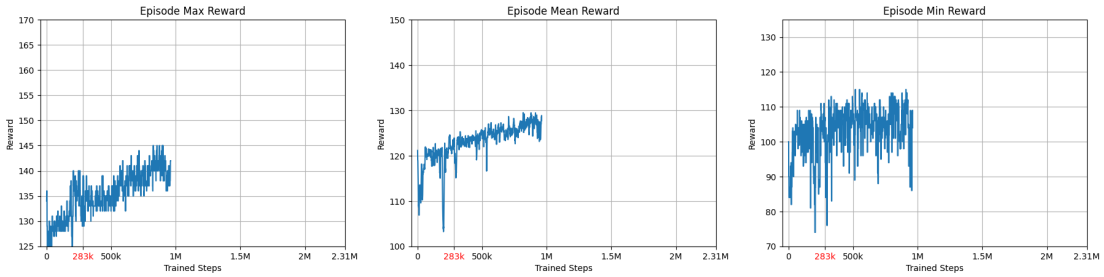


Figure E.91 DQN Episode Max, Mean and Min Scores

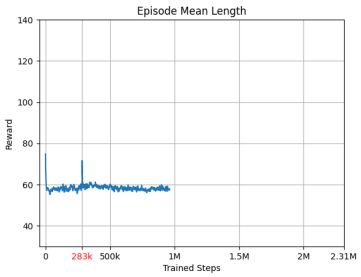


Figure E.92 DQN Episode Average Lengths

E.24 Policy Cloning during Training - DDQN

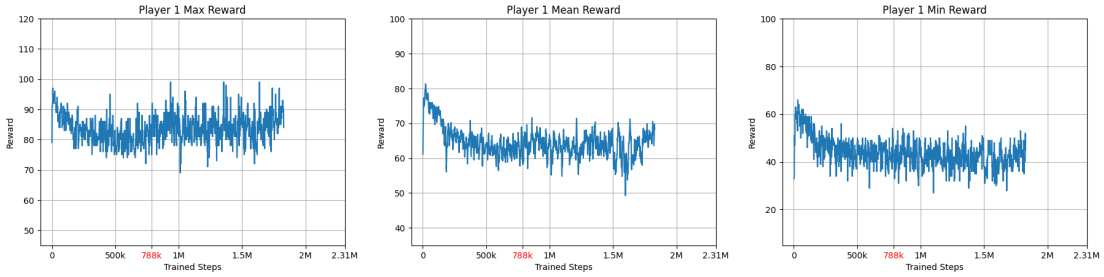


Figure E.93 DDQN Player 1 Max, Mean and Min Scores

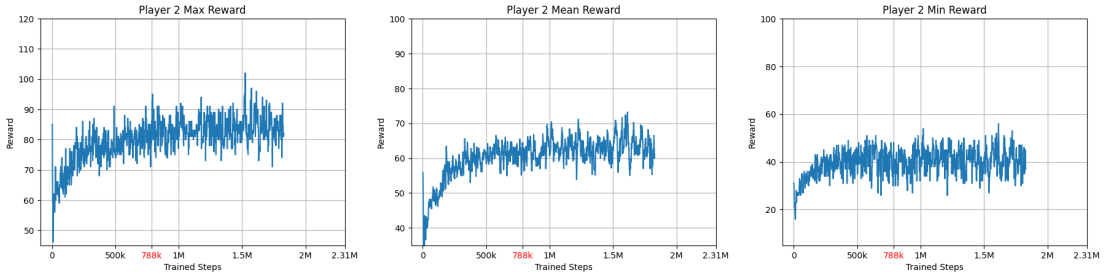


Figure E.94 DDQN Player 2 Max, Mean and Min Scores

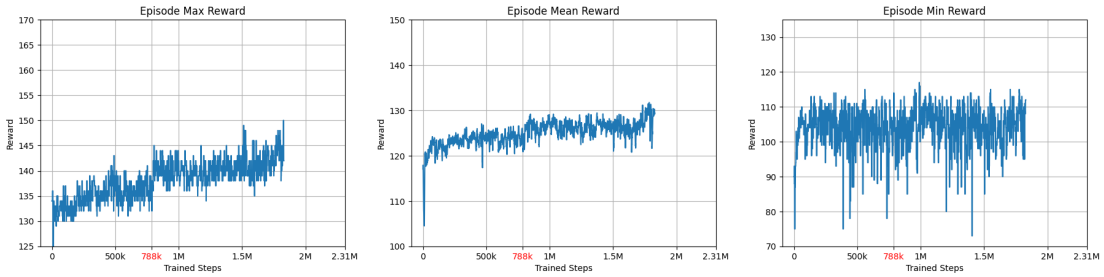


Figure E.95 DDQN Episode Max, Mean and Min Scores

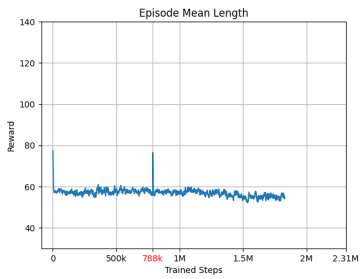


Figure E.96 DDQN Episode Average Lengths

E.25 Action Embedding - PPO

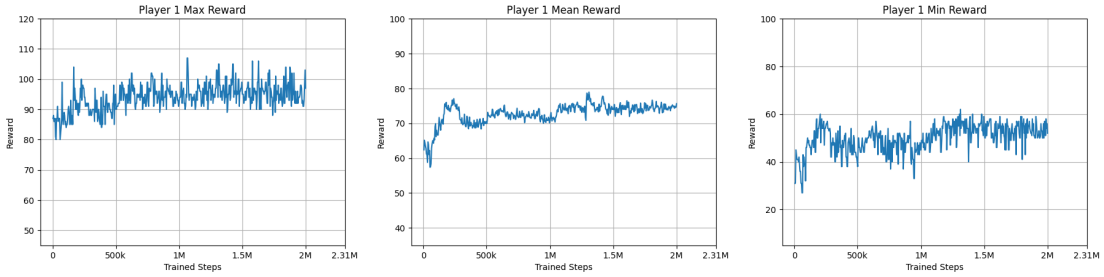


Figure E.97 PPO Player 1 Max, Mean and Min Scores

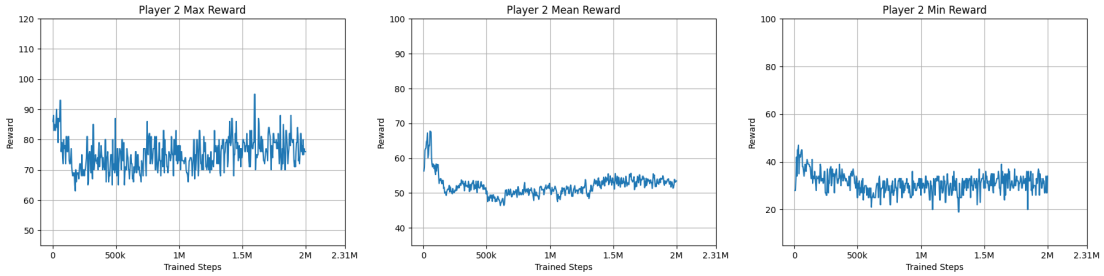


Figure E.98 PPO Player 2 Max, Mean and Min Scores

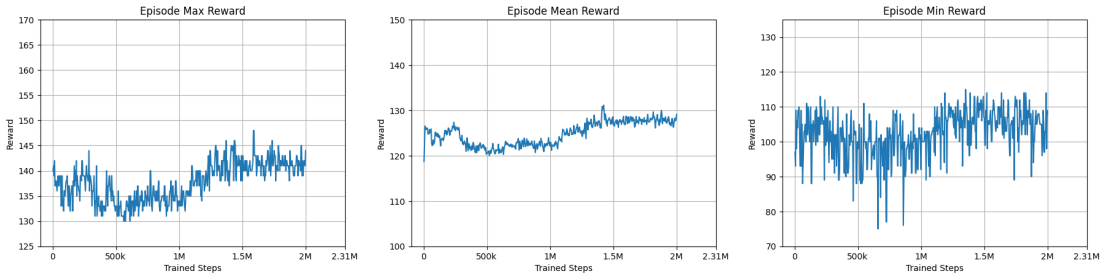


Figure E.99 PPO Episode Max, Mean and Min Scores

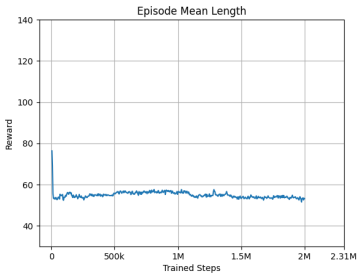


Figure E.100 PPO Episode Average Lengths

E.26 Action Embedding - A2C

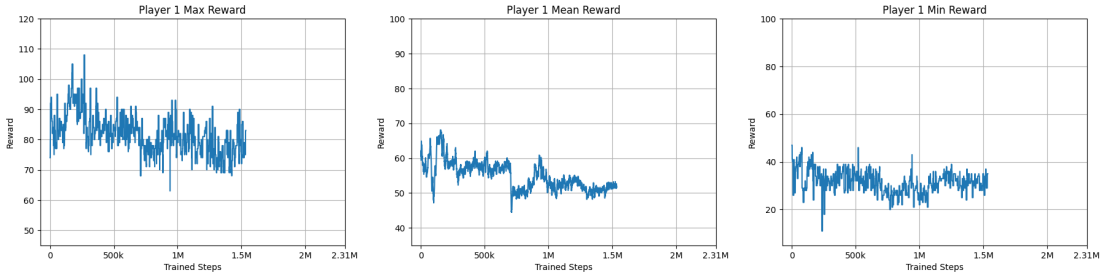


Figure E.101 A2C Player 1 Max, Mean and Min Scores

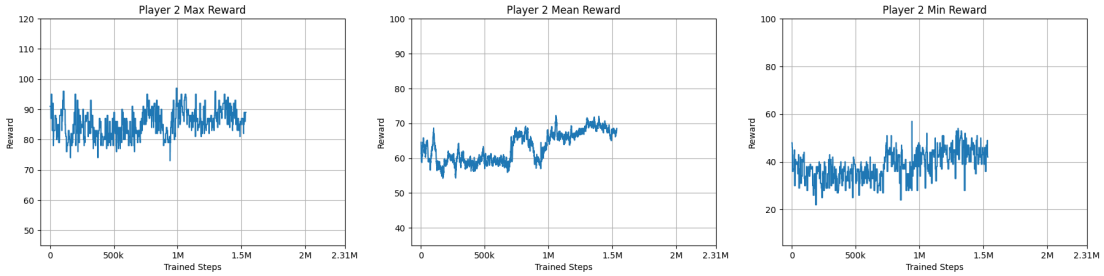


Figure E.102 A2C Player 2 Max, Mean and Min Scores

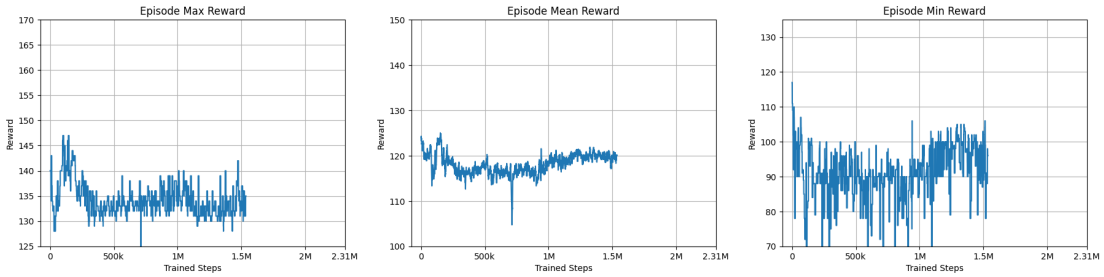


Figure E.103 A2C Episode Max, Mean and Min Scores

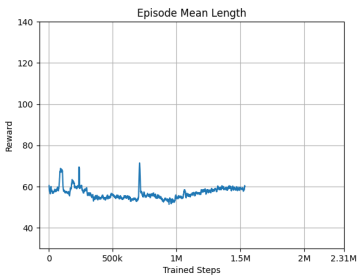


Figure E.104 A2C Episode Average Lengths

E.27 Action Embedding - DQN

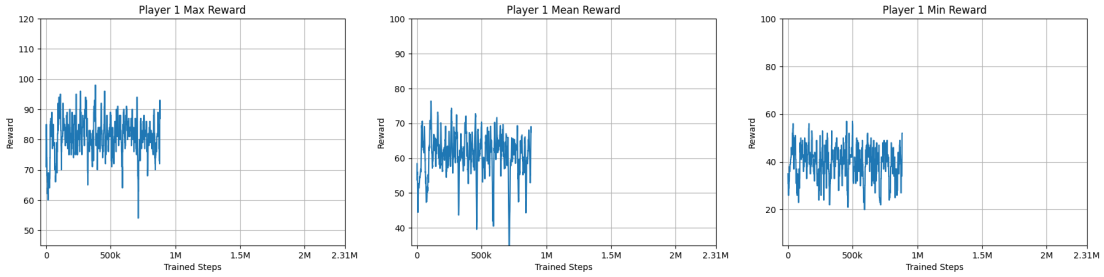


Figure E.105 DQN Player 1 Max, Mean and Min Scores

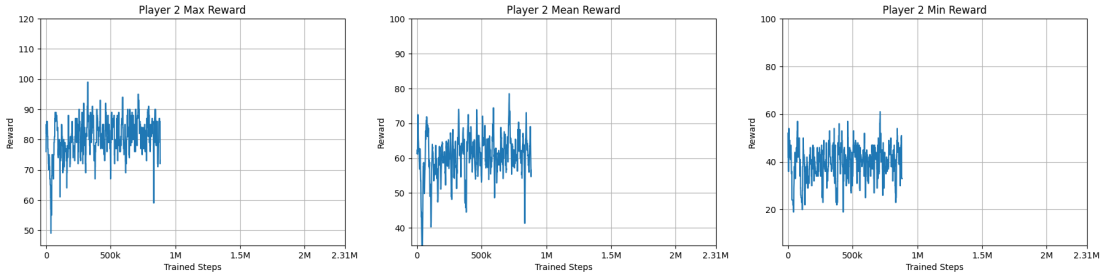


Figure E.106 DQN Player 2 Max, Mean and Min Scores

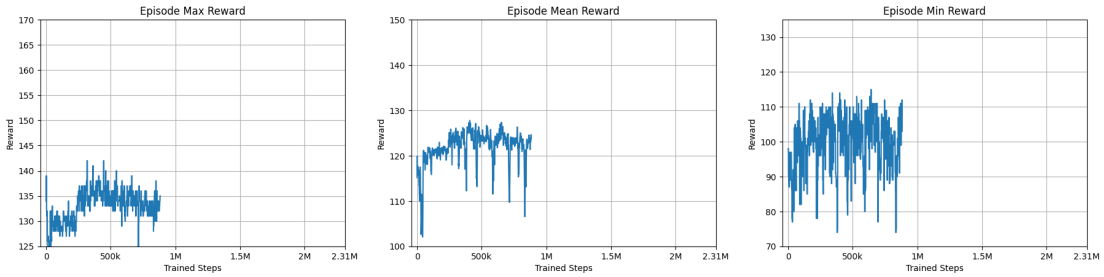


Figure E.107 DQN Episode Max, Mean and Min Scores

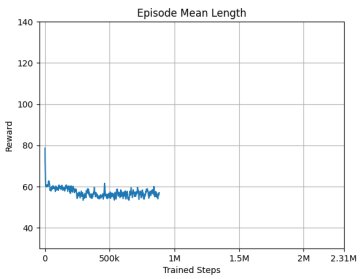


Figure E.108 DQN Episode Average Lengths

E.28 Action Embedding - DDQN

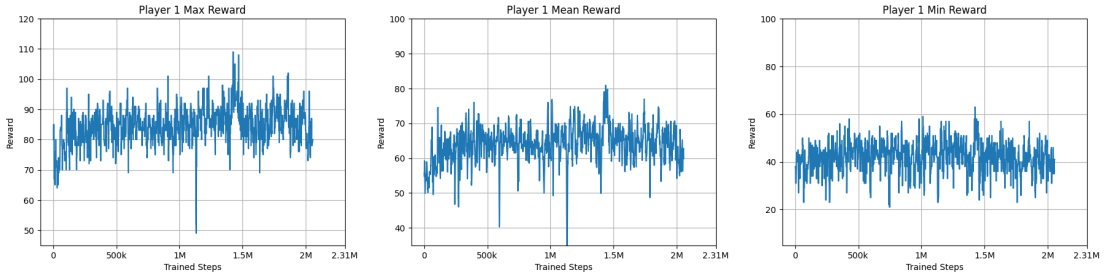


Figure E.109 DDQN Player 1 Max, Mean and Min Scores

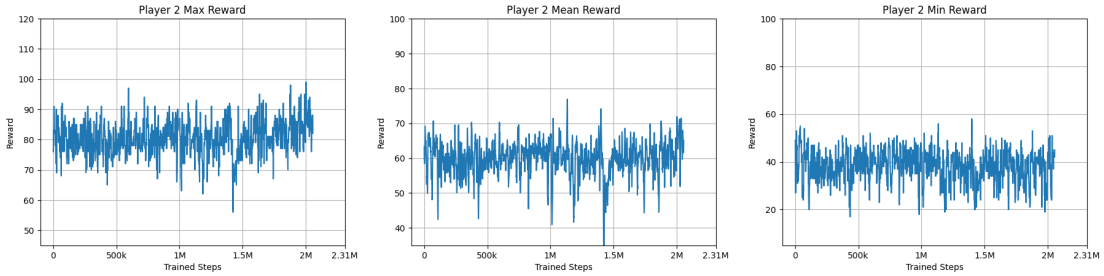


Figure E.110 DDQN Player 2 Max, Mean and Min Scores

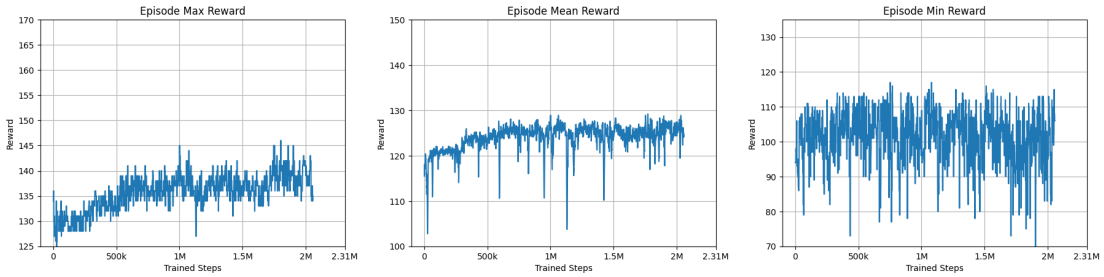


Figure E.111 DDQN Episode Max, Mean and Min Scores

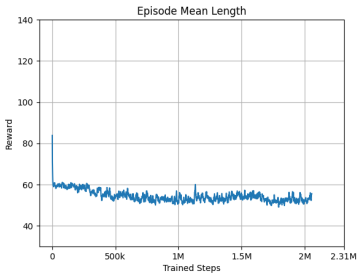


Figure E.112 DDQN Episode Average Lengths

E.29 Hierarchical RL with Centralised Critic - PPO

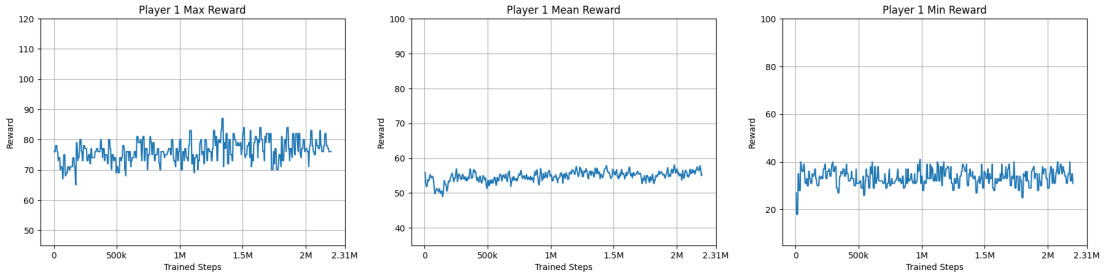


Figure E.113 PPO Player 1 Max, Mean and Min Scores

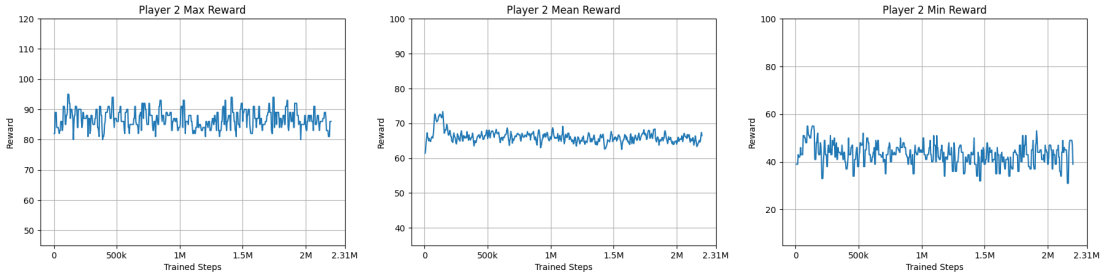


Figure E.114 PPO Player 2 Max, Mean and Min Scores

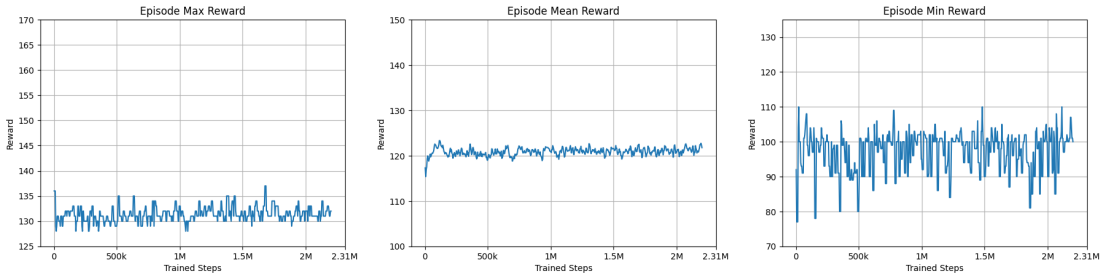


Figure E.115 PPO Episode Max, Mean and Min Scores

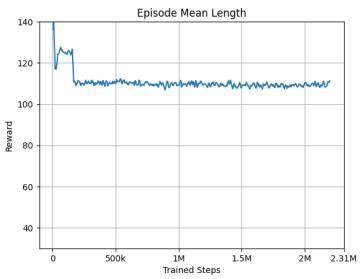


Figure E.116 PPO Episode Average Lengths

E.30 Hierarchical RL with Centralised Critic - A2C

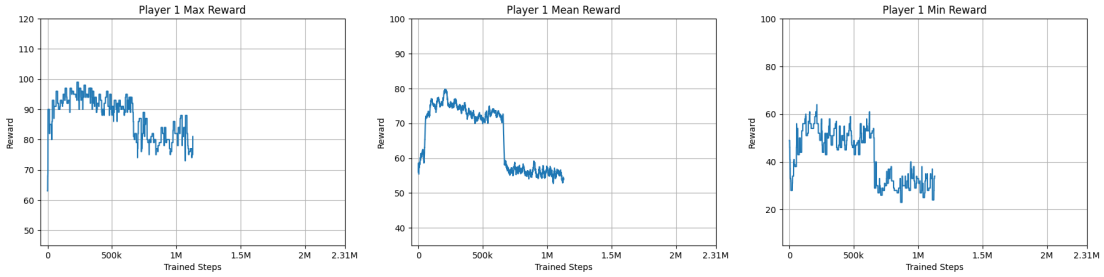


Figure E.117 A2C Player 1 Max, Mean and Min Scores

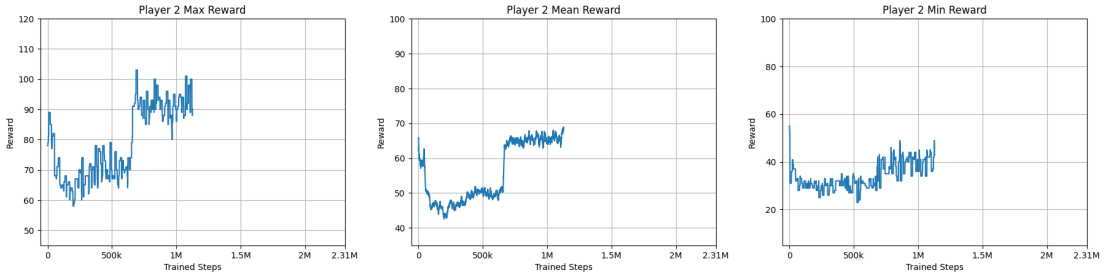


Figure E.118 A2C Player 2 Max, Mean and Min Scores

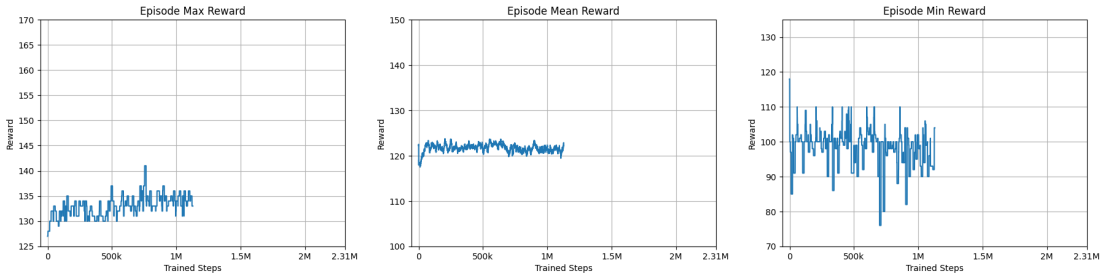


Figure E.119 A2C Episode Max, Mean and Min Scores

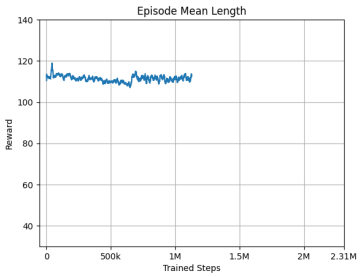


Figure E.120 A2C Episode Average Lengths

E.31 Hierarchical RL with Centralised Critic - DQN

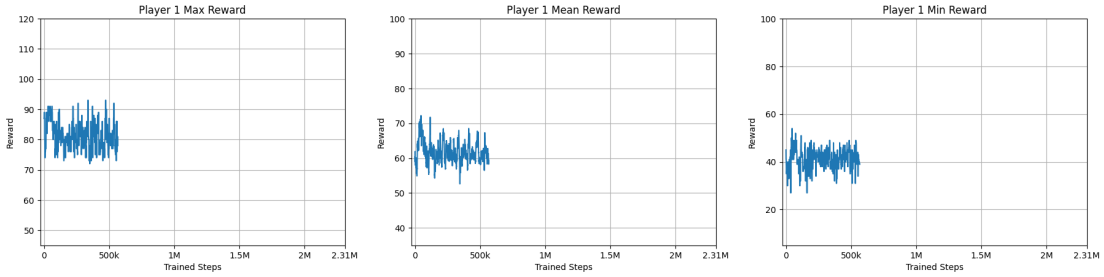


Figure E.121 DQN Player 1 Max, Mean and Min Scores

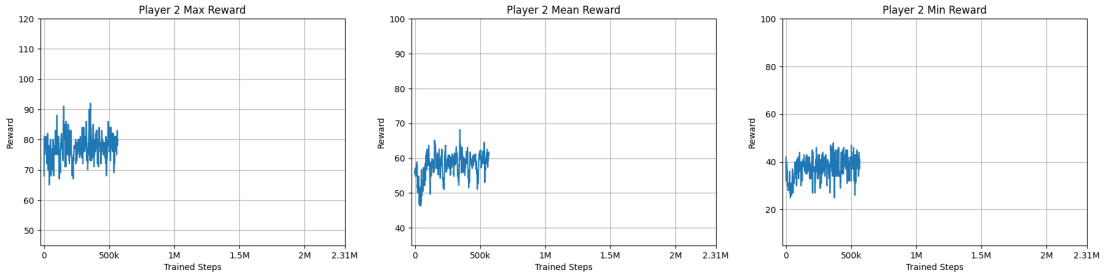


Figure E.122 DQN Player 2 Max, Mean and Min Scores

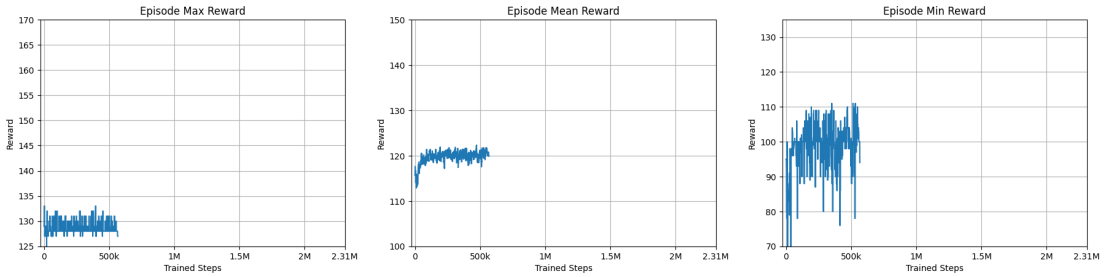


Figure E.123 DQN Episode Max, Mean and Min Scores

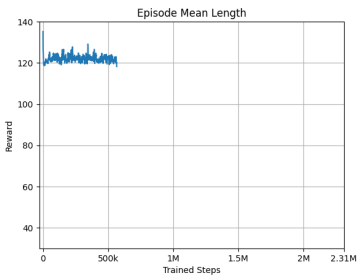


Figure E.124 DQN Episode Average Lengths

E.32 Hierarchical RL with Centralised Critic - DDQN

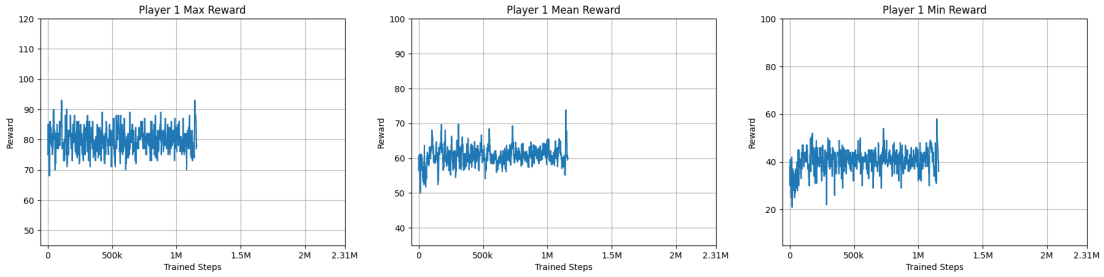


Figure E.125 DDQN Player 1 Max, Mean and Min Scores

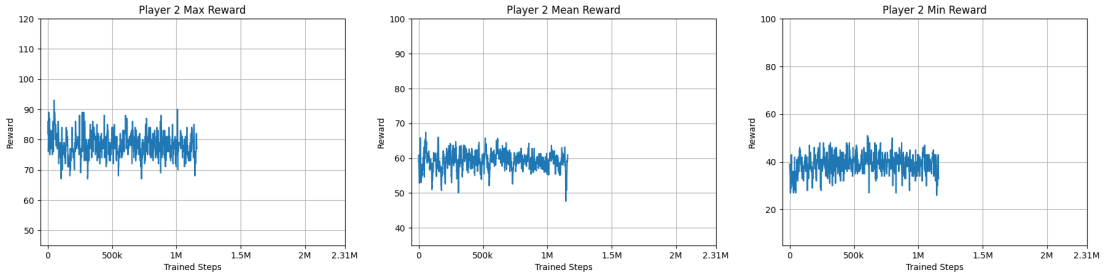


Figure E.126 DDQN Player 2 Max, Mean and Min Scores

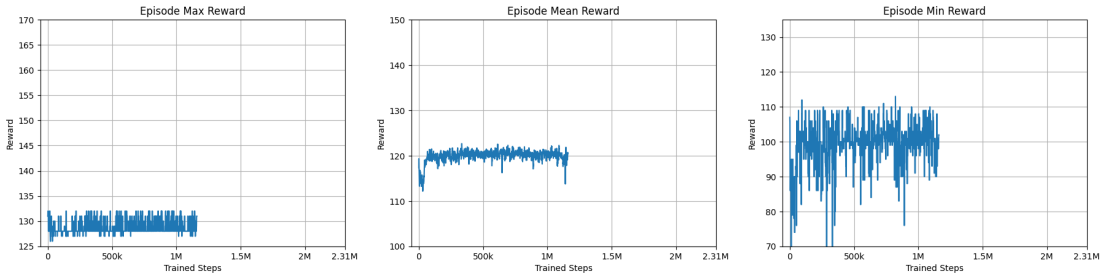


Figure E.127 DDQN Episode Max, Mean and Min Scores

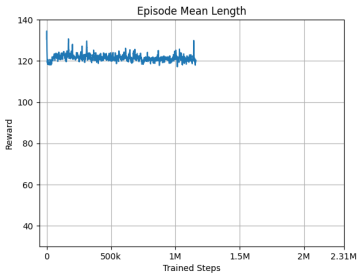


Figure E.128 DDQN Episode Average Lengths

Appendix F Action Selection Analysis Results

F.1 Initial Implementation with Action Masking

Table F.1 PPO Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 1G, 1Sv, 5Sk, 8C, 35R, 0oC M: 1Sk, 1Sp, 3L T: 1D, 2G, 1Sv, 6Sk, 2Sp, 4L, 6B3, 6B4, 5B5	Sold 5 Silk	Obtained the bonus token for selling 5 cards	Very good action, the player waited to sell cards to get the greatest number of bonus points.
P: 1D, 1Sv, 4Sk, 1L, 11C, 57R, 0oC M: 2G, 3Sk T: 1D, 1G, 6Sk, 2L, 6B3, 6B4, 5B5	Traded 1 Diamond and 1 Silver for 2 Gold	Obtained 2 Gold cards	Very good action, the player prioritised taking 2 Gold cards to be able to take last Gold token instead of waiting to potentially get another Diamond or Silver near the end of the game.
P: 2G, 4Sk, 1L, 11C, 57R, 0oC M: 1D, 1Sv, 3Sk T: 1D, 1G, 6Sk, 2L, 6B3, 6B4, 5B5	Sold 2 Gold	Took last Gold token and ended the game	Very good action, sold the last 2 Gold cards to take the last token, ended the game with more camel cards and won the game.
P: 3G, 2L, 2C, 10R, 2oC M: 1Sp, 4C T: 3D, 5G, 3Sv, 6Sk, 6Sp, 9L, 7B3, 6B4, 5B5	Sold 3 Gold	Took the first 3 Gold tokens and the bonus tokens for selling 3 cards	Very good action, collected and sold 3 Gold tokens to not only take the tokens worth the most points but also take the bonus token.
P: 1D, 1G, 1Sk, 2L, 0R, 2oC M: 2Sp, 3C T: 5D, 3G, 5Sv, 7Sk, 7Sp, 9L, 7B3, 6B4, 5B5	Took all Camel cards	Had more Camel cards than the opponent	Good action, the player decided to prioritise taking Camel cards to have more than the opponent.

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, oC: Opponent's Camel,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

Table F.2 A2C Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 3D, 1C, 62R, 7oC M: 1Sk, 2L, 2C T: 3D, 1G, 1Sv, 1Sp, 1L, 7B3, 6B4, 5B5	Sold 3 Diamond	Obtained the last 3 Diamond tokens and bonus tokens for selling 3 cards	Very good action, the player waited to sell 3 Diamonds to take the last 3 tokens and a bonus token.
P: 1D, 5C, 30R, 0oC M: 1D, 1Sk, 2Sp, 1L T: 5D, 1G, 1Sv, 3Sk, 5Sp, 4L, 7B3, 6B4, 5B5	Traded 2 Camel for 1 Diamond and 1 Silk	Obtained a Diamond whilst preventing the marketplace from being repopulated	Very good action, the player traded Camel cards instead of only taking the Diamond to prevent the opponent from getting a good card when the marketplace is re-populated. The player still had more Camel cards than the opponent.
P: 1G, 1Sk, 2L, 1C, 44R, 9oC M: 1G, 1Sv, 1Sk, 1L, 1C T: 1D, 1G, 1Sv, 3Sk, 4L, 7B3, 6B4, 5B5	Traded 1 Leather and 1 Camel for 1 Gold and 1 Silver	Took both Gold and Silver cards	Very good action, the player took both high value cards and traded out the Camel and and Leather cards instead of the Silk card or both Leather cards.
P: 1G, 6C, 59R, 0oC M: 1L, 4C T: 1G, 1Sv, 2Sp, 2L, 7B3, 6B4, 5B5	Took all Camel cards	Took all the Camel cards near the end of the game to secure the Camel token	Very good action, prevented the opponent from having more Camel cards at the end of the game and prioritised the Camel token over not re-populating the marketplace since only one token remained for two high value cards.
P: 1G, 6C, 49R, 1oC M: 1D, 1Sv, 1Sp, 2C T: 1D, 1G, 2Sp, 2L, 6B3, 6B4, 5B5	Traded 2 Camel for 1 Diamond and 1 Spice	Took a Diamond and Spice card and ignored the Silver card since no Silver tokens remained	Very good action, the player took the Diamond card to either be able to take the last Diamond token or prevent the opponent from taking it, whilst ignoring the Silver card given that no token remained.

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, oC: Opponent's Camel,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

Table F.3 DQN Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 7L, 58R, 8oC M: 4Sp, 1L T: 1D, 1G, 1Sv, 4Sp, 8L, 7B3, 6B4, 5B5	Sold 7 Leather	Took 7 Leather tokens and bonus token for selling 5 cards	Very good action, the player collected and sold 5 or more cards at once.
P: 1G, 2Sk, 1Sp, 1L, 0R, 0oC M: 1Sp, 4C T: 5D, 5G, 5Sv, 5Sk, 7Sp, 9L, 7B3, 6B4, 5B5	Sold 1 Spice	Took the first Spice token	Very good action, the player sold 1 Spice instead of 2 Silk since the other player had already taken the first 2 Silk tokens.
P: 1D, 1Sp, 1L, 74R, 6oC M: 1Sv, 1Sp, 1L, 2C T: 1Sv, 1Sk, 3Sp, 5L, 5B3, 5B4, 5B5	Traded 1 Diamond and 1 Spice for 1 Leather and 1 Silver	Took Silver and Leather	Very good action for discarding the Diamond card since no other token remained and for collecting another Leather card to sell more cards at once since more Leather tokens remained than Spice tokens.
P: 1D, 3G, 1Sv, 48R, 6oC M: 2Sv, 2L, 1C T: 3D, 3G, 5Sv, 2Sk, 3Sp, 5L, 7B3, 5B4, 5B5	Sold 3 Gold	Took last 3 Gold tokens and bonus token for selling 3 cards	Good action, did the right thing in not trading out Diamond and Gold cards to get the 2 Silver cards but could have taken 1 Silver card only and sold the Gold cards in another round.
P: 1Sv, 5C, 57R, 3oC M: 2G, 3C T: 1G, 1Sv, 1Sp, 1L, 7B3, 6B4, 4B5	Took all Camel cards	Took all the Camel cards near the end of the game to secure the Camel token	Very good action, prevented the opponent from having more Camel cards at the end of the game and prioritised the Camel token over taking 2 Gold cards for the last Gold token.

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, oC: Opponent's Camel,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

Table F.4 DDQN Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 1Sv, 1Sp, 5L, 5C, 51R, 6oC M: 1D, 1Sk, 1Sp, 2L T: 1D, 1G, 2Sp, 7L, 6B3, 6B4, 5B5	Sold 5 Leather cards	Obtained 5 Leather Tokens and bonus token for selling 5 cards	Very good action, the player collected and sold 5 cards at once to obtain the bonus token. Prioritised this action over taking Diamond card since only 1 token remained and game was near the end.
P: 3Sv, 1Sk, 1Sp, 3C, 27R, 2oC M: 1Sk, 1L, 3C T: 3D, 5G, 5Sv, 4Sk, 6Sp, 4L, 6B3, 6B4, 5B5	Sold 3 Silver	Obtained the first 3 Silver tokens and bonus token for selling 3 cards	Very good action, waited to sell 3 cards at once to obtain the bonus token.
P: 1Sv, 1Sp, 5C, 66R, 6oC M: 1G, 1Sk, 1Sp, 2L T: 1D, 1G, 2Sp, 2L, 6B3, 6B4, 4B5	Traded 2 Camel for 1 Gold and 1 Leather	Obtained Gold and Leather cards	Good action, the player took the Gold card to either stop the opponent from taking the last Gold token or to try to get it themselves. The player also took a Leather card potentially to be able prolong the game to get more points by selling the Leather and Spice cards individually.
P: 3C, 52R, 2oC M: 2Sp, 3C T: 1D, 5G, 2Sv, 1Sk, 5Sp, 4L, 5B3, 6B4, 5B5	Took all Camel cards	Prevented the opponent from having more Camel cards	Very good action, prioritised having more Camel cards rather than taking 2 Spice cards.
P: 1D, 1Sk, 3L, 0R, 2oC M: 1Sk, 1Sp, 3C T: 5D, 5G, 5Sv, 7Sk, 7Sp, 9L, 7B3, 6B4, 5B5 Start of the game	Sold 3 Leather	Took first 3 Leather tokens and bonus token of selling 3 cards	Very good, sold the highest combination at the start of the game.

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, oC: Opponent's Camel,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

F.2 Increased Action Space

Table F.5 PPO Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 2D, 1Sv, 2L, 2C, 23R, 4oC M: 1D, 2Sp, 2C T: 5D, 1G, 3Sv, 7Sk, 3Sp, 5L, 7B3, 6B4, 5B5	Took 1 Diamond	Collected 3 Diamond cards	Very good action, did not immediately sell 2 Diamonds or give the opponent the opportunity to take a Diamond card.
P: 2D, 2Sv, 1C, 0R, 0oC M: 2L, 3C T: 5D, 5G, 5Sv, 7Sk, 7Sp, 9L, 7B3, 6B4, 5B5	Sold 2 Diamond	Took the first 2 Diamond tokens	Very good, sold the highest combination at the start of the game and took the 2 highest valued tokens of the game.
P: 2D, 1Sv, 2Sp, 2C, 44R, 6oC M: 1Sp, 1L, 3C T: 2D, 1G, 2Sk, 2Sp, 1L, 4B3, 6B4, 5B5 Opponent had taken the last Diamond card	Sold 2 Diamond	Took the last 2 Diamond tokens	Very good action, upon seeing the opponent take the last Diamond card, the player sold the Diamond cards to take the last tokens.
P: 1D, 1G, 9C, 74R, 0oC M: 1D, 1G, 1Sp, 2C T: 1D, 3G, 1Sp, 3L, 7B3, 6B4, 5B5	Traded 3 Camel for 1 Diamond, 1 Gold and 1 Spice	Collected last 2 Diamond, 2 Gold and Spice cards	Very good action, took last Diamond card to be able to get last token as well as the Gold card. The player potentially took the Spice card to ensure that the opponent does not take it and sell it as that would end the game.
P: 1D, 1G, 1Sv, 1L, 1C, 11R, 0oC M: 1D, 1G, 3C T: 5D, 5G, 5Sv, 7Sk, 6Sp, 6L, 6B3, 6B4, 5B5	Traded 1 Leather and 1 Camel for 1 Diamond and 1 Gold	Collected 2 Diamond and 2 Gold cards	Very good action, the player prioritised taking both high value cards instead of keeping the Leather card since they only had 1 Camel.

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, oC: Opponent's Camel,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

Table F.6 A2C Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 1Sv, 1Sp, 20R, 6oC M: 1G, 1Sv, 1Sk, 1Sp, 1L T: 5D, 1G, 5Sv, 4Sk, 6Sp, 7L, 7B3, 6B4, 5B5	Took 1 Silver	Collected 2 Silver cards	Very good action, the player could not take both Gold and Silver cards, therefore they prioritised taking the Silver card given that 2 cards are needed to sell and they already had 1 Silver card.
P: 2D, 1Sv, 7C, 35R, 0oC M: 1D, 2Sk, 1L, 1C T: 5D, 1G, 3Sv, 4Sk, 4Sp, 4L, 6B3, 6B4, 5B5	Took 1 Diamond	Collected 3 Diamonds	Very good action, risked not taking the first two tokens to not give the opponent the opportunity to take the Diamond card.
P: 1G, 4Sv, 1Sk, 1Sp, 26R, 4oC M: 1L, 4C T: 3D, 3G, 5Sv, 4Sk, 6Sp, 9L, 7B3, 6B4, 5B5	Sold 4 Silver	Took 4 Silver tokens and a bonus token for selling 4 cards	Very good action, waited to collect and sell 4 cards at once, silver tokens are all of the same value, to not only obtain the Silver tokens but also a bonus token.
P: D, 1G, 1Sv, 1Sk, 1Sp, L, C, 62R, 8oC M: 2D, G, Sv, Sk, 1Sp, L, 2C T: 1D, 3G, 1Sv, 1Sk, 4Sp, 6L, 6B3, 5B4, 5B5	Traded 1 Silver and 1 Silk for 2 Diamond	Collected 2 Diamond	Very good action, since only one Silk, Silver and Diamond tokens remained, the player prioritised getting the Diamond token since it has a higher value than the Silk token and the player does not have another Silver card to get the Silver token.
P: 2Sv, 1Sp, 3C, 18R, 3oC M: 2Sk, 3C T: 3D, 3G, 5Sv, 7Sk, 6Sp, 8L, 7B3, 6B4, 5B5	Traded 1 Spice and 1 Camel for 2 Silk	Collected 2 Silk	Very good action, the player prioritised taking 2 Silk to get the first two Silk tokens instead of selling Spice since the first Spice token was already taken.

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, oC: Opponent's Camel,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

Table F.7 DQN Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 1D, 1G, Sv, 1Sk, Sp, 2L, C, 0R, 1oC M: D, G, Sv, 1Sk, Sp, L, 4C T: 5D, 5G, 5Sv, 7Sk, 7Sp, 9L, 7B3, 6B4, 5B5	Sold 2 Leather	Took the first 2 Leather tokens	Very good action, the player sold the combination of cards which would return the highest number of points
P: 1D, G, 1Sv, Sk, Sp, 1L, 4C, 60R, 4oC M: D, G, Sv, Sk, 3Sp, L, 2C T: 1D, G, 1Sv, Sk, 2Sp, 1L, 6B3, 5B4, 5B5	Took all Camel cards	Collected more Camel cards than the opponent	Good action, the player prioritised taking Camel cards near end of game to receive the Camel token
P: 1D, 2G, 2Sv, Sk, Sp, L, C, 11R, 2oC M: D, G, Sv, 1Sk, Sp, 1L, 3C T: 3D, 5G, 3Sv, 7Sk, 5Sp, 7L, 7B3, 6B4, 5B5	Sold 2 Gold	Took the first 2 Gold tokens	Very good action, the player immediately took the first 2 Gold tokens and sold the card combination returning the highest value
P: 1D, G, 2Sv, Sk, Sp, 4L, 1C, 38R, 7oC M: 1D, G, Sv, 1Sk, 3Sp, L, C T: 1D, G, 3Sv, 2Sk, 5Sp, 5L, 7B3, 6B4, 5B5	Sold 4 Leather	Took 4 Leather tokens and bonus token for selling 4 cards	Very good action, collected 4 cards of the same type to get the bonus points. Could not have collected the Diamond card as it would have exceeded the hand's capacity and prioritised selling 4 Leather instead of trading to collect it as only one Diamond token remained
P: D, 1G, 2Sv, Sk, Sp, 2L, 6C, 59R, 3oC M: D, 1G, Sv, 2Sk, Sp, L, 2C T: D, 1G, 1Sv, 4Sk, Sp, 1L, 7B3, 5B4, 5B5	Sold 2 Silver	Took the last Silver token and ended the game	Very good action, the player ended the game by taking the last Silver token whilst having more Camel cards and ended up winning

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, oC: Opponent's Camel,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

Table F.8 DDQN Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 2D, 1G, 1Sv, Sk, Sp, 1L, C, 0R, 2oC M: 1D, G, Sv, Sk, Sp, L, 4C T: 5D, 5G, 5Sv, 7Sk, 6Sp, 9L, 7B3, 6B4, 5B5	Took 1 Diamond	Collected 3 Diamond	Very good action, the player prioritised not giving the opponent the opportunity to take the Diamond instead of immediately selling
P: D, 2G, 1Sv, Sk, Sp, L, 11C, 78R, 0oC M: D, G, Sv, 1Sk, 3Sp, 1L, C T: D, 2G, 1Sv, 2Sk, 2Sp, L, 6B3, 4B4, 5B5	Traded 4 Camel for 3 Spice and 1 Silk	Collected 3 Spice and 1 Silk	Very good action, the player still had more Camel cards than the opponent could get and did not take Leather since no tokens remained. Even though only 2 Spice tokens remained, the player still took 3 to get the bonus token
P: 1G, 4L, 8C, 52R, 0oC M: 1D, 1Sv, 3C T: 1D, 1G, 1Sv, 3L, 7B3, 5B4, 5B5	Sold 4 Leather	Took last 3 Leather tokens and bonus token for selling 4 cards	Very good action, sold 4 cards to get the bonus token and ended the game with more Camel cards
P: 3D, G, 1Sv, Sk, Sp, 3L, 11C, 45R, 0oC M: D, 1G, Sv, 3Sk, 1Sp, L, C T: 3D, G, 1Sv, 2Sk, 2Sp, 3L, 6B3, 6B4, 5B5	Sold 3 Diamond	Took last 3 Diamond tokens and bonus token for selling 3 cards	Very good action, waited to sell 3 Diamond to obtain last 3 tokens and ignored the Gold card since no Gold tokens remained
P: 1D, G, 1Sv, Sk, 1Sp, 1L, C, 61R, 11oC M: D, 1G, Sv, 3Sk, 1Sp, L, C T: D, G, 1Sv, 2Sk, 2Sp, 3L, 5B3, 6B4, 5B5	Traded 1 Diamond and 1 Spice for 2 Silk	Collected 2 Silk	Very good action as no Diamond tokens remained and by taking 2 Silk, the player prevented the opponent from taking and selling 3 Silk to get a bonus token

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, oC: Opponent's Camel,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

F.3 Hyperparameter Tuning

Table F.9 PPO Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 3G, 1Sv, 1L, 0R, 2oC M: 2L, 3C T: 5D, 5G, 5Sv, 7Sk, 7Sp, 9L, 7B3, 6B4, 5B5	Sold 3 Gold	Took the first 3 Gold tokens and a bonus token for selling 3 cards	Very good action, immediately sold the 3 Gold cards to get the first Gold tokens
P: 1D, G, Sv, Sk, 2Sp, 2L, 3C, 42R, 5oC M: 1D, 1G, Sv, Sk, Sp, 3L, C T: 3D, 2G, 2Sv, 3Sk, 4Sp, 6L, 2B3, 6B4, 5B5	Traded 2 Camel for 1 Diamond and 1 Gold	Collected 2 Diamond and 1 Gold	Very good action, the player took all the high-value cards from the marketplace to not only be able to sell the Diamond cards but not give the opponent the opportunity to take and sell Gold
P: 1D, 4Sk, 1L, 16R, 1oC M: 5C T: 3D, 3G, 2Sv, 7Sk, 7Sp, 9L, 6B3, 6B4, 5B5	Sold 4 Silk	Took first 4 Silk tokens and bonus token for selling 4 cards	Very good action, waited to sell 4 Silk cards at once for the bonus token
P: D, G, Sv, 2Sk, Sp, 2L, 3C, 34R, 4oC M: 1D, G, 2Sv, Sk, Sp, 2L, C T: 3D, 3G, 2Sv, 3Sk, 3Sp, 9L, 6B3, 4B4, 5B5	Traded 3 Camel for 1 Diamond and 2 Silver	Collected 1 Diamond and 2 Silver	Very good action, took all the high-value cards from the marketplace
P: 2Sp, 5C, 36R, 2oC M: 1D, 1G, 1Sk, 2L T: 3D, 1G, 1Sv, 5Sk, 4Sp, 6L, 6B3, 6B4, 5B5	Traded 5 Camel for 1 Diamond, 1 Gold, 1 Silk and 2 Leather	Took all the Goods cards from the marketplace	Very good and interesting action, since the game was not close to ending and there were a lot of Camel cards remaining in the deck, it was smart of the player to take all the Goods cards to be able to sell

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, oC: Opponent's Camel,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

Table F.10 A2C Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 1Sk, 1Sp, 1L, 4C, 24R, 5oC M: 1D, 2Sk, 1Sp, 1L T: 3D, 5G, 3Sv, 6Sk, 5Sp, 9L, 7B3, 6B4, 5B5	Traded 3 Camel for 1 Diamond and 2 Silk	Collected 1 Diamond and 3 Silk	Very good action, the player not only took the Diamond card but also prioritised getting 2 Silk to have a set of 3
P: 1G, 1Sv, 5C, 13R, 1oC M: 1Sp, 1L, 3C T: 3D, 5G, 3Sv, 6Sk, 5Sp, 9L, 7B3, 6B4, 5B5 Opponent had just took 3 Goods cards and had 6 cards in hand	Took all Camel cards	Collected more Camel cards	Very good action, the player obtained more than half the available Camel cards without a great risk that the opponent will be able to take high-value cards from the marketplace
P: 1D, 1Sv, 2Sk, 1Sp, 1L, 4C, 25R, 4oC M: 3L, 2C T: 1D, 3G, 3Sv, 3Sk, 2Sp, 9L, 5B3, 6B4, 5B5	Traded 2 Silk and 1 Camel for 3 Leather	Collected 4 Leather	Very good action, the player prioritised a set of 4 Leather instead of 2 Silk not only for the bonus token but the remaining Leather tokens had a higher value than the remaining Silk tokens
P: 1D, 1Sv, 1Sp, 4L, 3C, 25R, 7oC M: D, G, 1Sv, 2Sk, Sp, 2L, C T: 1D, 3G, 3Sv, 3Sk, 2Sp, 9L, 5B3, 6B4, 5B5	Sold 4 Leather	Took first 4 Leather tokens and bonus token for selling 4 cards	Very good action, the player prioritised selling 4 Leather to obtain the first 4 Leather tokens and a bonus token instead of taking the Silver card by trading out 1 Spice and Leather.
P: 1D, 1Sp, 2L, 6C, 0R, 0oC M: 1D, 1G, 1Sk, 1L, 1C T: 5D, 5G, 5Sv, 7Sk, 4Sp, 9L, 6B3, 6B4, 5B5	Traded 2 Camel for 1 Diamond and 1 Gold	Collected 2 Diamond and 1 Gold	Very good action, took both high-value cards, could have also taken 1 Leather card to get a set of 3 but prioritised keeping more Camel cards

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, oC: Opponent's Camel,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

Table F.11 DQN Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 5C, 37R, 1oC M: 1D, 1Sk, 2Sp, 1L T: 3D, 3G, 5Sv, 4Sk, 5Sp, 7L, 7B3, 6B4, 5B5	Traded 4 Camel for 1 Diamond, 2 Spice and 1 Leather	Collected 1 Diamond, 2 Spice and 1 Leather cards	Good and interesting action, the player not only took the high-value card but also some low-value cards in exchange for Camel cards given that the game was in its early stages and the player did not have any Goods cards in their hand
P: 1Sv, 4L, 5C, 43R, 0oC M: 2Sk, 1Sp, 2C T: 3D, 1Sv, 1Sk, 1Sp, 4L, 6B3, 6B4, 5B5	Sold 4 Leather	Took last 4 Leather tokens and a bonus token for selling 4 cards	Very good action, waited to sell multiple cards at once and sold to take the last 4 tokens apart from a bonus token
P: 1D, 1G, 1Sk, 1L, 1C, 49R, 10oC M: 4Sp, 1L T: 1D, 1G, 1Sv, 4Sp, 4L, 7B3, 6B4, 5B5	Traded 1 Gold, 1 Silk and 1 Camel for 3 Spice	Collected 3 Spice	Good action, prioritised having 3 Spice instead of the last Camel card, since the opponent had nearly all, the Silk card, as no tokens remained, and the Gold instead of Leather as only one Gold token remained whilst 4 Leather tokens remained
P: 1D, 3C, 39R, 4oC M: 1Sp, 1L, 3C T: 1D, 1G, 3Sv, 2Sk, 3Sp, 4L, 6B3, 6B4, 5B5	Took all Camel cards	Collected more Camel cards than the opponent	Good action, the player prioritised taking Camel cards to compete with the opponent by having more than half the game Camel cards
P: 1D, 3Sp, 1L, 49R, 10oC M: 1Sk, 2Sp, 1L, 1C T: 1D, 1G, 1Sv, 4Sp, 4L, 7B3, 6B4, 5B5	Sold 3 Spice	Took 3 Spice tokens and a bonus token for selling 3 cards	Good action, the player could have taken 1 other Spice card however that would risk the opponent taking 1 of the Spice tokens, therefore the player prioritised immediately selling 3 Spice

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, oC: Opponent's Camel,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

Table F.12 DDQN Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 3D, 1Sp, 1L, 6C, 12R, 0oC M: 1Sv, 1Sp, 2L, 1C T: 5D, 1G, 3Sv, 5Sk, 5Sp, 7L, 7B3, 6B4, 5B5	Sold 3 Diamond	Took first 3 Diamond tokens and a bonus token for selling 3 cards	Very good action, despite not taking the Silver card, the player prioritised getting the first 3 Diamond tokens, which have the highest number of points in the game, as well as a bonus token
P: 2D, 4L, 6C, 38R, 3oC M: 2Sk, 2L, 1C T: 2D, 1G, 1Sv, 3Sk, 2Sp, 7L, 6B3, 6B4, 5B5	Sold 4 Leather	Took 4 Leather tokens and a bonus token for selling 4 cards	Very good action for waiting to collect and sell 4 cards, however could have taken another Leather card to sell 5 at once. The player may have immediately sold to avoid having a full hand or to leave 3 Leather tokens to sell a set of 3 cards
P: 2D, 10C, 65R, 0oC M: 2G, 2Sk, 1C T: 1D, 1G, 1Sv, 2Sk, 2Sp, 7B3, 4B4, 5B5	Took 1 Gold	Collected 1 Gold but left 1 Gold in the marketplace	Questionable action, good for the player to take Gold instead of selling last 2 Diamond cards but could have taken both Gold cards instead of 1 only
P: 2D, 1Sv, 1Sp, 28R, 4oC M: 2L, 3C T: 5D, 3G, 5Sv, 3Sk, 5Sp, 5L, 7B3, 6B4, 5B5	Took all Camel cards	Collected Camel cards to compete with opponent	Very good action, prioritised taking Camel cards to prevent the opponent from having more than half, making it unlikely for the player to take the Camel token
P: 2G, 1Sv, 1Sk, 1Sp, 0R, 1oC M: 1L, 4C T: 5D, 5G, 5Sv, 7Sk, 7Sp, 9L, 7B3, 6B4, 5B5 Start of game	Sold 2 Gold	Took first 2 Gold tokens	Very good action, especially at the start of the game, prioritised taking the first 2 Gold tokens to not risk the opponent taking them

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, oC: Opponent's Camel,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

F.4 Increased Agent Observation

Table F.13 PPO Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 2L, 1C, 12R O: 1C, 14R M: 1Sk, 1L, 3C T: 5D, 3G, 5Sv, 4Sk, 7Sp, 9L, 6B3, 6B4, 5B5	Took all Camel cards	Collected more Camel cards than the opponent	Very good action, prioritised having more Camels instead of getting first Leather tokens as opponent had no Leather cards
P: 1Sk, 2Sp, 5C, 24R O: 2D, 1G, 1Sv, 1Sk, 1L, 1C, 14R M: 1Sv, 1Sp, 2L, 1C T: 5D, 3G, 5Sv, 4Sk, 7Sp, 6L, 5B3, 6B4, 5B5	Traded 1 Silk and 3 Camel for 1 Silver, 1 Spice and 2 Leather	Collected 1 Silver, 3 Spice and 2 Leather	Very good action, took Silver and traded out Silk for Spice and Leather, as the remaining Silk tokens had less value than the other tokens
P: 1D, 2Sk, 1Sp, 3L, 3C, 30R O: 1G, 2Sv, 5C, 29R M: 2G, 1Sv, 1Sp, 1C T: 3D, 5G, 3Sv, 6Sk, 3Sp, 6L, 6B3, 5B4, 5B5	Traded 1 Silk and 2 Leather for 2 Gold and 1 Silver	Collected 2 Gold and 1 Silver	Very good action, took all high value cards and prevented the opponent from selling Gold or collecting a set of 3 Silver, chose to trade out Leather and Silk as the opponent does not have these cards and to keep Camel cards
P: 1D, 1G, 1Sv, 1Sk, 5C, 0R O: 2Sp, 0C, 14R M: 1D, 1Sk, 3L T: 3D, 5G, 5Sv, 7Sk, 7Sp, 9L, 7B3, 6B4, 5B5	Traded 1 Silk and 3 Camel for 1 Diamond and 3 Leather	Collected 2 Diamond and 3 Leather	Very good action, took Diamond card to be able to sell and a set of 3 cards
P: 5C, 69R O: 1Sv, 1L, 6C, 64R M: 2D, 1Sk, 1Sp, 1L T: 3D, 1Sk, 1Sp, 1L, 3B3, 4B4, 5B5	Traded 4 Camel for 2 Diamond, 1 Silk and 1 Leather	Collected 2 Diamond, 1 Silk and 1 Leather	Very good action, took Goods cards to try to have more control and sell 2 Diamond before the game ends especially as the opponent had more Camel cards

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, O: Opponent,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

Table F.14 A2C Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 1G, 1Sv, 1L, 6C, 0R O: 1G, 1C, 8R M: 1Sk, 1Sp, 3L T: 5D, 5G, 5Sv, 5Sk, 7Sp, 9L, 7B3, 6B4, 5B5	Traded 4 Camel for 1 Spice and 3 Leather	Collected 1 Spice and 4 Leather	Very good action, got many Goods cards instead of Camel cards at the early stages of the game and prioritised having a set of 4 cards instead of immediately selling
P: 2Sk, 1Sp, 2L, 9C, 46R O: 1G, 1Sv, 1Sp, 1C, 65R M: 1Sv, 1Sp, 3L T: 1D, 1G, 1Sv, 1Sk, 4Sp, 5L, 6B3, 5B4, 5B5	Traded 1 Silk, 1 Spice and 2 Camel for 1 Silver and 3 Leather	Collected 1 Silver and 5 Leather	Very good action, took the Silver card to prevent the opponent from taking the last Silver token and prioritised getting a set of 5 cards whilst also having more Camel cards
P: 1Sv, 1Sk, 5L, 7C, 46R O: 1G, 1Sv, 1Sp, 3C, 65R M: 1D, 1Sk, 2Sp, 1C T: 1D, 1G, 1Sv, 1Sk, 4Sp, 5L, 6B3, 5B4, 5B5	Sold 5 Leather	Took last 5 Leather tokens and bonus token of selling 5 cards	Very good action, since the opponent did not have any Leather cards, the player waited to collect and sell 5 cards
P: 2Sv, 1Sk, 2Sp, 5C, 41R O: 2Sv, 2C, 54R M: 1Sk, 2L, 2C T: 1D, 3G, 3Sv, 4Sk, 2Sp, 3L, 3B3, 6B4, 5B5	Sold 2 Silver	Took 2 Silver tokens	Very good action, the player had postponed selling the cards to collect and sell more than 2 cards but as soon as the opponent took 2 Silver cards, the player immediately sold to take 2 out of the last 3 tokens instead of risking that the opponent takes them
P: 1D, 2Sk, 1Sp, 1L, 3C, 12R O: 1D, 1G, 2L, 4C, 10R M: 2D, 1Sk, 2Sp T: 5D, 3G, 3Sv, 7Sk, 7Sp, 9L, 7B3, 6B4, 5B5	Traded 2 Camel for 2 Diamond	Collected 3 Diamond	Very good action, prioritised taking both Diamond cards to get a set of 3 and to prevent the opponent from being able to take Diamond tokens

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, O: Opponent,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

Table F.15 DQN Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 2Sk, 2Sp, 1C, 0R O: 1C, 0R M: 1D, 1Sk, 3C T: 5D, 5G, 5Sv, 7Sk, 7Sp, 9L, 7B3, 6B4, 5B5 Start of game	Took 1 Diamond	Collected 1 Diamond	Very good action, prioritised taking the Diamond card instead of selling to prevent the opponent from taking it
P: 1D, 1G, 2L, 1C, 0R O: 0C, 0R M: 1Sk, 4C T: 5D, 5G, 5Sv, 7Sk, 7Sp, 9L, 7B3, 6B4, 5B5 Start of game	Took 1 Silk	Collected 1 Silk	Good action, prioritised taking the Silk card instead of immediately selling the Leather cards as the Silk token is worth more points
P: 2Sk, 48R O: 1Sp, 1L, 4C, 39R M: 1Sp, 1L, 3C T: 1D, 1G, 3Sv, 3Sk, 5Sp, 5L, 7B3, 6B4, 5B5	Took all Camel cards	Prevented the opponent from having more than half the Camel cards	Good action, prioritised the Camel cards to prevent the opponent from having too many Camel cards which would make it unlikely for the player to get the Camel token
P: 1Sv, 1Sp, 4L, 4C, 44R O: 1G, 1Sv, 3Sk, 5C, 49R M: 1G, 1Sk, 2Sp, 1L T: 1G, 3Sv, 3Sk, 5Sp, 4L, 7B3, 6B4, 5B5	Sold 4 Leather	Took last 4 Leather cards and a bonus token of selling 4 cards	Good action for collecting and selling 4 cards, it would have been ideal to postpone the action to take the Gold card to prevent the opponent from taking the last Gold token, however this action had more points than the last Gold token
P: 1D, 1Sv, 1Sp, 4C, 62R O: 1Sv, 1Sk, 1Sp, 6C, 50R M: 1D, 4L T: 1D, 1G, 1Sv, 2Sk, 2Sp, 4L, 7B3, 5B4, 5B5	Traded 1 Silver and 3 Camel for 4 Leather	Collected 4 Leather	Questionable action, prioritised giving Silver away for a set of 4 cards to not have a full hand or use all Camel cards however it would have been ideal to give Spice away instead of Silver as the opponent had the last Silver card needed for the last Silver token

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, O: Opponent,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

Table F.16 DDQN Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 1G, 1Sv, 1Sk, 1L, 1C, 0R O: 1C, 5R M: 1G, 1Sk, 3C T: 5D, 5G, 5Sv, 7Sk, 6Sp, 9L, 7B3, 6B4, 5B5	Took 1 Gold	Collected 2 Gold	Very good action, prioritised taking Gold to be able to sell instead of immediately selling other cards
P: 2L, 1C, 19R O: 1Sv, 1C, 12R M: 1Sk, 4C T: 3D, 3G, 5Sv, 7Sk, 6Sp, 9L, 7B3, 6B4, 5B5	Took all Camel cards	Collected 5 Camels	Very good action, prioritised taking 4 Camel instead of selling Leather as the opponent did not have any Leather cards and by doing so the player collected much more Camel cards to trade out or to get the Camel token
P: 1G, 1Sv, 2L, 5C, 19R O: 1D, 1C, 27R M: 3Sk, 2L T: 3D, 3G, 3Sv, 6Sk, 6Sp, 9L, 7B3, 6B4, 5B5	Traded 2 Camel for 2 Silk	Collected 2 Silk	Questionable action, would have been more ideal to collect and sell 4 Leather however the next 2 Silk tokens still have a lot of points
P: 1D, 1G, 1Sv, 2L, 7C, 29R O: 1G, 2L, 1C, 50R M: 4Sp, 1L T: 1D, 3G, 3Sv, 2Sk, 6Sp, 5L, 6B3, 6B4, 5B5	Traded 2 Camel for 2 Spice	Collected 2 Spice	Very good action, prioritised keeping all Goods cards and prevented the opponent from getting a set of 4 cards by taking as much Spice cards as possible and the remaining Spice tokens are worth more than the remaining Leather tokens
P: 1D, 1G, 1Sv, 4L, 6C, 42R O: 1Sv, 1Sk, 1L, 5C, 45R M: 4Sk, 1Sp T: 3D, 1G, 1Sv, 5Sk, 4Sp, 6L, 6B3, 6B4, 5B5	Sold 4 Leather	Took 4 Leather tokens and a bonus token of selling 4 cards	Good action for selling the 4 Leather cards but could have traded them out for 4 Silk since Silk tokens are worth slightly more, however opponent could have sold Silk prior to player and taken the higher value token

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, O: Opponent,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

F.5 Full Opponent Observation

Table F.17 PPO Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 2Sv, 1L, 5C, 14R O: 3G, 1Sv, 1Sk, 1L, 0C, 8R M: 2Sk, 3Sp T: 3D, 5G, 5Sv, 5Sk, 7Sp, 9L, 7B3, 6B4, 5B5	Traded 4 Camel for 2 Silk and 2 Spice	Collected 2 Silk and 2 Spice	Very good action, the player prioritised getting multiple Goods cards at the early stages of the game rather than keeping Camel cards
P: 1G, 1Sv, 1Sk, 1L, 3C, 8R O: 2D, 1G, 2Sv, 1Sk, 1L, 2C, 0R M: 1Sp, 3L, 1C T: 5D, 5G, 5Sv, 7Sk, 5Sp, 9L, 7B3, 6B4, 5B5 Start of game	Traded 3 Camel for 3 Leather	Collected 4 Leather	Very good action, the player prioritised getting a set of 4 cards rather than selling immediately whilst keeping all the other Goods cards
P: 1Sk, 2L, 51R O: 2G, 1Sv, 6C, 55R M: 3Sp, 2C T: 1D, 1G, 3Sv, 2Sk, 2Sp, 1L, 4B3, 6B4, 5B5	Traded 1 Silk and 2 Leather for 3 Spice	Collected 3 Spice	Very good action, the player prioritised getting a set of 3 cards to sell and get the bonus token apart from the normal tokens
P: 1G, 4Sp, 2C, 33R O: 1G, 1Sv, 2L, 2C, 53R M: 5C T: 1G, 3Sv, 4Sk, 6Sp, 7L, 6B3, 6B4, 5B5	Sold 4 Spice	Took 4 Spice tokens and a bonus token of selling 4 cards	Very good action, the player chose to sell a set of 4 cards instead of taking all the Camel cards which would cause the
P: 1D, 2L, 5C, 49R O: 2G, 1Sv, 4C, 62R M: 1Sv, 3Sk, 1Sp T: 1G, 3Sv, 4Sk, 2Sp, 3L, 6B3, 4B4, 5B5	Traded 1 Diamond and 3 Camel for 1 Silver and 3 Silk	Collected 1 Silver and 3 Silk	Very good action, traded out the Diamond card when no Diamond tokens remained, took a set of 3 cards as well as the high value cards

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, O: Opponent,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

Table F.18 A2C Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 1G, 2Sv, 2Sk, 0R O: 1D, 2Sk, 1Sp, 1L, 0R M: 1G, 1Sk, 3C T: 5D, 5G, 5Sv, 7Sk, 7Sp, 9L, 7B3, 6B4, 5B5 Start of game	Took 1 Gold	Collected 2 Gold	Very good action, prioritised getting the Gold card instead of selling the other cards
P: 1D, 1Sp, 1L, 7C, 8R O: 27R M: 3Sk, 2L T: 5D, 3G, 3Sv, 3Sk, 7Sp, 9L, 7B3, 6B4, 5B5	Traded 3 Camel for 3 Silk	Collected 3 Silk	Very good action, whilst taking and selling Leather would have provided higher rewards, by taking the 3 Silk cards, the player prevented the opponent from having a set of 3 cards and since the opponent did not have any Leather cards, the player still could take the first Leather token
P: 1G, 1Sk, 4C, 56R O: 5C, 64R M: 1D, 1Sp, 3L T: 1D, 1G, 1Sv, 1Sp, 3L, 2B3, 6B4, 5B5	Traded 1 Silk and 2 Camel for 3 Leather	Collected 3 Leather	Very good action, traded out Silk card since no tokens remained and took a set of 3 cards
P: 3G, 1Sk, 1C, 0R O: 1D, 2L, 1C, 5R M: 1Sp, 1L, 3C T: 5D, 5G, 5Sv, 6Sk, 7Sp, 9L, 7B3, 6B4, 5B5 Start of game	Sold 2 Gold	Took the first 2 Gold tokens	Very good action, sold only 2 to take the first 2 Gold tokens which are worth the most and kept the other card to make it easier to get the other tokens
P: 1G, 2L, 7C, 19R O: 2Sv, 1Sk, 2Sp, 1L, 22R M: 1Sk, 1Sp, 3L T: 3D, 1G, 5Sv, 6Sk, 7Sp, 9L, 7B3, 6B4, 5B5	Traded 3 Camel for 3 Leather	Collected 5 Leather	Very good action, prioritised getting a set of 5 cards over taking the first Leather token, whilst still having more Camel cards than the opponent

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, O: Opponent,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

Table F.19 DQN Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 1G, 5L, 1C, 55R O: 3D, 1Sv, 1Sp, 5C, 35R M: 1Sp, 1L, 3C T: 3D, 1G, 1Sv, 3Sp, 8L, 7B3, 6B4, 5B5	Sold 5 Leather	Took 5 Leather tokens and a bonus token of selling 5 cards	Very good action, the player waited to collect and sell 5 cards at once
P: 1D, 1Sv, 1Sp, 5C, 45R O: 1G, 1L, 1C, 71R M: 2Sp, 3C T: 1D, 1G, 1Sv, 3Sp, 3L, 7B3, 6B4, 4B5	Traded 1 Diamond and 1 Camel for 2 Spice	Collected 3 Spice but removed 1 Diamond	Good action, the opponent does not have the other Diamond card to take the last Diamond token and selling a set of 3 Spice may provide more points than the last Diamond token
P: 1Sv, 1Sp, 1L, 1C, 12R O: 1D, 1C, 19R M: 1Sk, 1L, 3C T: 3D, 3G, 5Sv, 7Sk, 6Sp, 9L, 7B3, 6B4, 5B5	Sold 1 Leather	Prevented repopulating the marketplace	Very good action, the Leather token had more points than the Spice token and the player chose to sell to prevent re-populating the marketplace with a potentially good card for the opponent, especially since the opponent is restricted to take cards
P: 1Sv, 2L, 6C, 25R O: 1G, 1Sv, 1C, 30R M: 3Sk, 2L T: 3D, 1G, 5Sv, 4Sk, 5Sp, 9L, 7B3, 6B4, 5B5	Traded 2 Camel for 2 Leather	Collected 4 Leather	Very good, prioritised getting 4 Leather instead of 3 Silk as the Leather tokens had more value and the player kept more Camel cards than the opponent
P: 1Sk, 3C, 59R O: 1D, 4L, 7C, 58R M: 1D, 3Sp, 1L T: 1D, 1Sv, 3Sp, 5L, 7B3, 5B4, 5B5	Traded 3 Camel for 3 Spice	Collected 3 Spice	Good action for collecting a set of 3 cards but it would have been ideal for the player to also trade out the Silk card, since there weren't any tokens left, and take the last Diamond card to prevent the opponent from taking the last Diamond token

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, O: Opponent,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

Table F.20 DDQN Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 1D, 1G, 2L, 5C, 54R O: 1G, 5C, 53R M: 2Sk, 3Sp T: 1D, 1G, 2Sk, 4Sp, 5L, 7B3, 6B4, 5B5	Traded 1 Diamond and 1 Camel for 2 Spice	Collected 2 Spice	Questionable action, could have taken 1 Spice only, but it was good of the player to trade out Diamond instead of Gold since the opponent has another Gold and would have been able to take the last Gold token
P: 4L, 3C, 18R O: 3D, 1Sv, 1L, 17R M: 1Sk, 1Sp, 3C T: 5D, 3G, 3Sv, 3Sk, 7Sp, 9L, 7B3, 6B4, 5B5	Sold 4 Leather	Took the first 4 Leather tokens and a bonus token of selling 4 cards	Very good action, the player sold a set of 4 instead of waiting to collect and sell 5 cards to not miss out on the first Leather token, which costs a lot of points, as the opponent obtained a Leather card
P: 1G, 2L, 8C, 37R O: 1Sp, 1L, 61R M: 1Sk, 4Sp T: 1D, 1G, 3Sk, 7Sp, 5L, 7B3, 5B4, 5B5	Traded 1 Leather and 3 Camel for 4 Spice	Collected 4 Spice	Very good action, the player collected a set of 4 cards whilst keeping a greater number of Camel cards than the opponent could get
P: 1D, 1G, 2C, 10R O: 1Sv, 2Sp, 1L, 2C, 12R M: 3L, 2C T: 5D, 3G, 3Sv, 7Sk, 7Sp, 9L, 7B3, 6B4, 5B5	Traded 2 Camel for 2 Leather	Collected 2 Leather	Very good action, prioritised getting as much Leather cards as possible at the early stage of the game, without trading out Diamond or Gold, and prevented the opponent from getting a set of 4 cards
P: 1G, 1Sp, 9C, 60R O: 1D, 1G, 1Sv, 2C, 66R M: 1Sv, 3Sk, 1L T: 1D, 1G, 1Sv, 2Sk, 2Sp, 7B3, 3B4, 5B5	Traded 4 Camel for 1 Silver and 3 Silk	Collected 3 Silk and 1 Silver	Very good action, collected a set of 3 cards and took the Silver card to prevent the opponent from getting the last Silver token, whilst still having a lot of Camel cards

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, O: Opponent,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

F.6 Policy Cloning during Training

Table F.21 PPO Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 1G, 2Sv, 3C, 67R, 5oC M: 2D, 3C T: 1D, 2Sv, 2Sk, 3L, 2B3, 5B4, 5B5	Traded 1 Gold and 1 Camel for 2 Diamond	Collected last 2 Diamond	Very good action, took the last 2 Diamond cards to take the last Diamond token and gave out the Gold card as no Gold tokens remained
P: 1D, 2Sk, 1Sp, 1L, 5C, 39R, 5oC M: 1Sp, 4L T: 1D, 3G, 2Sv, 3Sk, 2Sp, 5L, 5B3, 5B4, 5B5	Traded 2 Silk and 2 Camel for 4 Leather	Collected 5 Leather	Very good action, prioritised getting a set of 5 cards, especially since the remaining Silk and Leather tokens had the same value
P: D, G, Sv, 1Sk, Sp, 1L, 5C, 8R, 0oC M: D, G, Sv, 1Sk, 1Sp, 2L, 1C T: 5D, 2G, 5Sv, 7Sk, 5Sp, 9L, 6B3, 6B4, 5B5 Early stages of the game	Traded 4 Camel for 1 Silk, 1 Spice and 2 Leather	Collected 3 Leather, 2 Silk and 1 Spice	Very good and interesting action, prioritised taking all the Goods cards in the early stages of the game to collect sets of cards and forced the opponent to either take all the Camel cards and re-populate the entire marketplace or sell low number of cards
P: 1G, 5L, 9C, 56R, 0oC M: 3Sp, 2C T: 1D, 1G, 4Sp, 5L, 4B3, 5B4, 5B5	Sold 5 Leather	Took the last 5 Leather tokens and a bonus token for selling 5 cards	Very good action, waited to sell 5 cards at once to get the highest valued bonus token
P: 2Sk, 55R, 10oC M: 2D, 1G, 1Sv, 1Sp T: 3D, 2G, 2Sv, 6Sk, 2Sp, 3L, 2B3, 6B4, 5B5	Traded 2 Silk for 2 Diamond	Collected 2 Diamond	Very good action, prioritised giving out 2 Silk for 2 Diamond and opted to take a set of 2 Diamond instead of 2 high-value cards of different types as unable to take all

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, oC: Opponent's Camel,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

Table F.22 A2C Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 1D, 1G, 1Sp, 2L, 9C, 57R, 1oC M: 1D, 1Sk, 2Sp, 1C T: 1D, 1G, 2Sk, 3Sp, 6L, 5B3, 5B4, 5B5	Traded 1 Leather and 2 Camel for 1 Diamond and 2 Spice	Collected 2 Diamond and 3 Spice	Very good action, the player not only took the last Diamond card to be able to take the last Diamond token but also took 2 Spice to get a set of 3 cards
P: 1D, 1G, 1Sv, 1Sk, 7C, 24R, 0oC M: 1D, 1Sv, 2Sk, 1L T: 5D, 3G, 5Sv, 3Sk, 6L, 5B3, 4B4, 5B5	Traded 3 Camel for 1 Diamond, 1 Silver and 1 Silk	Collected 2 Diamond, 2 Silver and 2 Silk	Very good action, not only took the high-value cards but also took the Silk card but could not take more cards due to hand size. The player also kept more Camel cards than the opponent could possibly get
P: 1Sv, 1Sk, 1L, 2C, 43R, 4oC M: 1G, 1Sk, 3L T: 3D, 3G, 1Sv, 4Sk, 3Sp, 9L, 6B3, 5B4, 5B5	Traded 1 Silk and 1 Camel for 1 Gold and 2 Leather	Collected 1 Gold and 3 Leather	Very good action, the player prioritised getting 3 Leather and 1 Gold instead of 1 Silk as the remaining Leather tokens had more value than the Silk tokens
P: D, G, 1Sv, 1Sk, 1Sp, L, 7C, 38R, 0oC M: D, G, Sv, 2Sk, 1Sp, 1L, 1C T: 3D, 3G, 1Sv, 5Sk, 3Sp, 6L, 6B3, 5B4, 5B5	Traded 1 Silver and 3 Camel for 2 Silk, 1 Spice and 1 Leather	Collected 3 Silk, 2 Spice and 1 Leather	Good action for collecting 3 Silk and 2 Spice however it was questionable for the player to trade 1 Silver for 1 Leather but it could be due to more Leather tokens remaining which would total to more points as well as the possibility of taking bonus tokens
P: 4Sk, 1Sp, 2L, 2C, 20R, 4oC M: 1Sv, 1Sk, 1Sp, 2L T: 3D, 1G, 3Sv, 7Sk, 7Sp, 6L, 6B3, 6B4, 5B5	Sold 4 Silk	Took first 4 Silk tokens and a bonus token for selling 4 cards	Very good action, managed to collect 4 Silk cards but also prioritised selling to not miss out on the first 4 Silk tokens

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, oC: Opponent's Camel,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

Table F.23 DQN Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 1Sk, 1L, 4C, 45R, 4oC M: 5L T: 1D, 1G, 1Sv, 5Sk, 4Sp, 9L, 7B3, 6B4, 5B5	Traded 4 Camel for 4 Leather	Collected 5 Leather	Very good action, the player prioritised getting a set of 5 cards
P: 1Sk, 5L, 45R, 8oC M: 1D, 1Sv, 2Sk, 1L T: 1D, 1G, 1Sv, 5Sk, 4Sp, 9L, 7B3, 6B4, 5B5	Sold 5 Leather	Took first 5 Leather tokens and a bonus token for selling 5 cards	Very good action, the player collected and sold 5 cards at once and prioritised doing so over trading out 2 Goods cards to take the Diamond and Silver cards, given that the player did not have the last cards of these type to sell, and by selling 5 Leather cards the player obtained more points
P: 1Sv, 1Sk, 1Sp, 3C, 69R, 7oC M: 1Sk, 3Sp, 1L T: 1G, 1Sv, 1Sk, 4Sp, 1L, 6B3, 6B4, 4B5	Traded 1 Silk and 2 Camel for 2 Spice and 1 Leather	Collected 3 Spice and 1 Leather	Questionable action as it would have been more ideal to get 3 Spice for a total of 4. It is also unsure why the player exchanged Silk for Leather since the same number of tokens remained with the same number of points
P: 1D, 1Sv, 1Sk, 1Sp, 2L, 10C, 49R, 0oC M: 1D, 1Sk, 2Sp, 1C T: 1D, 1G, 1Sv, 1Sk, 4Sp, 1L, 7B3, 5B4, 5B5	Took 1 Diamond	Collected the last 2 Diamond	Good action, took the last Diamond card to be able to take the last Diamond token
P: 1D, 2C, 10R, 0oC M: 1Sk, 4C T: 5D, 5G, 5Sv, 6Sk, 5Sp, 7L, 7B3, 6B4, 5B5	Took all Camel cards	Collected a large amount of Camel cards	Good action, prioritised getting a large number of Camel cards at the early stages of the game to be able to trade out for Goods cards

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, oC: Opponent's Camel,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

Table F.24 DDQN Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 3D, 1Sk, 1Sp, 2C, 15R, 7oC M: 3Sk, 2Sp T: 5D, 5G, 3Sv, 4Sk, 6Sp, 5L, 7B3, 5B4, 5B5	Sold 3 Diamond	Took first 3 Diamond tokens and a bonus token for selling 3 cards	Very good action, the player prioritised getting the first 3 Diamond tokens, which have the highest number of points in the game, as well as a bonus token
P: 1Sv, 4L, 7C, 11R, 2oC M: 3Sk, 2Sp T: 5D, 5G, 3Sv, 4Sk, 6Sp, 9L, 7B3, 6B4, 5B5	Sold 4 Leather	Took first 4 Leather tokens and bonus token for selling 4 cards	Very good action, collected and sold 4 Leather cards to not only get the first Leather tokens but also the bonus token
P: 1D, 1G, 7C, 19R, 2oC M: 2Sk, 1Sp, 2L T: 3D, 5G, 3Sv, 5Sk, 6Sp, 8L, 7B3, 6B4, 5B5	Traded 2 Camel for 1 Spice and 1 Leather	Collected 1 Spice and 1 Leather	Good action as the next Spice and Leather tokens are worth more than the next 2 Leather or Silk tokens
P: 1Sv, 1L, 11R, 2oC M: 1Sk, 4C T: 5D, 5G, 5Sv, 4Sk, 6Sp, 9L, 7B3, 6B4, 5B5	Took all Camel cards	Took Camel cards to compete with opponent	Good action, prioritised getting a large number of Camel cards at the early stages of the game to be able to trade out for Goods cards and compete with opponent
P: 1D, 1Sk, 3C, 70R, 7oC M: 4Sp, 1L T: 2D, 1Sv, 3Sp, 1L, 6B3, 4B4, 5B5	Traded 3 Camel for 3 Spice	Collected 3 Spice	Good action as the player collected 3 Spice to be able to take the last 3 Spice tokens but would have been ideal to trade out Silk, as no more tokens remained, for the other Spice to obtain the bonus token of selling 4 cards instead of 3

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, oC: Opponent's Camel,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

F.7 Action Embedding

Table F.25 PPO Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 3G, 2Sp, 36R, 9oC M: 2D, 1Sk, 1L, 1C T: 3D, 5G, 3Sv, 1Sk, 2Sp, 4L, 5B3, 6B4, 5B5	Traded 2 Spice for 2 Diamond	Collected 2 Diamond	Very good action, prioritised getting 2 Diamond over keeping 2 Spice, selling 3 Gold immediately and risking that the opponent takes them.
P: 4G, 2Sk, 1L, 43R, 8oC M: 2Sp, 3C T: 1D, 3G, 1Sv, 3Sk, 2Sp, 2L, 5B3, 6B4, 5B5	Sold 4 Gold	Took last 3 Gold tokens and a bonus token of selling 4 cards	Very good action, collected last 4 Gold cards and sold them to not only take the last Gold tokens but also take a bonus token
P: 1Sk, 3Sp, 1L, 0R, 0oC M: 5C T: 5D, 5G, 5Sv, 7Sk, 7Sp, 9L, 7B3, 6B4, 5B5 Start of game	Sold 3 Spice	Took first 3 Spice tokens and bonus token for selling 3 cards	Very good action, sold the card combination which returns the highest number of points at the start of the game
P: 2Sp, 1L, 2C, 20R, 0oC M: 2Sk, 3C T: 3D, 3G, 5Sv, 7Sk, 5Sp, 7L, 7B3, 6B4, 5B5	Traded 2 Spice for 2 Silk	Collected 2 Silk	Very good action, the next Silk tokens have more points than the next Spice tokens
P: 1G, 2L, 9C, 55R, 0oC M: 3Sk, 1Sp, 1C T: 3D, 3G, 1Sv, 3Sk, 2Sp, 4L, 6B3, 6B4, 5B5	Traded 3 Camel for 3 Silk	Collected 3 Silk	Very good action, got a set of 3 Silk cards to obtain the last 3 Silk tokens and a bonus token

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, oC: Opponent's Camel,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

Table F.26 A2C Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 3D, 1Sk, 1L, 0R, 0oC M: 1Sv, 1Sk, 3C T: 5D, 5G, 5Sv, 7Sk, 7Sp, 9L, 7B3, 6B4, 5B5 Start of game	Sold 3 Diamond	Took the first 3 Diamond tokens and a bonus token of selling 3 cards	Very good action, prioritised taking the first 3 Diamond tokens since they have the most points in the game
P: 1Sv, 1Sp, 5C, 81R, 6oC M: 1Sv, 4L T: 1Sv, 1Sk, 1Sp, 3L, 4B3, 6B4, 5B5	Traded 1 Silver and 3 Camel for 4 Leather	Collected last 4 Leather	Very good and interesting action, prioritised taking the last 4 Leather to take the last 3 Leather tokens and a bonus token for selling 4 cards. Might have given away a Silver card to bait the opponent into taking that card instead of selling for the player to end the game by selling the 4 Leather cards
P: 4G, 1Sv, 1Sk, 1Sp, 27R, 7oC M: 1Sv, 1Sk, 3C T: 5G, 3Sv, 4Sk, 1Sp, 7B3, 6B4, 5B5	Sold 3 Gold	Took first 3 Gold tokens and a bonus token of selling 3 cards	Very good and interesting action, the player decided to keep 1 Gold card to not miss out on the first Gold tokens and to make it easier to get the last 2 Gold tokens
P: 7C, 52R, 0oC M: 4Sv, 1Sp T: 3D, 1G, 5Sv, 1Sk, 5Sp, 3L, 7B3, 6B4, 5B5	Traded 4 Camel for 3 Silver and 1 Spice	Collected 3 Silver and 1 Spice	Questionable action, whilst it was very good for the player to take 3 Silver cards, it is unsure why they left 1 Silver, especially since they also took 1 Spice
P: 1D, 7C, 20R, 0oC M: 1Sv, 3Sp, 1L T: 5D, 2G, 3Sv, 3Sk, 5Sp, 8L, 6B3, 6B4, 5B5	Traded 2 Camel for 1 Silver and 1 Leather	Collected 1 Silver and 1 Leather	Good move, took the Silver card and chose Leather over Spice, as the remaining Leather and Silk tokens are of the same value but there are more Leather tokens, without using a lot of Camel cards

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, oC: Opponent's Camel,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

Table F.27 DQN Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 1Sv, 1L, 6C, 58R, 4oC M: 1Sk, 3Sp, 1C T: 1Sv, 1Sk, 2Sp, 1L, 7B3, 6B4, 5B5	Sold 1 Leather	Took last Leather token and ended the game with more Camel cards	Very good action, the player chose to end the game whist having more Camel cards to get the Camel bonus token
P: 1G, 1Sv, 1Sp, 1L, 1C, 0R, 0oC M: 1G, 1Sp, 3C T: 5D, 5G, 5Sv, 7Sk, 7Sp, 9L, 7B3, 6B4, 5B5 Start of game	Took 1 Gold	Collected 2 Gold	Very good action, the player prioritised taking the Gold card, to get 2 cards and be able to sell, instead of taking the first Spice or Leather token
P: 1D, 1G, 1Sv, 8C, 53R, 2oC M: 1D, 1Sk, 2Sp, 1C T: 1D, 1G, 1Sv, 1Sk, 4Sp, 6B3, 6B4, 5B5	Traded 2 Camel for 1 Diamond and 1 Silk	Collected 2 Diamond and 1 Silk	Very good action, not only took the Diamond card to be able to take the last Diamond token but also took a Silk card to have more control on when the game is ended, whilst keeping more Camel cards than the opponent could get
P: 1G, 1Sv, 8C, 46R, 2oC M: 1D, 4Sp T: 1D, 1G, 3Sv, 1Sk, 4Sp, 6B3, 6B4, 5B5	Traded 5 Camel for 1 Diamond and 4 Spice	Collected 1 Diamond and 4 Spice	Very good action, the player prioritised getting the Diamond card as well as a set of 4 cards
P: 1D, 1G, 1Sv, 3C, 42R, 6oC M: 1Sp, 3L, 1C T: 1D, 1G, 3Sv, 1Sk, 3L, 7B3, 6B4, 5B5	Traded 3 Camel for 3 Leather	Collected 3 Leather	Very good action, the player prioritised getting a set of 3 cards in exchange for all the Camel cards, especially since the opponent had most Camel cards

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, oC: Opponent's Camel,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

Table F.28 DDQN Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 1D, 1Sv, 33R, 5oC M: 1Sk, 1L, 3C T: 5D, 3Sv, 3Sk, 2Sp, 6L, 6B3, 6B4, 5B5	Took all Camel cards	Collected Camel cards to compete with opponent	Good action, the player chose to take the Camel cards to try to compete with opponent
P: 1D, 1Sv, 4C, 33R, 5oC M: 2Sk, 3L T: 5D, 3Sv, 3Sk, 2Sp, 6L, 6B3, 6B4, 5B5	Traded 3 Camel for 3 Leather	Collected 3 Leather	Very good action, the player prioritised taking a set of 3 cards
P: 2G, 1Sv, 5C, 22R, 1oC M: 1Sv, 1Sk, 2L, 1C T: 3D, 3G, 5Sv, 5Sk, 6Sp, 7L, 7B3, 6B4, 5B5	Traded 2 Camel for 1 Silver and 1 Silk	Collected 2 Silver and 1 Silk	Very good action, the player not only took the Silver card to be able to sell but also took a Silk card to potentially sell in another turn whilst keeping more Camel cards than the opponent
P: 1G, 7C, 71R, 3oC M: 3Sk, 1L, 1C T: 1D, 1G, 1Sv, 2Sk, 5B3, 6B4, 5B5	Traded 1 Gold and 1 Camel for 2 Silk	Collected 2 Silk	Good action, despite trading out the Gold card, the player took the 2 Silk cards to get the last 2 Silk tokens and end the game with more Camel cards
P: 1G, 2Sk, 2Sp, 9C, 57R, 0oC M: 3L, 2C T: 1G, 1Sk, 1Sp, 2L, 6B3, 6B4, 5B5	Traded 1 Gold, 1 Silk and 1 Spice for 3 Leather	Collected 3 Leather	Very good action, the player prioritised keeping the Camel cards and giving out the Goods cards to take the last 3 Leather cards in order to obtain the last 2 Leather tokens and a bonus token, ending the game and taking the Camel token

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, oC: Opponent's Camel,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

F.8 Hierarchical RL with Centralised Critic

Table F.29 PPO Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 2D, 1Sk, 1Sp, 1C, 0R, 0oC M: 1Sk, 1L, 3C T: 5D, 5G, 5Sv, 7Sk, 7Sp, 9L, 7B3, 6B4, 5B5 Start of game	Sold 2 Diamond	Took first 2 Diamond tokens	Very good action, took the first 2 Diamond tokens which have the most points in the game
P: 1G, 2C, 52R, 5oC M: 1G, 1Sv, 3C T: 3G, 1Sv, 1Sp, 2L, 7B3, 6B4, 5B5	Traded 2 Camel for 1 Gold and 1 Silver	Collected 2 Gold and 1 Silver	Very good action, prioritised taking both high value cards instead of keeping Camel cards
P: 1Sv, 1C, 62R, 10oC M: 1G, 2Sp, 2L T: 1G, 1Sv, 1Sp, 2L, 7B3, 6B4, 5B5	Traded 1 Silver and 1 Camel for 2 Spice	Collected last 2 Spice	Very good and interesting action, prioritised taking the last 2 Spice over keeping Silver or taking Gold with the intention to sell and end the game
P: 7C, 37R, 1oC M: 2Sk, 3L T: 1D, 3G, 3Sv, 2Sk, 1Sp, 7L, 7B3, 6B4, 5B5	Traded 3 Camel for 3 Leather	Collected 3 Leather	Very good action, took a set of 3 cards and prioritised Leather over Silk, as the respective Leather and bonus tokens would return higher rewards, whilst keeping more Camel cards than the opponent
P: 8C, 56R, 1oC M: 2Sp, 2L, 1C T: 1D, 3G, 1Sv, 1Sp, 2L, 6B3, 6B4, 5B5	Traded 4 Camel for 2 Spice and 2 Leather	Collected last 2 Spice and 2 Leather	Good and interesting action, took all the Goods cards which could cause the game to end upon being sold to have more control over when the game ends

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, oC: Opponent's Camel,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

Table F.30 A2C Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 1D, 10C, 26R, 0oC M: 1D, 1Sv, 2Sp, 1L T: 3D, 3G, 5Sv, 2Sk, 4Sp, 3L, 7B3, 6B4, 5B5	Traded 4 Camel for 1 Diamond, 1 Silver and 2 Spice	Collected 2 Diamond, 1 Silver and 2 Spice	Very good action, took all high-value cards as well as 2 Spice, instead of Leather as the remaining Spice tokens had more points than Leather tokens, and kept more Camel cards than the opponent
P: 1G, 1Sv, 10C, 16R, 0oC M: 1G, 2Sv, 2L T: 3D, 3G, 5Sv, 3Sp, 5L, 7B3, 6B4, 5B5	Traded 3 Camel for 1 Gold and 2 Silver	Collected 2 Gold and 3 Silver	Very good action, took all high-value cards and kept more Camel cards than the opponent
P: 1G, 1C, 10R, 5oC M: 2D, 3Sv T: 5D, 5G, 5Sv, 6Sk, 5Sp, 7L, 7B3, 6B4, 5B5	Traded 1 Gold and 1 Camel for 2 Diamond	Collected 2 Diamond	Very good and interesting action, the player could not take both Diamond cards without loosing 1 Gold and hence prioritise taking 2 Diamond instead of having 1 Diamond and 1 Gold to get the first 2 Diamond tokens which have the most points in the game
P: 8C, 10R, 0oC M: 2G, 1Sv, 1Sk, 1L T: 5D, 5G, 5Sv, 5Sk, 4Sp, 5L, 7B3, 6B4, 5B5	Traded 4 Camel for 2 Gold, 1 Silver and 1 Leather	Collected 2 Gold, 1 Silver and 1 Leather	Very good action for taking all high-value card, however would have been more ideal to take Silk over Leather as the Silk tokens are worth more than the Leather tokens and the same number of cards remain for each type
P: 2G, 1Sv, 1L, 4C, 10R, 0oC M: 1Sk, 4C T: 5D, 5G, 5Sv, 5Sk, 4Sp, 5L, 7B3, 6B4, 5B5	Sold 2 Gold	Took first 2 Gold tokens	Very good action, prioritised taking the first 2 Gold tokens which are worth the most

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, oC: Opponent's Camel,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

Table F.31 DQN Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 1Sv, 1Sk, 1Sp, 1L, 1C, 0R, 0oC M: 1Sk, 4C T: 5D, 5G, 5Sv, 6Sk, 7Sp, 9L, 7B3, 6B4, 5B5	Sold 1 Spice	Took first Spice token	Very good action, sold Spice instead of Silk or Leather since the first Silk token had already been taken and the first Leather token is worth less points
P: 2G, 1Sv, 1Sk, 1Sp, 0R, 0oC M: 1D, 1L, 3C T: 5D, 5G, 5Sv, 7Sk, 7Sp, 9L, 7B3, 6B4, 5B5 Start of game	Sold 2 Gold	Took the first 2 Gold tokens	Good action for taking the first 2 Gold tokens however there was a Diamond in the marketplace, the player risked loosing this card to the opponent instead of loosing the first 2 Gold tokens
P: 1D, 1G, 2L, 1C, 0R, 1oC M: 1Sp, 4C T: 5D, 5G, 5Sv, 7Sk, 7Sp, 9L, 7B3, 6B4, 5B5 Start of game	Sold 1 Leather	Took first Leather token	Very good and interesting action, sold only 1 Leather to take the first token which is worth the most points with the intention to collect more Leather cards and sell them as a set for the bonus tokens
P: 6C, 57R, 1oC M: 3Sk, 2C T: 1D, 3G, 1Sv, 5Sk, 4Sp, 4L, 7B3, 6B4, 5B5	Traded 2 Camel for 2 Silk	Collected 2 Silk	Questionable action, would have been ideal to take the set of 3 cards however the player prioritised keeping more Camel cards, potentially to not miss out on the Camel token
P: 1D, 1G, 1Sv, 1Sp, 4C, 67R, 6oC M: 2Sk, 2L, 1C T: 1D, 1G, 1Sv, 1Sk, 1L, 7B3, 6B4, 5B5	Sold 1 Spice	No Spice tokens remained	Questionable action as no Spice tokens remained, may have performed this action to prolong the game and force the opponent to modify the marketplace for a good card

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, oC: Opponent's Camel,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards

Table F.32 DDQN Action Selection Analysis Results

State	Action	Outcome	Analysis
P: 1D, 1G, 5C, 53R, 1oC M: 1L, 4C T: 1D, 3G, 1Sv, 2Sk, 3Sp, 4L, 7B3, 6B4, 5B5	Took all Camel cards	Collected 9 Camel cards	Very good action, prioritised getting the majority of Camel cards to get the Camel token or to trade for Goods cards
P: 1Sv, 7C, 51R, 4oC M: 2Sk, 3L T: 1D, 1Sv, 3Sk, 5L, 7B3, 6B4, 5B5	Traded 3 Camel for 3 Leather	Collected 3 Leather	Very good action, prioritised getting a set of 3 cards
P: 1Sv, 3L, 4C, 51R, 7oC M: 1D, 3Sk, 1L T: 1D, 1Sv, 3Sk, 5L, 7B3, 6B4, 5B5	Sold 3 Leather	Took 3 Leather tokens and a bonus token of selling 3 cards	Questionable action, it was good to collect and sell 3 cards, however the player left a Diamond card and also could have taken the other Leather card to sell 4 cards at once for a higher bonus token
P: 1D, 1G, 5C, 18R, 1oC M: 1Sk, 3Sp, 1L T: 5D, 5G, 5Sv, 4Sk, 4Sp, 5L, 7B3, 6B4, 5B5	Traded 2 Camel for 1 Silk and 1 Leather	Collected 1 Silk and 1 Leather	Good action, would have made more sense to take the 3 Spice instead however the player prioritised keeping more than half the Camel cards to not have less than the opponent
P: 1Sk, 1C, 60R, 8oC M: 1Sp, 4L T: 1D, 1Sv, 1Sk, 3L, 7B3, 6B4, 5B5	Traded 1 Silk and 1 Camel for 2 Leather	Collected 2 Leather	Very good action, prioritised taking 2 Leather over 1 Silk to get more points

D: Diamond, G: Gold, Sv: Silver, Sk: Silk, Sp: Spice, L: Leather, C: Camel,
P: Player, M: Marketplace, T: Tokens, R: Rupees, oC: Opponent's Camel,
B3: Bonus tokens for selling 3 cards, B4: Bonus tokens for selling 4 cards, B5: Bonus tokens for selling 5 cards