UNIVERSITY OF
LIVERPOOL

Reinforcement Learning for Heads up Limit Hold'em Poker

Rhys Cooper
201680340
Supervisor: John Fearnley
Second Marker: Martin Gairing

A DISSERTATION
Submitted to
The University of Liverpool
in partial fulfilment of the requirements
for the degree of
MASTER OF SCIENCE

1 Abstract

The Proximal Policy Optimisation reinforcement (PPO) learning algorithm is applied to the zero-sum, two player poker game of Texas Limit Hold'em. Using a version of the game available through the PettingZoo library, this was modified from a multi-agent to a single agent environment such that the PPO algorithm imported from StableBaselines3 could be deployed. After an experiment was ran to determine the optimal observation space type for the algorithm, the primary and secondary hyperparameters were tuned and then the agent was trained through a process of self-play. This involves the agent essentially playing against a previous version of itself in order for the learn a superior strategy as the ability of the opponent increases through out the repeating process. The aim is for the agent to discover a Nash Equilibrium strategy: one that the agent has no incentive to deviate from to increase its payoff and constitutes an optimal strategy for the game.

Two different forms of self-play approaches were tested, one where the agent starts with the same policy as its opponent and attempts to learn a superior one, and then one in which the policy of the agent is reset and it learns anew. The former of these self-play approaches proved to be highly successful, resulting in an agent who achieved a resource constrained equilibrium strategy that proved superior to all opponents it was tested against. This included a beginner level human opponent, a opponent using a heuristic based strategy and also the similar Advantage Actor-Critic algorithm which underwent the same extent of hyperparameter tuning and self-play training. The fact that the resultant policy of the trained PPO displayed convergence to an equilibrium strategy in its moving mean reward, achieved a difference in probability distributions to the previous version of itself equal to zero and the level of regret shown in the best response analysis all provide support for the interpretation that the agent achieved an effective resource constrained equilibrium that is proximal to Nash Equilibrium.

2 Student Declaration

I confirm that I have read and understood the University's Academic Integrity Policy.

I confirm that I have acted honestly, ethically and professionally in conduct leading to assessment for the programme of study.

I confirm that I have not copied material from another source nor committed plagiarism nor fabricated data when completing the attached piece of work. I confirm that I have not previously presented the work or part thereof for assessment for another University of Liverpool module. I confirm that I have not copied material from another source, nor colluded with any other student in the preparation and production of this work.

I confirm that I have not incorporated into this assignment material that has been submitted by me or any other person in support of a successful application for a degree of this or any other university or degree-awarding body.

SIGNATURE _____R T COOPER_____

DATE November 8, 2023

3 Acknowledgments

I would like to extend my gratitude to the primary supervisor, Dr John Fearnley, for his invaluable advice and guidance during the pursuit of this project. Many thanks is also offered to Professor Martin Gairing for his supportive feedback and to Dr Meng Fang for sparking my interest in reinforcement learning and establishing my foundational knowledge of the subject.

Contents

List of Figures

4 Introduction

This project applies two similar reinforcement learning algorithms to a two player poker game called Heads up Texas limit hold'em (HULH). The essence of reinforcement learning (RL) is that the agent will learn some degree of strategy through the process of observing the game, taking an action and then receiving a reward. By playing tens of thousands of games, the learning algorithm of the agent will form a policy: what action to take given a certain state. In the context of this game, a state will be the players hand of cards and the chips the players bet with. Terminology wise, the player that is using the reinforcement algorithm will be referred to to as the 'agent' where as the other player will be the 'opponent'. The aim of the agent is to reach a Nash Equilibrium strategy. Nash Equilibrium is a concept from game theory heavily used in economics and decision theory to describe a strategy such that the agent can not improve its expected payoff by deviating.

The HULH game from the PettingZoo library has to be modified through the use of a custom wrapper that automatically provides an observation to the opponent and steps the environment with the chosen action. From the agents perspective, this has transformed the environment from multi-agent to a single agent, which allows the PPO and A2C single agent learning algorithms to be inserted. To achieve the best possible performance, the optimal observation space type is determined and the primary and secondary hyperparameters of the model are tuned. The models then undergo the self-play training scheme in order to find the most effective strategy for the game. The performance of the resultant strategies is tested through a range of opponents including a random player, the A2C algorithm, a human player and also a heuristic policy based agent where the range of abilities allows the relative skill level of the trained PPO to be established. The use of two different self-play approaches, symmetric compared to asymmetric initial policies, produces the recommendation of the former in finding optimal equilibrium strategies through self-play.

Several methods are used to analyse the effectiveness of the discovered strategy. This includes the use of KL divergence, a measure of the difference in probability distribution when an agent and the former version of itself are provided a set of the same observations. If an equilibrium strategy has been discovered, this divergence will equal zero for the last generation, with the inclusion of best response regret analysis to supplement this.

4.1 Problem Statement

This project seeks to determine the ability of the PPO single agent reinforcement learning algorithm to find the Nash Equilibrium strategy for HULH. Furthermore, since Section 2 identified that there exists a gap in existing studies to determine if the PPO algorithm can find a NE strategy in HULH, this project has value in expanding the knowledge associated with reinforcement learning to find Nash Equilibrium.

4.2 Scope

The scope is limited to determining the success of the PPO algorithm to find a NE strategy, as opposed to being a comparative study between the efficacy of the PPO and A2c algorithms. The latter RL algorithm is only included as a basis for evaluating the performance of the trained PPO algorithm, yet still undergoes the same extent of training and tuning to ensure a fair comparison.

4.3 Approach

The overarching approach is to create a custom RL environment for the HULH game that is compatible with the single agent RL learning algorithms to which the algorithm can be deployed. The hyperparameters of the algorithms are tuned and then undergo a self-play training process. The learnt strategy of the final trained agent is inspected to determine the extent to which a NE strategy is achieved and then the performance of the agent is evaluated against a range of opponents. This includes an opponent with a random policy, a self-play trained A2C opponent, a beginner human player and also a heuristic policy based agent. All of these steps, and interim procedures are detailed in Section 8.

5 Literature Review

5.1 Poker domains

The nature of Poker as a game allows the testing of many single and multi-agent reinforcement learning algorithms and as such, quickly became a challenge domain (Lockhart et al, 2020). It is important to note that there are four different versions of the Texas Hold'em game used by studies within the literature. The most simplified version - Kuhn hold'em - has no community cards and only one round of betting. This simpler game is commonly used because NE strategies can be easily identified since a closed form, parameterized solution exists (Heinreich and Silver, 2015).

Another commonly used version is Leduc poker: Kuhn poker but with additional betting rounds and two final community cards. The next most developed version is the one used in this project, is the two player version of Limit Texas Hold'em typically called Heads up Limit Hold'em (HULH). This has research value since it is the most simplified poker game still played by real professional players.

The limit condition of HULH means that the game has fixed increments of betting and a limit to the number of raises an agent can make. This limit condition simplifies the agent's action space and makes it more compatible with RL methods. Furthermore, since HULH is an extensive form zero sum game - the payoff of one agent is the direct opposite to the other - this allows for a possible NE to exist. Additionally, HULH provides further simplification to the RL environment since the game always terminates once a player folds or loses. Since an agent does not know the hand of the other player, the environment is characterized as being an imperfect information game (IIG) (Bowling et al., 2015). As a result, a non-deterministic, mixed solution is what an agent aims to learn, one of which is classified as being an optimal poker strategy if a NE strategy is reached.

5.2 Counterfactual Regret Minimisation Algorithms

Counterfactual regret minimisation (CFR) are a family of algorithms that are used by the most famous no limit Hold'em programs: Libratus (Brown and Sandholm, 2017) and Deep stack (Moravcik et al., 2017). They operate through a pair of regret minimizing algorithms that undergo self play and are able to find a NE strategy, an ability arising from the suitability of CFR's to IIG's (He et al., 2022). The aforementioned CFR based programs successfully applied to heads up no-limit hold'em (HUNL) are tabular methods that demand extensive amounts of memory. This led to the development of Deep CFR; model free methods that replace the tabular storage with neural networks and are similarly able to converge to ϵ-nash equilibrium strategies. Since the success of Deep-CFRs for no-limit hold'em has already been established, these types of algorithms are removed as possible learning algorithms for this project.

5.3 Neural Fictitious Self Play

The second most successful learning algorithm applied to HULH is Neural Fictitious Self Play (NFSP). The core premise of fictitious self play is the average strategy of the opponent is modeled and the agent calculates a best response, the latter of which is approximated using neural networks. NFSP and similar learning algorithms are successful in finding NE strategies in HULH (Heinrich and Silver, 2016). Similarly to CFR's, this type of algorithm is therefore excluded as a possible algorithm for this project.

5.4 Policy Gradient Methods

Policy gradient methods operate by directly updating the parameters of an agent's policy using gradient ascent on the estimated expected return, as opposed to estimating a direct value function. The candidate policy gradient methods for this project include Advantage Actor Critic (A2C) and Proximal Policy Optimisation (PPO).

Many studies have applied policy gradient methods to Poker. Deepholdem (Wang et al., 2022) successfully deployed deep CFR alongside PPO to NLTH. This considered, a research area appears in applying PPO, without deep CFR, to HULH. Expanding on this notion, a similar study by Lazaridis et al (2022) use PPO trained under self play on NLTH. They discover that the PPO based agent has a higher mean reward than both a random strategy opponent and a Monte Carlo search based 'hand evaluator, yet a negative mean reward when played against a static opponent who always checks or raised. The authors suggest instability to be the cause of this, to which the unstable learning curve of this PPO from Lazardis et al. (2022) is shown in Figure 1 below.

Figure 1: Unstable PPO Learning, extracted from Lazardis et al.(2022)

In terms of explaining this relatively poor performance, the main limitation of policy gradient methods is that they face difficulty in the non-markovian and non-stationary setting of a multi-agent environment. Consequently, in the context of imperfect information games there are no convergence guarantees in finding NE strategy through self play (Fu et al., 2022). With this considered, the novel research question appears to see if the relatively poor performance exhibited by the PPO algorithm on the HUNLH (Lazardis et al., (2022)) also occurs for HULH.

In contrast to the finding of Lazaridis et al. (2022), Charlesworth (2018) applied an PPO trained through self play to a four player IIG called 'Big 2' and found the PPO was able to achieve better than human performance. Past the differences in the games, a possible explanation for this contrasting finding is that Charlesworth (2018) used parallel environments to create large batches in order to reduce the amount of variance. Fortunately the Pettingzoo API supports the use of parallel environments and this functionality will be implemented when deploying the PPO.

A similar study uses a self trained PPO for a four player imperfect information card game called Guandan (Pan et al, 2022). The overall PPO neural network structure is identical to that of Charlesworth (2002), other than the latters use of RELU activation function rather than TANH. Naturally this provides possible hyperparameter options for this project. Additionally, it has been noted that little hyperparameter tuning was required to observe strong performance by the PPO model, and allows for action masking (Charlesworth, 2020; Pan et al., 2023).

Yang et al. (2023), applying PPO to the 'ticket to ride' game with a state space larger than chess, found that a PPO that undergoes self play was the highest performing agent against all the other RL opponents. This lends support to both the self play training schedule and choice of PPO as the learning algorithm for this project.

In terms of Advantage Actor Critic (A2C) learning algorithm of Mnih et al. (2016), this algorithm was applied by Srinivasan et al. (2018) to the domains of Leduc and Kuhn poker. Naturally another research area appears in testing the application of A2C to HULH and as such, this is chosen as the algorithm to compare the performance of the PPO against.

5.5 DQN

Deep Q Networks (DQN) are a common framework of reinforcement learning algorithms due to which their relative simplicity and easy incorporation of action masking make them an attractive learning algorithm to choose. Despite this, the standard DQN algorithm applied to HULH is unable to find a NE strategy (Heinrich and Silver, 2016). This is due to the limited stability of the DQN algorithm where the non-stationary nature of a two player game means the MDP property that conventional single agent RL algorithm exploit no longer holds. Considering the developments made to DQN's since, a possible research area arises of deploying an improved version of DQN - such as rainbow DQN - to investigate if there exists improved stability such that it can discover a NE strategy. Despite this, there is a notable absence of studies deploying DQN, and its variations, to imperfect information zero sum games. Therefore a DQN variant will not be chosen as the learning algorithm for this project.

5.6 Multi Agent Reinforcement Algorithms

In terms of the use of multi agent reinforcement algorithms (MARL), this can at the most basic level involve both agents using single agent RL methods. However, as mentioned by Zhong et al. (2020), these RL algorithms applied in the MARL environment will likely face difficulty in finding NE, as shown for DQN (Heinrich and Silver, 2016). MARL algorithms instead attempt to account for the non-stationary and non-markovian properties of the multi-agent environment. Within the MARL context, Deep CFR and NFSP have all been able to find NE for HULH. There remains a selection of possible MARL algorithms that can be applied to HULH, namely Policy Space Responce Oracles (PSRO) and Exploitability descent (ED). ED was successfully applied to Leduc and Kuhn Poker (Lockhart et al., 2019), yet there remains the research area of applying it to the more advanced HULH game. Due to constraints in time and expertise, MARL algorithms will not be implemented in this project.

6 Aims and Objectives

This section re-states the aims and objectives from the project specification and design along with an appraisal on the extent to which they have been achieved.

Aim 1: Create a reinforcement learning environment of the HULH game.
- *Build a custom wrapper that modifies the two player game into a single agent environment.*
- *Modify the functionality of the resulting custom environment to gymnasium syntax to allow the implementation of prebuilt algorithms.*

Aim 1 was fully achieved. This was confirmed by running a test from StableBaseline 3 (SB3) to check this custom environment conforms to the gymnasium syntax.

Aim 2: Deploy the PPO reinforcement learning algorithm to HULH.
- *Train algorithm using self play.*
- *Tune hyperparameters.*

Aim 2 was fully achieved. Fulfilling the self-play objective required modification of the imported code to allow complete loading of the the self-play agents network architecture. Hyperparameter tuning required longer than expected to do the environment not being able to run in parallel due to the way action masking was applied.

Aim 3: Evaluate the performance of the trained PPO agent.
- *Use key evaluation techniques from similar studies.*
- *Detect to what extent the agent has learnt a strategy for the game using loss curves.*
- *Compare performance against a range of opponents.*

Aim 3 was fully achieved and exceeded. Optimal action rate included as an additional evaluation metrics, and three more opponents were included to increase the strength of performance evaluation. Here, the use of a random agent as an evaluation technique required modification of the imported PPO model in order to allow random action selection such that compatibility with the rest of the code was maintained.

Aim 4: Compare the performance in the game of PPO to Advantage Actor Critic algorithm (A2C).
- *Implement and train A2C algorithm.*
- *Tune hyperparameters.*

Aim 4 was fully achieved. The use of an imported A2C model from the same SB3 library as the former PPO meant all the previous modifications involving action masking and random policy implementation carried over. Equally, the similarities of the two models meant the same primary hyperparameter tuning values could be searched for and all the secondary hyperparameters, other than clip-range, could be tested for the A2C.

Aim 5: Find the Nash equilibrium strategy of the game through self play.
- repeat policy transfer to opponent until no significant change in the difference between the score of the agent and the opponent.

Aim 5 was fully achieved. The key graph showing the convergence of mean rewards across self-play generations illustrated the agent reaching a potential NE strategy. Yet the use of regret analysis using best response strategies alongside examination of KL divergence further support the interpretation that a resource constrained equilibrium strategy, proximal to NE was found.

7 Project design

7.1 Original design and changes

The original project specification and design document (PSD) can be found in Section 13.10 of the Appendix. The final project design and implementation included several more procedures past that specified in the preceding PSD. These include a training convergence tests, observation space amendments, an expanded hyperparameter tuning approach and additional evaluation metrics and opponents for performance testing.

The hyperparameter tuning approach implemented was more comprehensive than planned in the PSD. This was due to the use of the Optuna [1] optimisation library which provided more detailed metrics such as hyperparameter importance, and the tuning of secondary hyperparameters, detailed in Section 8.1. Agent performance evaluation also turned out to be more comprehensive than planned in the PSD. This was due to the inclusion of two more opponents: a heuristic based opponent and a human player. The human opponent used was the author of this project and as such, this did not give rise to any concerns regarding ethical use of data.

[1] https://optuna.org

Another oversight of the PSD was planning and designing a tool that would store and plot all the data metrics, such as rewards, loss and KL divergence collected throughout the procedures of the project. This data storage object had to be compatible for use within hyperparameter tuning, self-play and agent vs opponent evaluation scenarios. The design of this is explained in Section 7.7.

Finally, whilst it was previously stated that the agents observation space only includes its own chips and not the opponents, this proved to be a misunderstanding of the Pettingzoo library code. As such, the previous statement that this project is limited by the agent only observing its own chips, and therefore undermining a key dynamic of the HULH game, is now invalidated.

7.2 Training convergence test

An additional tool was made for the primary purpose of reducing the time taken to run the tuning of the primary hyperparameters; a use case warranted by the inability to run vectorised environments in parallel. The optimal values for 'minimum evaluation period's and 'number of steps with no improvement' which training convergence of a default PPO model occurred at were calculated.

Specifically, a default PPO model was trained against a Gen-0 version of itself: one which learnt a strategy against a random opponent. The moving mean reward graph is plotted to determine the number of training steps required before convergence towards a stable policy commences. As shown in Figure 19 of the Appendix, this occurred at 30,00 steps. Therefore, after 30,000 training steps has occurred the callback checks if the mean reward has improved after determined number of steps, past the mean reward from the last time the callback was called, in order to detect training convergence and stop training.

7.3 Observation space amendments

Another change to the original PSD was a modification to the observation space of the agent Here, the default RLcard state space is a boolean vector comprised of 72 elements to which the first 52 index positions encode the agents and community cards whilst the remaining 20 position encode the chips raised during each betting round. This is represented in Table 1 of the PSD in the Appendix.

It is possible that this representation may lead to a conflation of the players and community cards, with the learning agent unable to differentiate between the two, which may consequently impact learning success. It is theorized that this is the default of RLcard library because in the latter betting rounds of 'turn' and 'flop' the optimal action is more heavily determined by the community cards and therefore a merging of the players cards and community cards will internalize some degree of card ranking into the agents policy, given enough total training steps. As such, the earlier betting rounds of pre-flop and flop, where the agents hand is the more determining factor in the optimal action, the conflated state space may prove detrimental to the agents learning.

With this considered, an experiment was run to test different state space representations that differentiate between the players cards and the community cards. This included a boolean vector of length 124 which indexes a decks of cards for the agent and separate deck for the community cards, and then also a 72+ state space which places a '2' rather than '1' for the players cards in the corresponding index position. The results arising from this amendment is discussed in Section 8.3.

7.4 Game specification

A full description of the HULH game, including its objectives, gameplay, hand rankings and betting rounds, can be found in Section 4.1 of the original project specification and design document (PSD) located in the Appendix. In summary, the objective is to win all the available chips by having a better hand than the opponent or by forcing the opponent to fold, where hand rankings are the same as in standard poker. For the four betting rounds, the limit component of HULH prescribes the fixed structure for betting. A full description of the observation space, action space and reward function can be found in Sections 4.1.1, 4.1.2, 4.1.3 respectively of the original PSD.

7.5 HULH environment

The HULH game is implemented using the Pettingzoo library; a multi-agent version of the popular OpenAI Gymnasium environment. It uses the *RLcard* library to deploy the core HULH game and implants it into an Agent Environment Cycle (AEC). The AEC steps the environment for each agent, returning agent specific observations, states, rewards and legal next actions. This is shown in Figure 2. For rendering and metadata storage purposes, this is passed into the 'raw environment' class and as such, the overall inheritance structure of this is shown in the UML diagram of Figure 4.

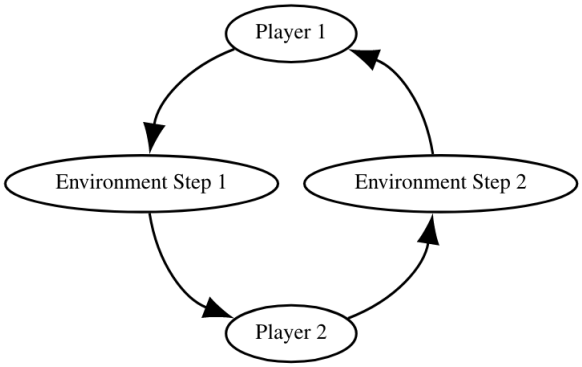


Figure 2: Agent Environment Cycle, extracted from Terry et al(2021)

This raw environment is passed through a series of functions that take as input an environment and outputs it with a wrapper applied. Each consecutive function adds a wrapper that provides functionality to impose the conditions of the game. The *terminate illegal* terminates the game if the agent chooses an illegal action, *out of bounds* ensures the action is in the action space and *order enforcing* forces the correct sequencing of environment stepping and resetting. This is shown in the sequence diagram of Figure 3.

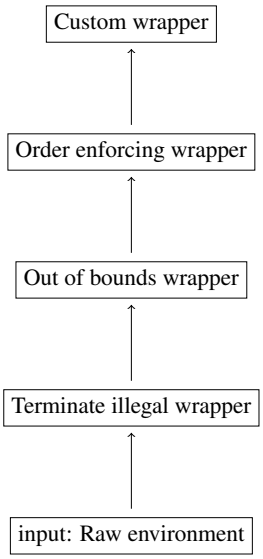


Figure 3: Sequence Diagram of environment wrappers

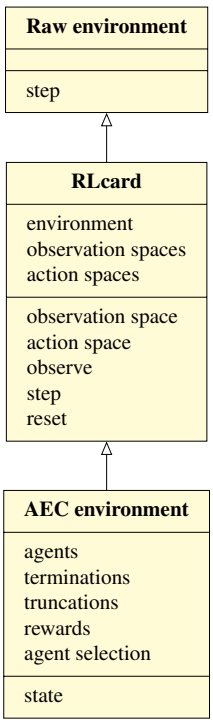


Figure 4: UML diagram showing inheritance

The modifications to the imported Pettingzoo library primarily included the transformation of the original multi-agent format into single agent through the use of a custom wrapper. This was achieved by editing the core methods of the gymnasium syntax: step, observe and reset. Most importantly, the step modification, as shown in Section 13.5 of the appendix, essentially embedded the opponent into the environment by providing a separate observation (line 9) and stepping the environment with the retrieved the action(line 15). To this effect, the agent interacts with the environment completely independent of the embedded opponent. This is equivalent to the automatic completion of the remainder of the AEC cycle shown in Figure 2, after the environment is stepped with the learning agents action.

7.6 Self-play architecture

The PPO algorithm is trained by collecting data from the environment and updating the policy parameters shown in the below steps.

1. **Data Collection:** Trajectories of the agent executing the current policy are collected, recording the agents observed states, actions, rewards, and other relevant information. These trajectories are stored as mini batches.
2. **Surrogate Objective Function:** PPO employs a surrogate objective function that measures the quality of the current policy is measured using the surrogate objective function, usually a clipped variation of the policy's objective function.
3. **Policy Update:** The collected trajectories are then used to compute the surrogate objective to which the policy parameters are adjusted to according to gradient ascent.
4. **Value Function Update:** The value network is typically trained using a regression loss between the predicted values and the actual observed returns.

A self-play training schedule is chosen, a process used by Charlesworth (2018), Lazardis et al., (2022) and Pan et al. (2023). The opponent will start with a random policy, and after the agent has learnt a stable policy, this policy will then be transferred to the opponent. The agent will then undergo the next era of learning in which it is effectively playing against the previous version of itself, as shown in Figure 5. This cycle of policy transfer will repeat until the agent reaches an equilibrium strategy. an approach that is different to the NFSP described in Section 5.3.

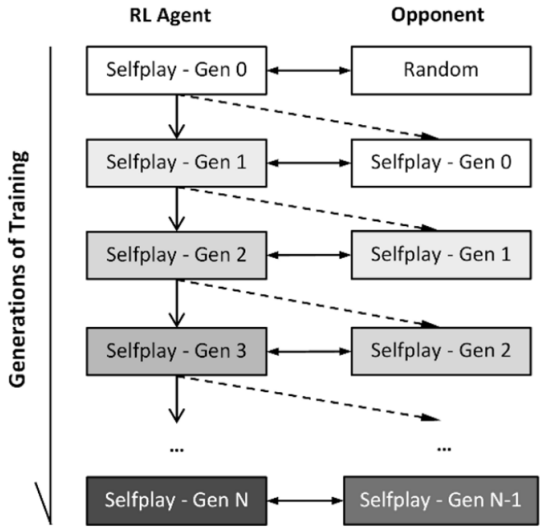


Figure 5: Self play training scheme, extracted from Yang et al.(2023)

Two self-play approaches were deployed, varying in the way in which the policies are set at the start of each generation. The first of which involves the agent and opponent starting with the same policy at the start of the generation, to which the agent will then continue learn, referred to as symmetric initial policies (SIP). This is in contrast to the other approach where at the start of a generation, the agents policy is reset, and then continues to learn, referred to as asymmetric initial policies (AIP).

The AIP approach resulted in the agent reaching an equilibrium strategy, but as shown in Figure 22 and figure 23 in the appendix, the relative in game performance of this trained agent was marginally better than random, but worse than a beginner human player. As such, the SIP self-train as implemented as the main approach and the code for this is shown in Section 13.6 of the Appendix.

7.7 Data storage and plotting

A custom callback used during training and modifications to the SB3 'evaluate policy' function are the two methods used to retrieve the required data from all the procedures of the project. The functionality of the custom callback first extracts the three loss values and rewards of the agent generated during training and then constructs the rewards, moving total and moving mean class attributes.

The modification to the SB3 'evaluate policy' function allows it to return the agents and opponents episode rewards and mean reward. A data storage object is used to store this extracted data. A separate 'graph metrics' class then takes this data storage object as an input argument and uses it to create the data series required to create the plots of the appropriate graphs.

7.8 Hyperparameter Tuning

Hyperparameter tuning consists of two components; primary hyperparameters and secondary hyperparameters. They are distinguished by the former being the common RL hyperparameters such as learning rate, network architecture and optimiser, where as the latter include hyperparameters specific to the actor-critic algorithms. This includes the likes of GAE lamda, clip range and normalised advantage.

When creating combinations of possible hyperparameters, referred to as Optuna 'trials', it is required that the total number of steps to fill the replay buffer must be divisible by the number of steps ran in each vectorised environment. Since Pettingzoo HULH can not run environments in parallel, only one environment can be used. As such the 'hyperparameter search' class inherits the batch multiplier class. The purpose of this class is to ensure what ever batch size is tested for in the Optuna trial, a valid corresponding value for n_steps is used.

The primary hyperparameter values tested for in this project are shown in Table 4 of the Appendix, to which they were influenced by those used in the most relevant papers, shown in Table 4 of the PSD report in the Appendix. The secondary hyperparameter values tested for in this project are shown in Table 5 of this Appendix.

During primary hyperparameter tuning, the opponent has been trained against a random opponent who shares the same hyperparameter combination. This is to find the hyperparameters that are best suited for the self-play training context. The top 6 combinations are selected and further tested by training for 30,000 steps with no callbacks to stop training at expected convergence, and evaluated for 1,000 steps. Since this training convergence callback was used in hyperparameter tuning to limit the run time, it is removed during the testing of the pruned combinations. The results of the hyperparameter tuning is discussed in Section 8.1.

7.9 PPO algorithm and modifications

Section 5 informed the choice of Proximal Policy Optimisation (PPO) as the single agent learning algorithm to be used in this project since it has not been applied to HULH yet, and also shows promise in finding a NE strategy.

The underlying neural network structure for implementing the PPO consists of two main components: a policy network and a value network, sometimes called actor and critic networks (Charlesworth, 2018). These are shown in Figure 6 as the two neural networks third in from the left. The two outputs of this displayed structure is a softmax distribution of available actions, in addition to a value between -1 and 1 representing the calculated 'advantage' of the action.

The PPO algorithm is 'On policy' and directly updates the parameters of an agent's policy using gradient ascent on the estimated expected return, as opposed to estimating a direct value function. Notably there are two types: PPO-penalty and PPO-clip (Achiam, 2017). PPO-penalty explicitly penalizes for KL divergence, the difference between the probability distributions of the old and new policy, in its objective function. In comparison, PPO clip rather than penalizing for KL-divergence, uses a clipped surrogate objective function, shown in Equation 1, that will constrain the policy change into a narrow range using a clip, to stabilise training. This is ensured since, if the ratio of the new policy to the old policy is outside of clip range, the gradient is zero for the gradient ascent.

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

The algorithm will be imported from the Stable baseline 3 (SB3) library since this was successfully used by Pan et al., (2020) and also due to the fact SB3 algorithms have proven compatibility with Pettingzoo environments (Terry et al., 2021). This specific version[2] is a combination of the two types, with the addition of advantage normalisation to further stabilise training and increase convergence speeds.

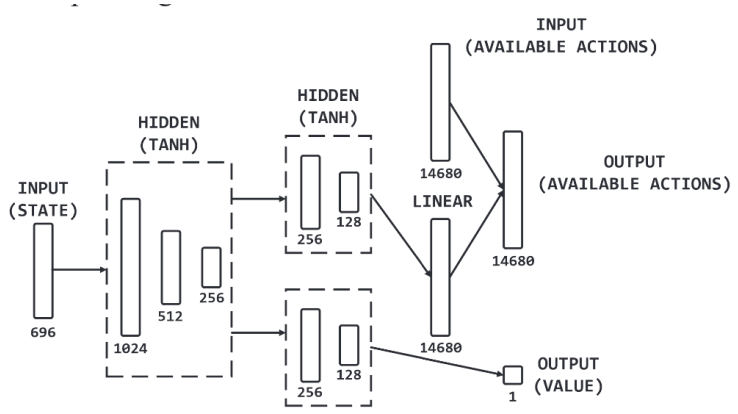


Figure 6: PPO Network Structure, extracted from Pan et al. (2022)

The modifications to the SB3 models mostly involved the provision of an action mask to the appropriate location in the model. This involved routing the action mask from the environment to the action network of the model. The input to this component of the PPO model is a vector of the the full observation space including the action mask, shown in the code in Section 13.7 of the Appendix. The action mask component of this vector was overridden such that all values were zero (lines 7-14). The rationale behind this was that the null values would not pollute the agent state observation and instead the action mask was explicitly inserted into the agents action network. Here, the 'forward_actor' function (lines 1-7) is called within 'self._get_action _dist _from _latent' of line 29. The actions output from the action distribution are modified such that illegal actions would have a minus negative infinity value which when passed through the softmax layer, would result in illegal actions having a probability of zero.

[2]https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html

7.10 Optimal Action

An optimal action tool was created with the purpose of supplementing the loss curves and mean reward graphs to evaluate the training of the agent. During certain stages of the game when there are a minimum of 2 player cards and 3 community cards, the optimal action is pre-calculated and compared to the action decided by the agent. The optimal action is calculated by converting the cards into their highest possible ranking configuration, and converting this into a numerical score by the imported Treys library[3]. The best combination of cards, the royal flush, has a score of 1 whilst the worst combination, unsuited, has a score of 7462.

As shown in Figure 7, this is split up into quarters. If a score lies in the first quarter, the optimal action is logically to raise, the second quarter would be to call, third quarter to check, and last quarter to fold. As such, this tool logs the percentages of optimal actions taken by the agent according to this heuristic and the code for this is shown in Section 13.8 of the Appendix.

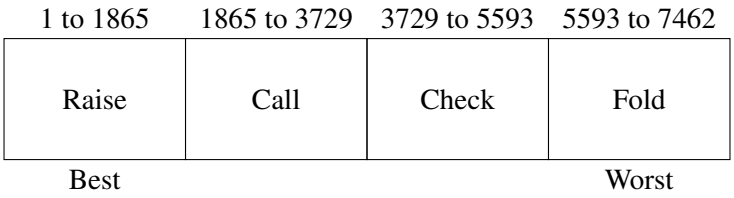


Figure 7: Optimal Actions based on Scores

7.11 Human opponent interface

With the inclusion of a human opponent, an interface and corresponding functionality was implemented. This initialises the opponents policy as human, provides the player with the game information and steps the environment with the action provided by the human opponent. Upon completion of the specified number of games, the reward distribution for both players and their mean rewards are returned.

Whilst Pettingzoo does have a human render mode, this also shows the opponents private hand to the player and as such a custom-built input interface was used and is shown in Figure 8 below. Similarly, the human agent[4] class provided by RLcard is not compatible with the custom wrapper architecture of the environment, so this was replaced with self made functionality shown in the code of Section 13.9 of the Appendix.

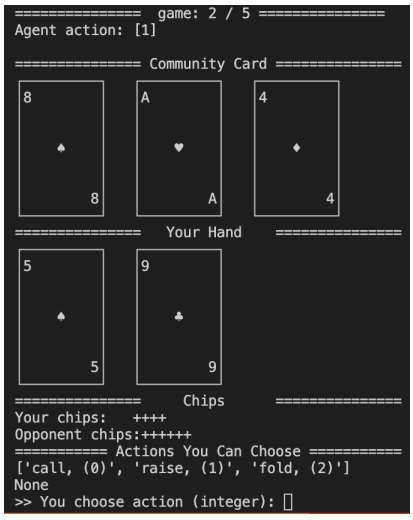


Figure 8: Human player interface screenshot

[3]https://github.com/ihendley/treys

[4]https://rlcard.org/rlcard.agents.html# module-rlcard.agents.human_agents.limit_holdem _human_agent

8 Implementation and results

8.1 Hyperparameter tuning results

The optimal primary and secondary hyperparameters for the PPO and A2C model shown in Table 1 and Table 2 below. There was a large difference in the optimal primary hyperparameters between the two algorithms, other than the optimal values for value function (vf) and entropy coefficients as expected, Whilst the optimal values for secondary hyperparameters are also very similar.

Model	Batch Size	Steps	Activation	Optimiser	Learning Rate	Epochs	Entropy Coef	Vf Coef	Mean Reward
PPO	32	3072	Tanh	Adam	0.0058	70.0	0.0025	0.25	4.02
A2C	na	10000	ReLU	RMSprop	0.0457	na	0.0025	0.25	3.23

Table 1: Optimal primary hyperparameters

Model	GAE lambda	Clip range	Normalize advantage	Max grad norm
PPO	0.95	0.1	False	0.6
A2C	0.85	na	False	0.5

Table 2: Optimal secondary hyperparameters

There was a significant difference in the most important hyperparameter for the PPO with, as shown in Figure 9 (Top), the optimiser accounting for 53% compared to the 10% importance of the second most important hyperparameter of learning rate. The A2C hyperparameter importance was less concentrated with the value function coefficient being the most important at 35%. Other than learning rate, there is not much similarity in the importance ranking between the two algorithms. This divergence could stem from inherent differences between the algorithms, or due to the assumed training convergence point and optimal state observation representation for the A2C. Notably, the latter of the two proposed explanations may also explain why, as shown in Figure 21 of the Appendix, the large majority of the A2C trials performed very poorly.

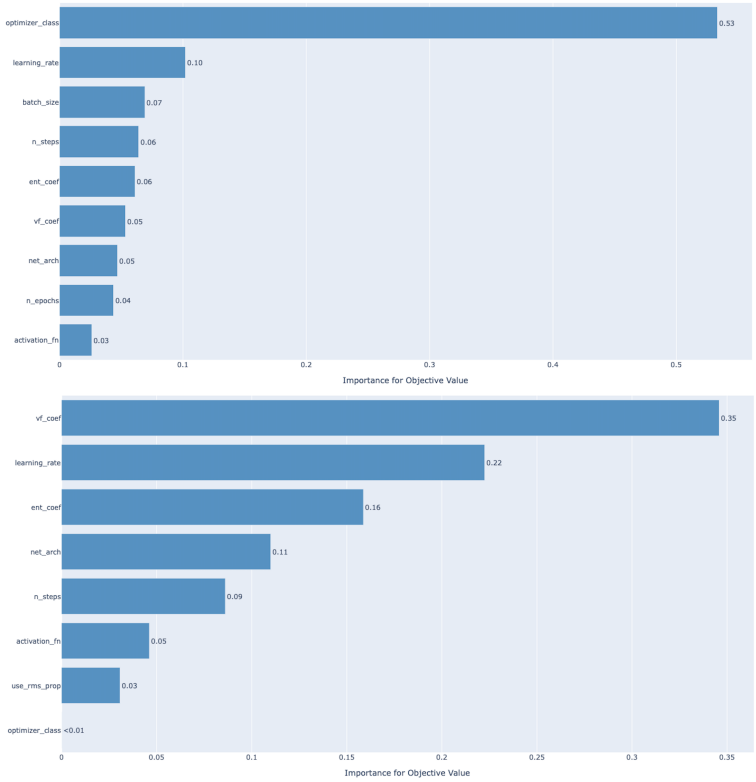


Figure 9: PPO (Top) and A2C (Bottom) primary hyperparameter importance

8.2 Self-play results

For the self-play process, the first opponent is one using a random policy, and after the agent has learnt for a period of time equal to the expected training convergence point at approximately 30,000 steps, the policy is transferred to the opponent and the agent then undergoes the next era of learning in which it is effectively playing against itself. For the PPO, the per generation number of training steps was set at 30,720 since this is divisible by the optimal number of replay buffer steps (3072). Since the A2C has an optimal number of steps equal to 10,000 the per generation training steps was set at 30,000. This cycle of policy transfer and learning is repeated for a total of 10 generations, and each vertical line in Figures 11 represent the policy transfer to the next generation. Each PPO generation took approximately 15 minutes to train, where as the A2C was significantly quicker at 90 seconds.

The performance of the generation 0 agent is showed by the the moving reward(cumulative total) of Figure 10 and moving mean of Figure 11. The green line shows the approximate mean reward from an untrained agent as a benchmark. The results here is as expected: the first generation has the highest performance since its opponent is only using a random policy. Following this, the moving reward in Figure 10 and moving mean in Figure 11 both deteriorate as its opponents policy improves throughout the generation updates. Most notably, the convergence to an almost stable mean reward during training and evaluation can be interpreted as the self-play agent reaching a possible Nash Equilibrium strategy.

Figure 10: PPO Training moving reward across 10 generation updates

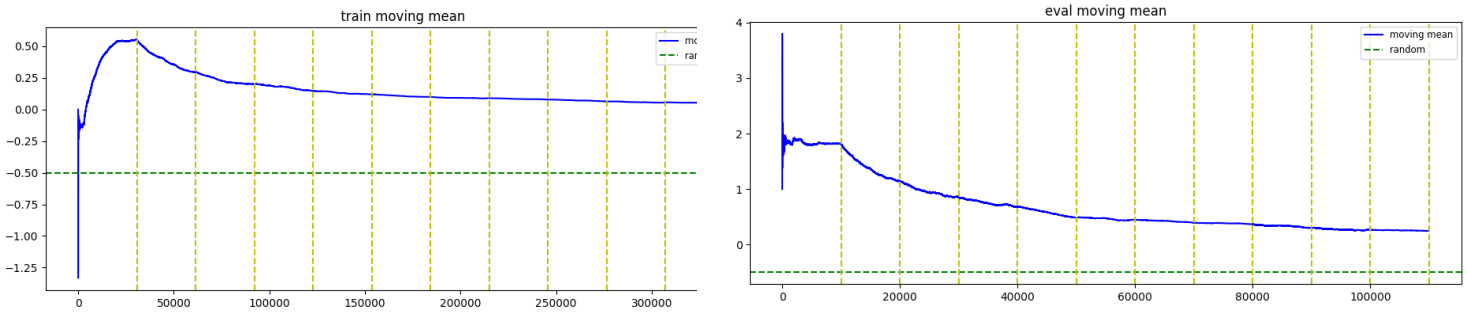


Figure 11: PPO moving mean for training (left) and evaluation (right)

8.3 Observation space amendment results

Figure 13.4 of the appendix shows the results from the preliminary observation space amendment experiment, to which the moving reward for the 72+ modified state space was more advantageous to learning against a trained opponent. During this experiment, it was observed that there was no significant difference in the state representation when trained against a random agent; likely due to a random opponents inability to force the agent to learn the most optimal policy. In terms of the material impact this had in the final self play training, comparing Figure 12 with Figure 11 (right) shows that the default observation type resulted in

poorer performance, most notably during the earlier generations. This is qualified with the 10th generation agent using the default 72 observation space type achieving a mean reward of 1.03 against a random opponent[5], in comparison to the agent using the amended 72+ observation space type achieving a mean reward of 2.12.

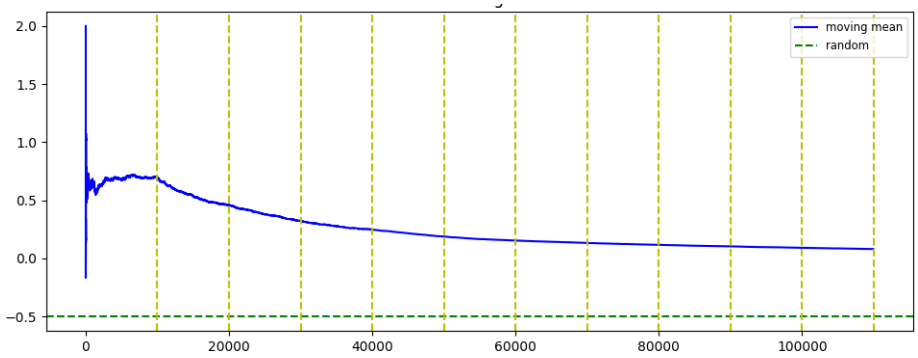


Figure 12: Evaluation moving mean for self-play agent supplied with default observation space

8.4 Loss curves

The use of the loss curve to determine the extent of a learned strategy proved reasonably effective during self-play training. As shown in Figure 14, the overall loss is composed of three values: Value loss, policy loss and entropy loss. The PPO algorithm has value function coefficient (α) and entropy coefficient (β) hyperparameters which are used to vary the proportion that value function function loss and entropy loss contribute to the overall loss.

$$loss = \alpha \cdot loss_{value} + loss_{policy} + \beta \cdot loss_{entropy} \quad (2)$$

$$loss_{value} = (value - return)^2 \quad (3)$$

$$loss_{policy} = - advantage \cdot e^{\pi'_\theta - \pi_\theta} \quad (4)$$

$$loss_{entropy} = - \pi_\theta \cdot \log \pi_\theta \quad (5)$$

Figure 13: Loss equations extracted from Pan et al. (2022)

Value loss was isolated and plotted, as done in Pan et al.(2022), along with the other two loss values in Figure 14. It should be noted that these are the raw loss values; they have not been multiplied by the coefficients present in the clipped surrogate objective function.

Figure 14: Value, policy and entropy loss throughout the self-play process

[5]These two agents can not run in the same game because they have different observation spaces and the environment only supports one.

The expected PPO value loss curve "should increase while the agent is learning, and then decrease once the reward stabilizes" (Unity Technologies, 2021), a trend displayed here albeit with a small region of stabilisation. Policy loss shows the difference between the desired policy behaviour and the behaviour of the one currently being used. Here, the policy loss curve exhibits a tighter range of values, likely the product of the clipping function of the PPO algorithm to limit the size of each policy update. In analysing the policy loss curve shown, "The magnitude of this should decrease during a successful training session" (Unity Technologies, 2021) to which this trend is not present since it instead approximately fluctuates about zero, with a period of increased negativity during the final few generations

Entropy loss measures the extent of randomness present in the policy and "Should slowly decrease during a successful training process" (Unity Technologies, 2021), which similarly is not present here. In fact, entropy loss is essentially constant other than increasing during the training of the first generation, explained through the agent increasing the amount of exploration of the environment. The observation can be made that the loss curves of Figure 14 appear relatively underpopulated considering the number of training steps involved. This is caused by the infrequent value network updates relative to the overall training steps. For example, the typical 30,720 training steps and rollout buffer size of 3072 would result in 10 loss values. This is another consequence of the environment not being able to run in parallel since the rollout buffer would be filled in fewer total training steps.

8.5 Nash Equilibrium analysis

In order to further analyse the extent to which a NE strategy has been achieved, past that available from the converging mean reward shown in Figure 11, best response regret analysis is applied. Utilizing concepts from game theory, once training has been completed, the mean reward achieved by $agent_{i-1}$ against opponent $agent_{i-2}$ of the previous generation is the equilibrium payoff. After this, the mean reward achieved by $agent_i$ against opponent $agent_{i-1}$ is the best response payoff. The difference between the equilibrium payoff and best response payoff provides the regret of $agent_i$. These equations are provided in Box (2) below.

$$\begin{aligned} \textbf{Equilibrium (EQ) Payoff:} \quad & \text{EQ\_payoff}_i = \text{Trained\_learner}_{i-1} \text{ vs Trained\_learner}_{i-2} \\ \textbf{Best Response (BR) Payoff:} \quad & \text{BR\_payoff}_i = \text{Trained\_learner}_i \text{ vs Trained\_learner}_{i-1} \\ \textbf{Regret:} \quad & \text{Regret}_i = \text{BR\_payoff}_i - \text{EQ\_payoff}_i \end{aligned} \tag{2}$$

As shown in Figure 15 below, the regret curve is roughly sinusoidal about zero exhibiting a degree of inconsistency of the next generation to find a best response strategy. This is expected given the complexity of the game and a more extensive training session which doesn't limit the amount of per generation training to the expected convergence point at 30,000 steps could lead to the agent discovering a more effective best response strategy. Furthermore, Figure 15 shows that some generations failed to discover a best response that was better than the previous generation, it is implied that certain points of the self-play training near a state of "resource-constrained equilibrium", indicating that the agent faces challenges in finding a superior best response using the available resources from the 30,000 training steps.

Additionally, separate from the availability of the agent to discover a superior best response strategy during training, the accuracy's of the regret values is dependent upon the chosen number of evaluation steps. 10,000 steps was used for the evaluation period of each fully trained generation model. Given longer evaluation periods, it is possible the regret values would be more representative of the true value and the moving mean regret would fully converge to zero.

Importantly, the displayed convergence to approximately zero provides strong evidence that an equilibrium strategy has been reached. Specifically this means that a subsequent generation can not learn a strategy better than the one currently being used. Consequently, the strategy used by the final model 10 agent can, by definition, be classified as an optimal strategy (quote). It should be noted that this interpretation using the moving mean is limited since it would be regret of the model 10 agent being zero which fully displays

a Nash Equilibrium has been reached. As such, Section 8.6 analysing KL divergence throughout self-play seeks to supplement this.

Figure 15: Regret and moving mean regret for each generation agent across the self-play training

8.6 KL divergence throughout self-play

Kullback-Leibler (KL) divergence, measures the difference between two probability distributions, and whilst a core component of the SB3 version of the PPO algorithm, it can also can be used as a tool to analyse the convergence of the self-play training generations towards a NE strategy. Here, each trained generation of agent is paired with its respective opponent, and both are supplied the same observation. The probability distribution over the action space is extracted from the models and used to calculate the KL divergence. Six observations are given to each pair, a set composed of the two rarest observations (royal flush and straight flush), two partially rare (flush and straight flush) and then two of the most common observations (high card and pair). The KL divergence of each generation pair for this set of observations is then calculated and monitored throughout the self-play process. Table 3 below shows that after 7 generations, the KL divergence becomes zero for all 6 observations. Referring back to Figure 15 above, model 6 and model 7 also show the smallest amount of regret corroborating this finding.

This can be interpreted as a convergence point where for the set of the 6 artificially made observations, the agent and the opponent have the exact same action probability distribution and as such, supplements the mean reward analysis of Section 8.2 with further evidence that an equilibrium strategy has been achieved. It should be noted that convergence shown through decreasing KL divergence values towards zero is not as smooth as expected, especially with the sudden increase in KL divergence between generations 5 and 6. A possible explanation for this is that its caused by the inherit non-exhaustive nature of this tool, a limitation discussed in Section 9.2

8.7 Action optimality

Using the optimal action tool described in Section 7.10, the action optimality throughout the self-play process can be analysed. As shown in Figure 16, whilst there is no smooth increase in optimality across the self-play training, the final generation agent achieved a level of optimality higher than all its predecessors of approximately 26%; an action optimality rate that is surprisingly low considering the high performance

	KL divergence of Agent and opponent pair								
Observation	G0, G1	G1, G2	G2, G3	G3, G4	G5, G6	G6, G7	G7, G8	G8, G9	G9, G10
High Card	5.031×10^{-6}	0.003	25.10	7.52×10^{-14}	25.33	25.3284	0	0	0
Pair	3.58	0.27	0.0003	0.006	4.34×10^{-10}	1.40×10^{-9}	0	0	0
Straight	25.30	0.0007	8.76	0.002	20.23	25.33	0	0	0
Flush	0.003	0.084	3.50×10^{-20}	5.43999	0.09	25.33	0	0	0
Straight Flush	24.73	25.33	25.29	0	25.33	25.33	0	0	0
Royal Flush	25.33	1.13978×10^{-13}	1.19712×10^{-13}	0	25.33	25.33	0	0	0

Table 3: KL divergence for agent and opponent pair throughout self-play training

of the agent in the subsequent sections. It should also be noted that the spikes present in the figure are due to the lack of stabilisation in the percentage calculation as a new generation is trained. The surprisingly low optimal action rate can be attributed to the inherit flaws of this tool, as opposed to poor performance of the agent. Section 9.2 details the limitations of this tool arising from its heuristic based nature and inability to be applied to all rounds of the game.

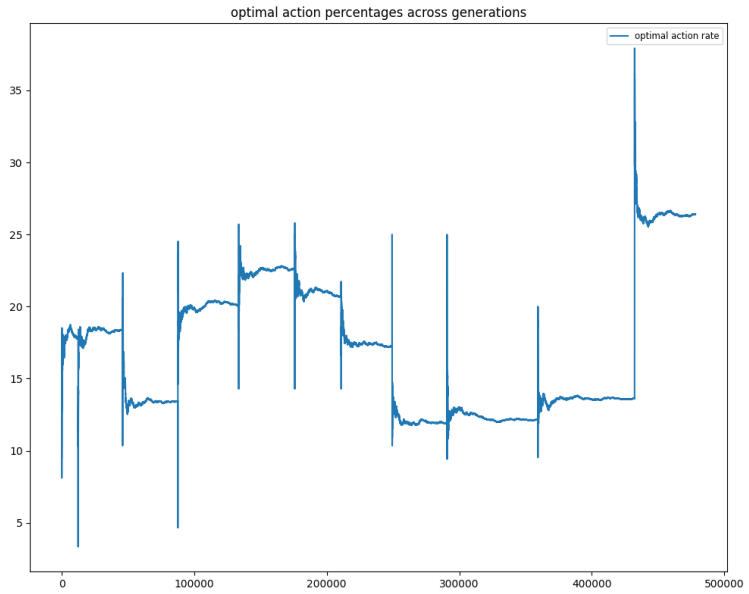


Figure 16: PPO mean rewards against the opponents

8.8 Performance evaluation against opponents

The trained PPO was evaluated against a random policy, self-play trained A2C, human player, and optimal action heuristic opponent each for 10,000 steps. The results are combined and shown in Figure 17 below.

8.9 Random policy opponent

The PPO achieved a mean reward of 2.12 against an opponent using a random policy, a large performance difference as expected. Attention should be raised to the fact this is marginally higher than the mean evaluation reward of 1.87 achieved by the generation 0 self-play agent shown in Figure 11 (right). This is interpreted as the random agent is unable to force the agent to learn a strong strategy resulting in the mean reward of 2.12, where as the latter generation opponents force the agent to learn the best possible strategy: resulting in the observed 2.12 mean reward by the 10th generation agent. Since the observation space amendment experiment of Section 8.3 raised the point that a random opponent was not able to cause any difference in the trialed observation space amendments, this provides evidence for the explanation that the random opponent is unable to force the agent to learn an optimal policy. Another observation is that the mean reward of 2.12 achieved by the self-play trained PPO is significantly larger than the 0.5 achieved when the agent was trained using the asymmetric initial policy (AIP) approach described in Section 7.6. This highlights the importance of the symmetric initial policy (SIP) approach to self-play training.

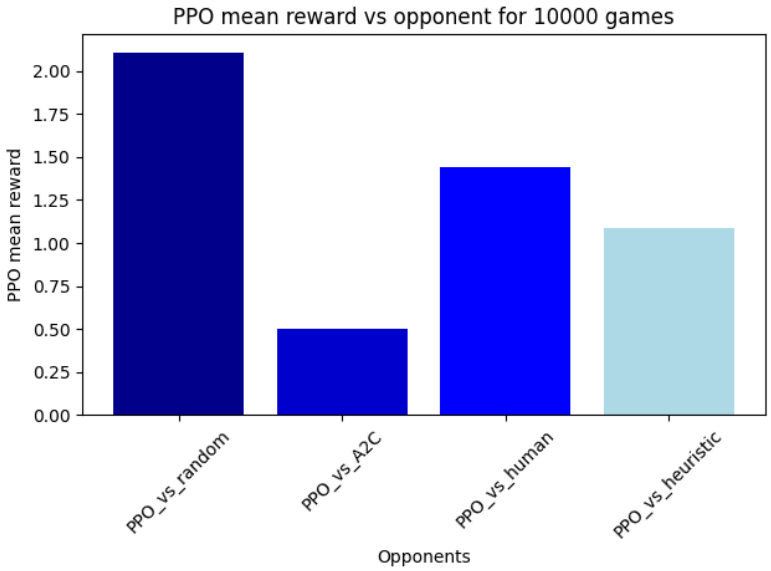


Figure 17: PPO mean rewards against the opponents

8.10 Self-play trained A2C opponent

The A2C opponent underwent the same extent of hyperparameter tuning and self-play training process as the PPO to ensure a fair comparison of the two similar actor critic, policy gradient ascent algorithms. This involved an A2C model who equally trained with 10 generation updates with 30,000 training steps per generation. As shown in Figure 17. The PPO outperformed the A2C with a mean reward of 0.5. The higher performance of the PPO is roughly proportional to the more complex features the algorithm implements relative to the A2C, specifically the reward clipping and a surrogate loss function. When the AIP self-play training approach is used, the A2C significantly outperforms the PPO, further reinforcing the importance of SIP self-play training.

8.11 Human player opponent

The trained PPO played against the owner of this project, a beginner human player for 50 games to which the reward distribution is shown in Figure 18 below. The trained PPO outperformed the human player with a mean reward of 1.44. As such in relativistic terms, the trained PPO has an equivalent skill level to that of an intermediate player. Yet since there is no opponent of a higher level to test it against, a limitation discussed in Section 9.2, it is possible that it has achieved a relative skill level of an advanced player. Since a NE strategy is described as being the optimal strategy for poker, (He et al, 2022) if the agent has discovered this strategy as suggested in Sections 8.2 to 8.6, or even a resource constrained strategy that is proximal to NE, this would promote the relative skill level to an advanced player.

8.12 Optimal action heuristic opponent

With the PPO performance exceeding that of all the previous opponents, a more skillful opponent is required to establish the upper bound of the trained PPO algorithms ability. Taking optimal action heuristic from the optimal action metric tool described in Section 7.10, an opponent was made which used this heuristic as the basis for its policy. Since the poker hand evaluator only applies to the latter rounds after the first betting round, the opponent is forced to choose a random action during the prior stage. When the PPO plays against this optimal action heuristic based opponent, it received a reward of 1.09. This attests to the optimal NE strategy used by the PPO, whilst also raising the limitations of the heuristic based policy, as discussed in Section 9.2. Despite this, the heuristic based opponent still performed better than the human agent against the PPO classifying its relative skill level as an intermediate player.

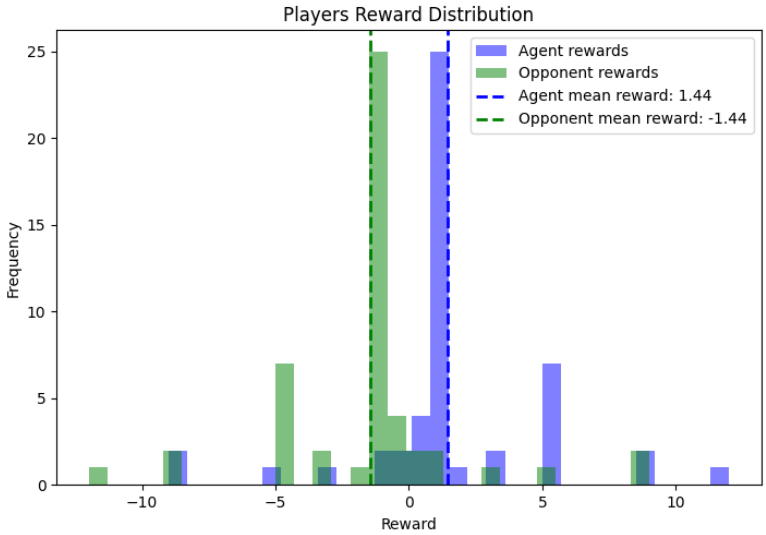


Figure 18: PPO mean rewards against the human opponent

8.13 Failed implementations

During the course of this project, there were a few design features that did not work when implemented. First, for the KL divergence tool for NE analysis used in Section 8.6, the preceding design did not use KL divergence, but instead attempted to measure the similarity in the probability distribution's using cosine similarity. The failure of this tool was caused by the inherit inability for cosine similarity to be used for probability distributions, hence the KL divergence was used in its place.

Second, before the data storage and plotting tool was implemented, there was an attempt to leverage the pre-made Tensorboard integration available through SB3 [6]. Unfamiliarity with Tensorboard library and the difficulties encountered in setting it up led to the use of the self-made data storage and plotting tool in its place.

Next, when trying to provide the trained PPO with a better performing opponent to determine the upper bound of its performance; a Perfect Information Game (PIG) agent was created. This agent, unlike the Imperfect Information Game (IIG) agents preceding it, has the opponents cards included in its observation space, grating it access to perfect information. The first attempt of this involved amending the 72+ observation type to include the opponents cards by indexing them with a 3, the cards of its own hand with a 2 and community cards as a 1. This resulted in a self-trained agent that performed poorer than the IIG PPO. The posited reason for this is the observation space became too compact for the agent to learn from.

Attempting to improve this, an amended observation type of 124+ was trialed; first 52 positions indexes the community cards with a 1, the players hand with a 2, and then the subsequent 52 positions indexes the opponents cards with a 3. This was trained for the same amount of training steps per generation (30, 720) and the same number of generations (10) yet with default hyperparameters. This similarly failed to perform better than the IIG agent, achieving a mean reward of 2.04 against a random opponent whereas the IIG agent achieved 2.12. The posited explanation behind this is that 30720 was not enough training steps for a better strategy to be learnt due to the PIG information content present in the larger observation space not being internalised into the policy. The intention was to calculate the KL divergence between the final generation agent with this PIG agent, along with their comparative performance against a random agent[7], to give an indication of the upper bound of the trained PPO's performance.

[6] https://stable-baselines3.readthedocs.io/en/master/guide/tensorboard.html

[7] both cant run in the same game because they have different observation spaces and the environment only supports one.

9 Evaluation

9.1 Strengths and merits

Overall, gauged by the fulfillment of its aims and objectives, this project as been a success. The project achieved the aims of creating a reinforcement learning environment of the HULH game. Similarly the aim of deploying, tuning and training the PPO algorithm was fully achieved. In terms of the hyperparameter tuning objective, this was more comprehensive than originally planned due to the addition of a second tuning stage for secondary hyperparameters. This included the use of normalised advantage and the optimal values for GAE lamda and clip range. These were not foreseen during the project specification and design since they naturally were not the focus of hyperparameter tuning discussions in the relevant papers.

The primary aim of investigation whether the PPO algorithm was capable of reaching a Nash equilibrium strategy was achieved with the result that it was able to converge to a proximal NE strategy. This was verified through the convergence shown in the self-play mean reward discussed in Section 8.2, the Kl divergence of zero for last generation pair analysed Section 8.6 and chiefly the moving mean regret converging to approximately zero in Section 8.5.

This project also implemented procedures not encountered in the papers visited during the construction of the literature review. The importance of SIP rather than AIP for self-play training that results in an optimal equilibrium strategy is not discussed. Similarly, the implementation of the KL divergence to supplement the NE analysis was not found in the papers reviewed. As such, this project contributes this knowledge to the field of using self-play to reach a NE strategy.

9.2 Assumptions and limitations

There exists some assumptions and limitations that should be raised. Whilst this project achieved its aim of tuning and training an A2C based opponent, the assumption was made that the A2C would be subject to the same training convergence points during primary hyperparameter tuning and that it shared the same optimal observation space format of 72+. Therefore, a possible extension would be to have tested this separately. Yet, the performance of the A2C proved this not to be detrimental since it achieved a level of performance proportional to complexity of the algorithm relative to the PPO algorithm

Another limitation surrounds the use of the optimal action tool. It should be noted that this is not an exacting method but is based on an approximate heuristic. Additionally, due to the nature of the imported library being limited to evaluating a minimum of 2 player cards and 3 community cards, the action taken by the agent in the interim pre-flop round are not subject to an optimality evaluation, rendering this tool non-extensive. Despite this, this tool was not the primary method for assessing performance but adds value as a supplementary performance metric. Similarly, the KL divergence tool to analyse the policy similarity of the self-play generations as they converge to a NE strategy can be described as non-exhaustive. This is because a set of 6 observations are supplied to the agent and opponent, when in fact there are 10 possible poker hand observations in total. As previously mentioned, this could be a possible cause of the uneven convergence shown in Table 3. Despite this, this limitation is not particularly detrimental since the set of 6 observations that are used in this tool cover the full range of rarity.

The nature of this project involved determining the number of Self-play generations required for NE strategy to be achieved. This was done manually by iterativley entering an arbitrary number of generations, observing the resultant moving meaning reward graph, and increasing the number of generation if no convergence was observed. Naturally this is not an optimal approach and a better solution would be to implement a callback to stop self-play training when the extracted KL divergence or regret for a generation pair falls below a certain threshold, say 0.5. This forms the primary alternative approach if a similar project is to be completed in the future.

Finally, arguably the main limitation is that the upper bound of the self-play trained PP0's performance was not established. This was primarily due to the failure of implementing a PIG agent that could outperform the IIG agent, but also due to the lack of availability of publicly available pre-trained RL algorithms for HULH poker. Cepheus, the CFR based HULH agent from Bowling et al. (2015) is publicly available to query the model[8] and a game against the trained PPO of this project could have been constructed. When attempting this, unfortunately this service is non-operational. As a result, the upper bound of the performance capability of this agent remains untested, giving rise to the biggest limitation of this project.

10 Learning points and professional issues

This project proved effective in testing both my level of programming skills and the other general skills required to complete a project of this nature to a high standard. The main programming skill which was further developed through this project was my understanding and implementation of Python class inheritance. The primary areas which required this was the design and creation of the custom wrapper outlined in Section 7.5 which remained compatible with all the other related classes shown in Figure 3 and 4.

A very important skill developed in the course of this project was the use of GitHub. This was used as way to safely backup the code to cloud storage, a risk contingency plan stated in the PSD, a way to work across multiple devices, and also to track and revert to changes in the code. Another skill developed was increased familiarity with the MatPlotLib library. A wide range of graphs, including bar charts, line graphs and histograms, were demanded by this project; ones to which beforehand I hadn't used before.

The modifications made to the SB3 and PettingZoo libraries required the working with third-party code for the first time. This developed my skill in being able to fully understand code designed by someone else, to the extent that the modifications made maintain the overall functionality of the third party code. In terms of the knowledge gained from completing this project, the inner workings of both the PPO and A2C algorithms was obtained. Completing the literature review also increased my knowledge surrounding the other RL algorithms used in this field, and the use real world use cases of Nash Equilibrium.

The most crucial actions of this project would be the creation of the self-play architecture. This was the most technically challenging component and did not rely on any imported libraries. Secondly, the design and implementation of the custom wrapper was also crucial in creating the desired environment that is compatible with the imported SB3 algorithms. If this project was to be undertaken again, the modifications to the SB3 and Pettingzoo library's would have been more effectively implemented using a class inheritance structure that implements the required modifications. This would have prevented the proliferation of amendments, mostly in regard to imports, to all the related files that led to the final code package of this project being quite large. Similarly on the topic of coding style, the data storage and plotting object was made with excessive use of the dictionary data type. Upon reflection, the use of classes could have resulted in less code. The final learning point arising from this project is the importance of searching for pre-existing solutions before implementing them myself. Here, lots of time could have been saved on modifying the SB3 algorithm to include action masking. It later became known that a pre-existing modified PPO that allows for action masking is available through a separate SB3 contrib library [9].

This section reflects upon the extent to which this project adheres to the British Computer Science code of conduct (BCS, 2022). In relation to 'Professional Competence and Integrity', this project stayed within its realm of competency. This was secured by choosing not to implement MARL algorithms and not attempting to restructure the RLcard HULH game to be paralleisable since they were unrealistic given my level of programming competency and time restrictions. Despite this, the project remained ambitious in its endeavours; implementing, tuning and comprehensively evaluating two single agent RL algorithms.

[8] http://poker.srv.ualberta.ca/strategy

[9] https://sb3-contrib.readthedocs.io/en/master/guide/examples.html#maskableppo

In regard to upholding the expected duty to public interests and third parties, the nature of this project involving no public participants or their private data and the absence of cooperation with third parties means this aspects of BCS code of conduct is not directly applicable. Classifying the authors of the imported libraries as third parties, full acknowledgment of any imported libraries is expressed and credit given to the designers. Finally, in relation to "respect and value alternative viewpoints and, seek, accept and offer honest criticisms of work" (BCS, 2022), feedback from the assigned supervisors was sort and on-boarded to the best of my ability, including for future undertakings. This involves ensuring text within presentations are clear and concise, and crucially, to be more careful with specified timings.

11 Conclusion

In summary, this project successfully deployed the PPO algorithm to a custom HULH environment, tuned the hyperparameters and trained the RL algorithms using a self-play training schedule in an attempt to learn an optimal NE strategy. The learnt equilibrium strategy was analysed using action optimality, best response regret analysis and KL divergence and the resulting performance was evaluated using a range of opponents, none of which could outperform the self-play trained PPO. These findings result in the conclusion that the agent was able to find a resource bound equilibrium strategy.

The aim to create a reinforcement learning environment of the HULH game through the use of a custom wrapper, detailed in Section 7.5, to modify the two player game into a single agent environment that adheres to standard gymnasium syntax was fully achieved. The aim to deploy and tune the PPO and A2C algorithms was exceeded due to the inclusion of secondary hyperparameters. The aim to evaluate the performance of the trained PPO agent was similarly exceeded through the use of key evaluation techniques from similar studies along with additional metrics. The aim of using loss curves to determine the extent of learning during self-play was partially achieved. Whilst the value loss curve conformed to the expected trend, the expected trends exhibited by the policy and entropy loss were not present. The identified cause for this are the infrequent updates arising from the large replay buffer size and inability for vectorised environments to increase the update frequency by decreasing the steps required to fill the buffer. Performance evaluation using a wide range of opponents was partially achieved since although two more opponents were included past that specified in the PSD, the unavailability of a superior opponent renders the upper bound of the PPO's ability undetermined. The aim to find the NE strategy of the game through self-play was exceeded. It was determined the agent achieved a resource bound equilibrium strategy, evidenced through the thorough analysis of mean reward curves, best response regret and KL divergence.

In this study, notable disparities emerged in both the optimal primary hyperparameters for the two algorithms and the relative importance of hyperparameters within each algorithm (Section 8.1). Furthermore, it became evident that the Proximal Policy Optimization (PPO) algorithm required a span of ten self-play generations to attain an equilibrium strategy, as evidenced by the convergence observed in training and evaluation curves (Section 8.2). Notably, a significant performance gain was provided when the observation space was amended from the default 72 to 72+, yielding a mean reward of 2.12 compared to the 1.03 when the default 72-observation space was employed (Section 8.3). Additionally, despite the inherent limitations arising from its heuristic nature, the analysis of optimal actions in Section 8.7 revealed that the 10th agent outperformed all preceding generations, attaining the highest percentage of optimal actions according to the heuristic.

Nash Equilibrium analysis using regret and best response strategies in Section 8.5 provided the key finding that the next generations ability to find a superior best response was inconsistent. This resulted in the conclusion that the 10th generation agent had discovered a resource bound strategy, not too distant from a NE strategy. As mentioned, this analysis was limited by two factors: limited training steps for a superior best response strategy to be discovered and limited evaluation steps impeding the accuracy of the extracted regret values. Its proximity to a true NE strategy was established by the KL divergence between generation pairs converging to zero after 7 generations, detailed in Section 8.6, and subsequently evidenced by its superior performance against all opponents it played against.

The literature review reiterates the point that for IIG's like HULH, there are no convergence guarantees in finding NE strategy through self play (Fu et al., 2022). This was confirmed by the findings of this project: a highly effective resource constrained equilibrium strategy was achieved, but not one that can be classed as a true NE. This is inline with the findings of the papers reviewed who applied the PPO algorithm to IIG's, where the lack of a found NE strategy might be due to the greater complexity of the games studied, and the fact they are non-zero sum making it harder for the algorithm to find a NE strategy because there's no direct mathematical relationship between the players' payoffs. This project also contributes to the field of finding NE through self-play by highlighting the importance of using symmetric rather than asymmetric initial policies, an aspect not investigated in the reviewed studies of the literature review.

Suggestions for future work arising from this project include firstly a comparative study between the A2C and PPO algorithm which involves equal attention to tuning each model. Secondly, since the upper-bound of the performance ability of the PPO remains untested, the suggestion arises to compare the performance against the other algorithms discussed in the literature review, such as NFSP or Deep CFR. Similarly, the successful use of a perfect information game agent could offer the means required to determine this upper-bound of the trained PPO algorithms performance. Overall, since all of the projects original aims were securely achieved and in some cases, extended; no significant shortcomings are recognised and it can be concluded that this project has been a success, with little departure from the original proposal.

12 Bibliography

Achiam, J. (2017). Proximal Policy Optimization — Spinning Up documentation. [online] Openai.com. Available at: https://spinningup.openai.com/en/latest/algorithms/ppo.html.

BCS. "BCS Code of Conduct | BCS." Www.bcs.org, 8 June 2022, www.bcs.org/membership-and-registrations/become-a-member/bcs-code-of-conduct/.

Bowling, M., Burch, N., Johanson, M. and Tammelin, O. (2015). Heads-up limit hold'em poker is solved. Science, 347(6218), pp.145–149. doi:https://doi.org/10.1126/science.1259433. (Accessed: 1 July 2023).

Brown, N. et al. (2019) 'Deep Counterfactual Regret Minimization', in Proceedings of the 36th International Conference on Machine Learning. International Conference on Machine Learning, PMLR, pp. 793–802. Available at: https://proceedings.mlr.press/v97/brown19b.html (Accessed: 1 July 2023).

Brown, N. and Sandholm, T. (2019) 'Superhuman AI for multiplayer poker', Science, 365(6456), pp. 885–890. Available at: https://doi.org/10.1126/science.aay2400. (Accessed: 2 July 2023)

Charlesworth, H. (2018) 'Application of Self-Play Reinforcement Learning to a Four-Player Game of Imperfect Information'. arXiv. Available at: http://arxiv.org/abs/1808.10442 (Accessed: 1 July 2023).

Fu, H., Liu, W., Wu, S., Wang, Y., Yang, T., Li, K., Xing, J., Li, B., Ma, B., Fu, Q. and Yang, W. (2022). ACTOR-CRITIC POLICY OPTIMIZATION IN A LARGE- SCALE IMPERFECT-INFORMATION GAME.(Accessed: 2 July 2023).

He, K. et al. (2022) 'Finding nash equilibrium for imperfect information games via fictitious play based on local regret minimization', International Journal of Intelligent Systems, 37(9), pp. 6152–6167. Available at: https://doi.org/10.1002/int.22837.(Accessed: 9 July 2023).

Heinrich, J. and Silver, D. (2016) 'Deep Reinforcement Learning from Self-Play in Imperfect-Information Games'. arXiv. Available at: http://arxiv.org/abs/1603.01121 (Accessed: 30 June 2023).

Heinrich, J. and Silver, D. (2015). Smooth UCT search in computer poker. pp.554–560.(Accessed: 4 July 2023).

Kuba, J.G. et al. (2022) 'Trust Region Policy Optimisation in Multi-Agent Reinforcement Learning'. arXiv. Available at: http://arxiv.org/abs/2109.11251 (Accessed: 4 July 2023).

Lanctot, M. et al. (2020) 'OpenSpiel: A Framework for Reinforcement Learning in Games'. arXiv. Available at: http://arxiv.org/abs/1908.09453 (Accessed: 30 June 2023).

Lazaridis, A. et al. (2022) 'AlphaBluff: An AI-Powered Heads-Up No-Limit Texas Hold'em Poker Video Game', in 2022 International Conference on INnovations in Intelligent SysTems and Applications (INISTA). pp. 1–6. Available at: https://doi.org/10.1109/INISTA55318.2022.9894244. (Accessed: 30 June 2023).

Li, H. and He, H. (2023) 'Multiagent Trust Region Policy Optimization', IEEE Transactions on Neural Networks and Learning Systems, pp. 1–15. Available at: https://doi.org/10.1109/TNNLS.2023.326 5358. (Accessed: 27 June 2023).

Lockhart, E. et al. (2020) 'Computing Approximate Equilibria in Sequential Adversarial Games by Exploitability Descent'. arXiv. Available at: http://arxiv.org/abs/1903.05614 (Accessed: 30 June 2023).

Pan, J. et al. (2022) 'Application of Deep Reinforcement Learning in Guandan Game', in 2022 34th Chinese Control and Decision Conference (CCDC). 2022 34th Chinese Control and Decision Conference (CCDC), pp. 3499–3504. Available at: https://doi.org/10.1109/CCDC55256.2022.10033565. (Accessed: 30 June 2023).

Shi, D. et al. (2022) 'Optimal Policy of Multiplayer Poker via Actor-Critic Reinforcement Learning', Entropy, 24(6), p. 774. Available at: https://doi.org/10.3390/e24060774. (Accessed: 29 June 2023).

Srinivasan, S. et al. (2018) 'Actor-Critic Policy Optimization in Partially Observable Multiagent Environments', ArXiv [Preprint]. Available at: https://www.semanticscholar.org/paper/90fac6e666cfbb2 c70 71610139ae65ef49066505 (Accessed: 1 July 2023).

Terry, J. et al. (2021) 'PettingZoo: Gym for Multi-Agent Reinforcement Learning', in Advances in Neural Information Processing Systems. Curran Associates, Inc., pp. 15032–15043. Available at: https://arxiv.org/abs/2009.14471 (Accessed: 13 July 2023).

Unity Technologies (2021). Using TensorBoard to Observe Training - Unity ML-Agents Toolkit. [online] unity-technologies.github.io. Available at: https://unity-technologies.github.io/ml-agents/Using-Tensorboard/ [Accessed 19 Sep. 2023]. Wang, K., Bai, D. and Zhou, Q. (2022) 'DeepHoldem: An Efficient End-to-End Texas Hold'em Artificial Intelligence Fusion of Algorithmic Game Theory and Game Information', in 2022 IEEE 8th International Conference on Computer and Communications (ICCC), pp. 2313–2320. Available at: https://doi.org/10.1109/ICCC56324.2022.10065796. (Accessed: 29 June 2023).

Yang, S. et al. (2023) 'Reinforcement Learning Agents Playing Ticket to Ride–A Complex Imperfect Information Board Game With Delayed Rewards', IEEE Access, 11, pp. 60737–60757. Available at: https://doi.org/10.1109/ACCESS.2023.3287100. (Accessed: 6 July 2023).

Zhong, Y., Zhou, Y. and Peng, J. (2020) 'Efficient Competitive Self-Play Policy Optimization', ArXiv [Preprint]. Available at: https://www.semanticscholar.org/paper/252b1f4f3bfbffe8400f1cfd3a2a95d93 2e959e0 (Accessed: 4 July 2023).

13 Appendix

13.1 Training convergence test graph

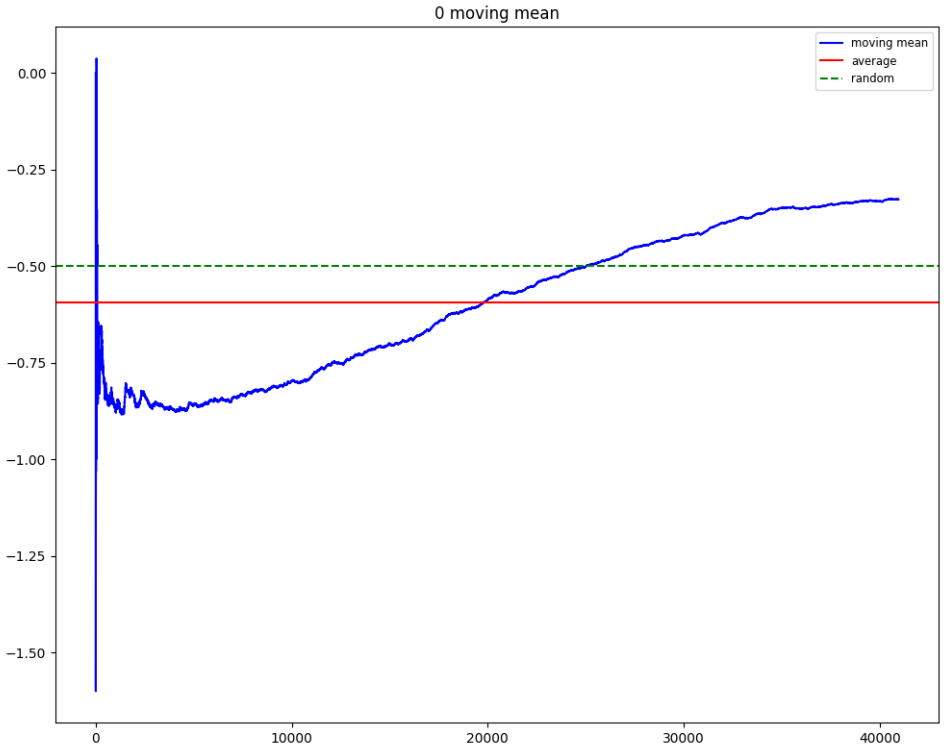


Figure 19: Movinf mean of default PPO trained for 40000 steps

13.2 Primary Hyperparameter search

Table 4: Hyperparameters

Hyperparameter	**Values**
batch_size	32, 64, 128, 256, 512, 1024
n_steps	1024, 2048, 3072, 4096, 5120
learning_rate	log uniform between 1e-6 - 1]
n_epochs	10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
ent_coef	0.000125, 0.0025, 0.005, 0.01, 0.02, 0.03
vf_coef	0.25, 0.5, 0.75, 0.80, 0.85
activation_fn	nn.ReLU, nn.Tanh
optimizer_class	th.optim.Adam, th.optim.SGD
net_arch	pi:[64,64], vf: [64,64], pi:[256], vf: [256], pi:[256,128], vf': [256,128]]

13.3 Secondary Hyperparameter search

Hyperparameter	Values
`gae_lambda`	0.85, 0.90, 0.95
`clip_range`	0.1, 0.2, 0.3
`normalize_advantage`	True, False
`max_grad_norm`	0.3, 0.4, 0.5, 0.6

Table 5: Secondary Hyperparameter Values

13.4 Observation space amendment results

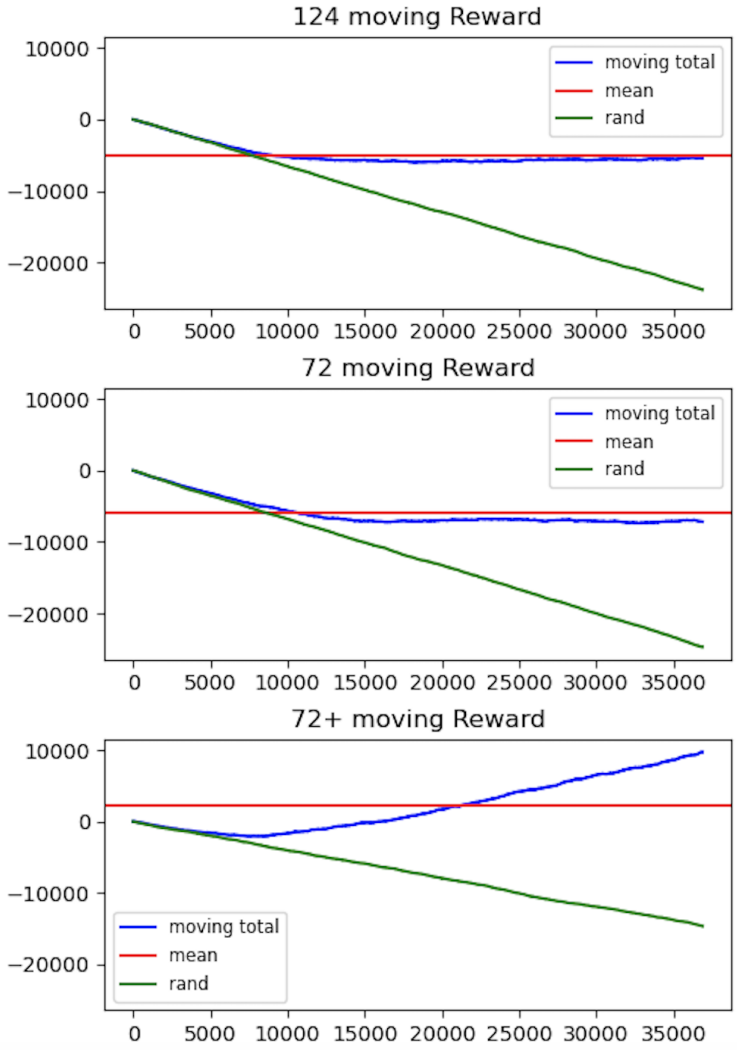


Figure 20: Observation space experiment results

13.5 Code: Environment step method of custom wrapper

```
    def step(self, action):
        #calculate if agent chose the optimal action
        self.optimal_action(action, self.AGENT)
        #step environment with agents action
        super().step(action)
        # if its the opponents turn and not truncated or terminated
        if self.agent_selection != self.learner and not self.observe()[2] and not self
    .observe()[3]:
            op_action_mask = self.observe()[0]['action_mask']
            op_obs = super().observe(self.baked)
            # retrive opponnent action from its private observation
            ops_action = self.OPPONENT.get_action(op_obs)
            # calculate if opponent chose the optimal action
            self.optimal_action(action, self.OPPONENT)
            #step environment with opponents action
            super().step(ops_action)
            op_reward = self._cumulative_rewards[self.baked]
            self.OPPONENT.rewardz.append(op_reward)
        # return agents observation
        return self.observe()
    %
```

Listing 1: custom wrapper step method

13.6 Code: Self-play architecture

It should be noted that extraneous code, such as updating the data storage objects with the rewards, and the training of a random agent for a benchmark are not included for brevity.

```
def run(self, eval_opponent_random):
    n_gens = self.n_gens
    for gen in range(0, n_gens+1):
        if gen == 0:
            env = self.env
            env.AGENT.model = self.base_model
            env.AGENT.policy = self.env.AGENT.policy
            # train
            env.AGENT.model.learn(dumb_mode = False)
            # save policy
            root = self.gen_lib[gen]
            env.AGENT.model.save(path=root, include = ['policy'])
            # evaluate
            Eval_env = texas_holdem.env(self.obs_type, render_mode = "rgb_array")
            Eval_env = Monitor(Eval_env)
            Eval_env.OPPONENT.policy = 'random'
            Eval_env.AGENT.policy = self.env.AGENT.policy

            mean_reward_ag,episode_rewards_ag, episode_lengths= evaluate_policy(
            env.AGENT.model, Eval_env)
        else:
            prev_gen_path = self.gen_lib[gen-1]
            # init agent model
            if self.model == 'PPO':
                env.AGENT.model  = PPO('MultiInputPolicy', env, **hyperparams)
                self.env.AGENT.policy = 'PPO'
            elif self.model == 'A2C':
                env.AGENT.model  = A2C('MultiInputPolicy', env, **hyperparams)
                self.env.AGENT.policy = 'A2C'
            # init opponent model
            if self.model == 'PPO':
                env.OPPONENT.model  = PPO('MultiInputPolicy', env, **hyperparams)
                self.env.OPPONENT.policy = 'PPO'
            elif self.model == 'A2C':
```

```
                env.OPPONENT.model  = A2C('MultiInputPolicy', env, **hyperparams
                self.env.OPPONENT.policy = 'A2C'
            # load prev gen params to opponent and agent
            env.OPPONENT.model.set_parameters(load_path_or_dict= prev_gen_path)
            env.AGENT.model.set_parameters(load_path_or_dict= prev_gen_path)
            # train
            env.AGENT.model.learn(dumb_mode= False)
            # save pol
            env.AGENT.model.save(self.gen_lib[gen], include = ['policy'])
            env.reset()
            #evaluate compared to oppponent
            print("eval", gen)
            Eval_env = texas_holdem.env(self.obs_type)
            Eval_env.AGENT.policy = self.base_model.policy
            Eval_env.OPPONENT.policy = self.model
            Eval_env.OPPONENT.model = env.OPPONENT.model

            mean_reward,episode_rewards= evaluate_policy(env.AGENT.model, Eval_env)
```

Listing 2: custom wrapper step method

13.7 Code: PPO modifications

```
def forward_actor(self, features: th.Tensor, action_msk_pass) -> th.Tensor:
        forward_actor_output = self.policy_net(features)
        adjusted_output = self.format_action_mask(action_msk_pass,
    forward_actor_output)

        return adjusted_output

def format_action_mask(self, tensor1, tensor2):
        try:
            processed_tensor = th.where(tensor1 == 1, tensor2, float('-inf'))
        except TypeError:
            tensor1 = th.tensor(tensor1)
            tensor2 = tensor2.detach()
            processed_tensor = th.where(tensor1 == 1, tensor2, float('-inf'))
        return processed_tensor

def forward(self, obs: th.Tensor, deterministic: bool = False) -> Tuple[th.Tensor, th.
    Tensor, th.Tensor]:
        # Preprocess the observation if needed
        action_msk_pass = obs['action_mask']
        % nullify remenant action mask in observation
        obs['action_mask'] = th.zeros((1,4))
        features = self.extract_features(obs)
        % forward pass in all the networks (actor and critic)
        if self.share_features_extractor:
            latent_pi, latent_vf = self.mlp_extractor(features, action_msk_pass))
        # Evaluate the values for the given observations
        values = self.value_net(latent_vf)

        distribution = self._get_action_dist_from_latent(latent_pi, action_msk_pass)
        actions = distribution.get_actions(deterministic=deterministic)
        log_prob = distribution.log_prob(actions)
        actions = actions.reshape((-1, *self.action_space.shape))
        return actions, values, log_prob
```

Listing 3: custom wrapper step method

13.8 Code: Optimal action tool

```
def optimal_action(self, action, player):
    score_max = 7462
    quartiles = [score_max * 0.25, score_max * 0.5, score_max * 0.75]
    game = self.game
    hand = []
    for c in player.hand:
        c1r = c.rank
        c1s = c.suit.lower()
        c1 = c1r +  c1s
        hand.append(c1)
    pc = []
    if len(game.public_cards) > 0:
        public_cards = game.public_cards

        for c in public_cards:
            cr_temp = c.rank
            cs_temp = c.suit.lower()
            pc.append(cr_temp +  cs_temp)
    hand_objs = []
    pc_objs = []
    for c in hand:
        hand_objs.append(Card.new(c))
    for c in pc:
        pc_objs.append(Card.new(c))

    if len(pc) >= 3:
        evaluator = Evaluator()
        try:
            score = evaluator.evaluate(hand_objs, pc_objs)
        except:
            KeyError
            score = 0

        if score <= quartiles[0]:
            op_act = 1
        if score >= quartiles[0] and score <= quartiles[1]:
            op_act = 0
        if score >= quartiles[1] and score <= quartiles[2]:
            op_act = 3
        if score >= quartiles[2]:
            op_act = 2
        if action == op_act:
            if player.player_id== 'player_1':
                self.opt_acts_ag.append(1)
            if player.player_id == 'player_0':
                self.opt_acts_op.append(1)
        else:
            if player.player_id == 'player_1':
                self.opt_acts_ag.append(0)
            if player.player_id == 'player_0':
                self.opt_acts_op.append(0)
```

13.9 Code: Human opponent input

```
def get_action(self, player_obs):
    if self.policy == 'human':
            lhm_env = self.env.env.env.env.env
            state  = lhm_env.get_state(0)
            print(_print_state(state))
            try:
                action = int(input('>> You choose action (integer): '))
            except ValueError:
                action = int(input('>> You choose action (integer): '))
            action = action
        return action

class human_play():
    def __init__(self,obs_type, n_eval):
        self.obs_type = obs_type
        self.n_eval_episodes = n_eval
    def set_up_env(self):
        root = '/Users/rhyscooper/Desktop/MSc Project/Pages/models/1002_6.zip'
        Eval_env = texas_holdem.env(self.obs_type, render_mode = "rgb_array")
        Eval_env.AGENT.policy = 'PPO'
        Eval_env.AGENT.model = PPO('MultiInputPolicy', Eval_env, **hyperparams)
        Eval_env.AGENT.model.set_parameters(load_path_or_dict= root)
        Eval_env.OPPONENT.policy = 'human'
        self.Eval_env = Monitor(Eval_env)
        self.env = Eval_env

    def play(self):
        mean_reward,episode_rewards, episode_lengths= evaluate_policy(self.env.AGENT.
    model, self.Eval_env, n_eval_episodes = self.n_eval_episodes, verbose = True,
    return_episode_rewards= False)
```

13.10 Primary hyperparameter optimisation

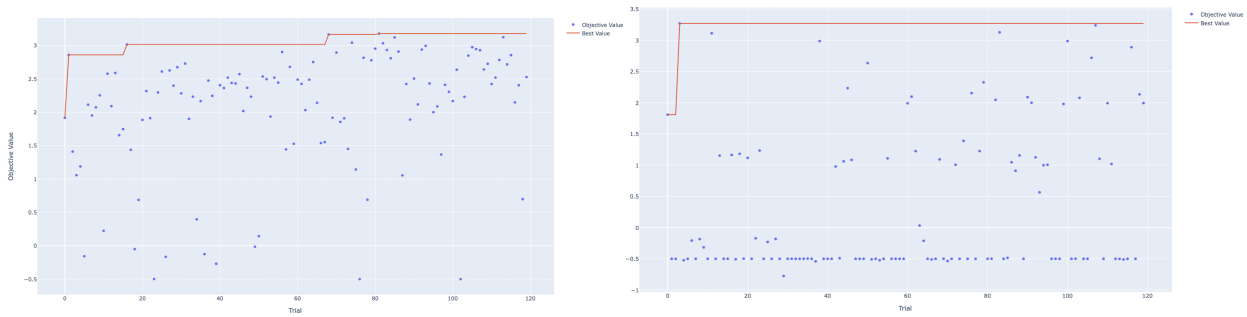


Figure 21: PPO (right) and A2C (left) hyperparameter tuning trial values

13.11 Mean reward of AIP self-play PPO vs random opponent

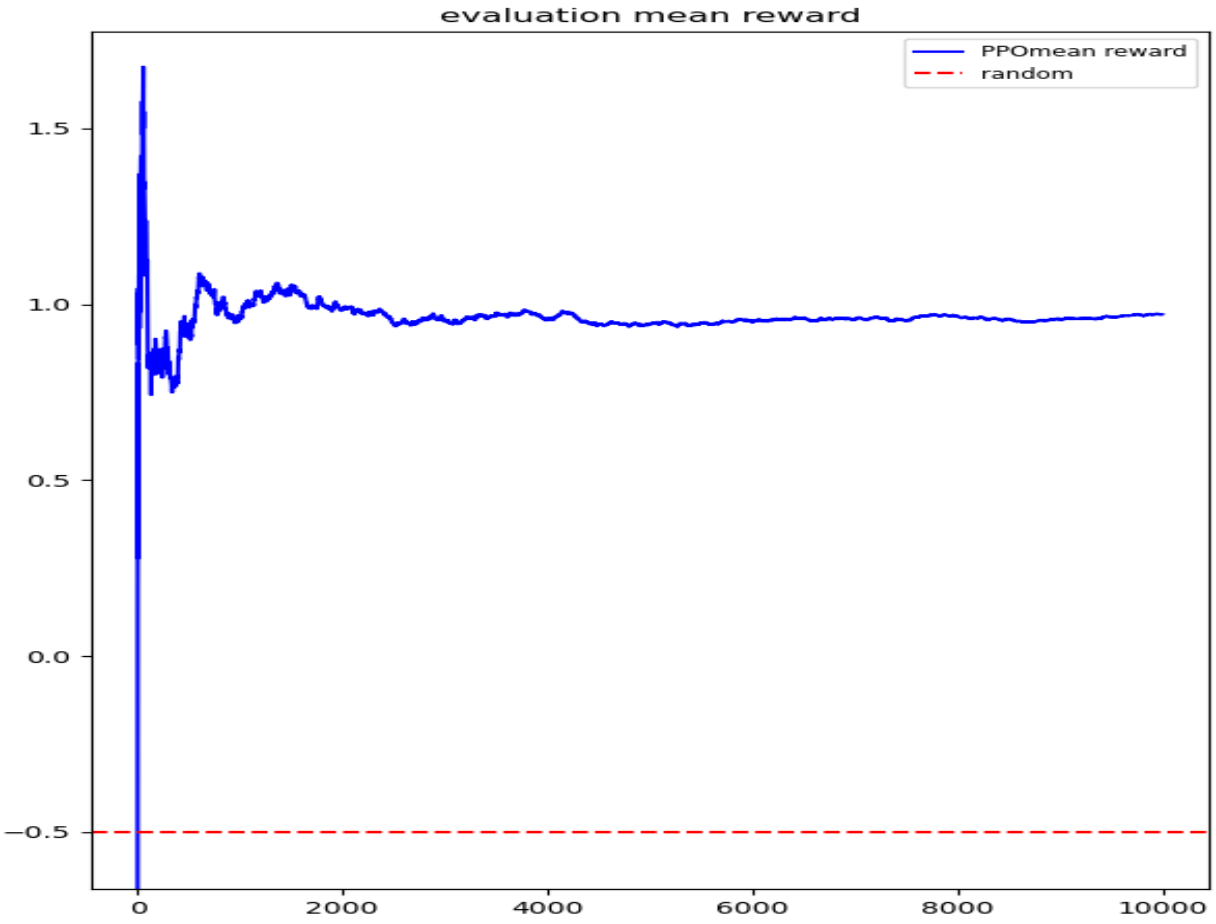


Figure 22: Mean reward of AIP self-play PPO vs random opponent for 10000 steps

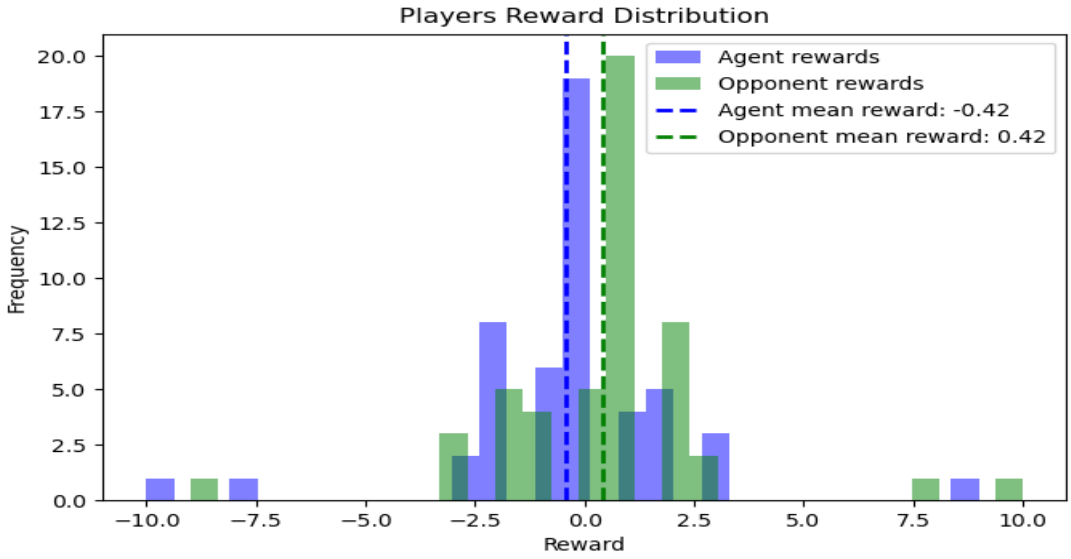


Figure 23: Reward distributions of AIP self-play PPO vs human player

13.12 Installation guide and instructions

Required packages:

1. plotly
2. stablebaselines3 extra
3. optuna
4. treys (poker hand evaluator) https://github.com/ihendley/treys

NOTES:

1. don't install rlcard as this is already included with required modifications
2. made using VScode, suggested to run in this IDE to assist with compatibility
3. file paths to save figures and trained models will likely have to be replaced with a filepath to a folder on your own system.

Instructions: The 'project_main' file collects all the required python files used to fulfill the project.The sections do not appear in sequential order, but in the order of that chosen in the dissertation.The first part relate to the project design and offer quick access to the corresponding files. please open in debugging mode and change JSON setting to "justMyCode": false. this will allow easy access to the files. The second part relate to the project implementation. ie, how the code was used to generate the results and findings used in the project. Finally, the third part include demonstrations:the exact same code as in the implementation and results section but with the variables changed to smaller values to demonstrate the code runs successfully.

when running for the first time. the 'patchgym' function of _patch_env will raise an exception saying the environment is not recognised as an open AI gymnasium environment. This is caused by the custom wrapper modifications making it not being recognised, despite being fully compatible. You can either delete this file on your device and replace it with the modified patch_gym.py file included in this folder. Alternatively you can easily modify the _patch_env function by hashing out all the code and just place 'return env'.

UNIVERSITY OF
LIVERPOOL

Reinforcement Learning for Heads-up Limit Hold'em

Project specification and design

Rhys Cooper, 201680340

Supervisor: John Fearnley

Second Marker: Martin Gairing

1 Project Description

This project is about applying reinforcement learning algorithms to a two player poker game called Heads up Texas limit hold'em (HULH). The essence of reinforcement learning is that the agent will learn some degree of strategy through the process of observing the game, taking an action and then receiving a reward. By playing tens of thousands of games, the learning algorithm of the agent will form a policy: what action to take given a certain state. In the context of this game, a state will be the players hand of cards and the chips it is betting with.

The other player of the game will not be a human but a computer agent with its own strategy, one which will change through out the course of the project. Terminology wise, the player that is using the reinforcement algorithm will be referred to to as the 'agent' where as the other player will be the 'opponent'.

The aim of the agent is to reach a Nash equilibrium strategy. Nash equilibrium is a concept from game theory heavily used in economics and decision theory to describe a strategy such that the agent can not improve its expected payoff by deviating. This is most commonly demonstrated through the prisoners dilemma game, however it can be more simply observed through the game of 'matching pennies'. Player A and B secretly choose heads or tails and then simultaneously reveal them. If the coins are the same, player A wins and if they are different, player B wins. Payoff is therefore maximised if a player chooses each side of the coin 50% of the time: a strategy that is a (mixed) Nash equilibrium. This also illustrates a zero sum game since the payoff for one player is the direct opposite of the other, a concept shared by the Texas limit hold'em game of this project.

The premise of finding Nash Equilibrium (NE) in simple games has potential applications in real world past finding a profitable poker strategy, particularly within the fields of defence, finance and air traffic control.

As such, this project has value in expanding the knowledge associated with reinforcement learning to find Nash equilibrium. Specifically, this project seeks to determine the ability of a single agent reinforcement learning algorithm to find the Nash equilibrium strategy for HULH.

2 Aims and Objectives

1. Create a reinforcement learning environment of the HULH game.
    - Build a custom wrapper that modifies the two player game into a single agent environment.
    - Modify the functionality of the resulting custom environment to gymnasium syntax to allow the implementation of prebuilt algorithms.
2. Deploy the PPO reinforcement learning algorithm to HULH.
    - Train algorithm using self play.
    - Tune hyperparameters
3. Evaluate the performance of the trained PPO agent
    - Use key evaluation techniques from similar studies.
    - Detect to what extent the agent has learnt a strategy for the game using loss curves.
    - Compare performance against a range of opponents.
4. Compare the performance in the game of PPO to Advantage Actor Critic algorithm (A2C).
    - Implement and train A2C algorithm.
    - Tune hyperparameters.
5. Find the Nash equilibrium strategy of the game through self play.

- repeat policy transfer to opponent until no significant change in the difference between the score of the agent and the opponent.

3 Key Literature and Background Reading

3.1 Poker domains

The nature of Poker as a game allows the testing of many single and multi-agent reinforcement learning algorithms and as such, quickly became a challenge domain (Lockhart et al, 2020). It is important to note that there are four different versions of the Texas Hold'em game used by studies within the literature. The most simplified version - Kuhn hold'em - has no community cards and only one round of betting. This simpler game is commonly used because NE strategies can be easily identified since a closed form, parameterized solution exists (Heinreich and Silver, 2015).

Another commonly used version is Leduc poker: Kuhn poker but with additional betting rounds and two final community cards. The next most developed version is the one used in this project, is the two player version of Limit Texas Hold'em typically called Heads up Limit Hold'em (HULH). This has research value since it is the most simplified poker game still played by real professional players.

The limit condition of HULH means that the game has fixed increments of betting and a limit to the number of raises an agent can make. This limit condition simplifies the agent's action space and makes it more compatible with RL methods. Furthermore, since HULH is an extensive form zero sum game - the payoff of one agent is the direct opposite to the other - this allows for a possible NE to exist. Additionally, HULH provides further simplification to the RL environment since the game always terminates once a player folds or loses. Since an agent does not know the hand of the other player, the environment is characterized as being an imperfect information game (IIG) (Bowling et al., 2015). As a result, a non-deterministic, mixed solution is what an agent aims to learn, one of which is classified as being an optimal poker strategy if a NE strategy is reached.

3.2 Counterfactual Regret Minimisation Algorithms

Counterfactual regret minimisation (CFR) are a family of algorithms that are used by the most famous no limit Hold'em programs: Libratus (Brown and Sandholm, 2017) and Deep stack (Moravcik et al., 2017). They operate through a pair of regret minimizing algorithms that undergo self play and are able to find a NE strategy, an ability arising from the suitability of CFR's to IIG's (He et al., 2022). The aforementioned CFR based programs successfully applied to heads up no-limit hold'em (HUNL) are tabular methods that demand extensive amounts of memory. This led to the development of Deep CFR; model free methods that replace the tabular storage with neural networks and are similarly able to converge to ϵ-nash equilibrium strategies. Since the success of Deep-CFRs for no-limit hold'em has already been established, these types of algorithms are removed as possible learning algorithms for this project.

3.3 Neural Fictitious Self Play

The second most successful learning algorithm applied to HULH is Neural Fictitious Self Play (NFSP). The core premise of fictitious self play is the average strategy of the opponent is modeled and the agent calculates a best response, the latter of which is approximated using neural networks. NFSP and similar learning algorithms are successful in finding NE strategies in HULH (Heinrich and Silver, 2016). Similarly to CFR's, this type of algorithm is therefore excluded as a possible algorithm for this project.

3.4 Policy Gradient Methods

Policy gradient methods operate by directly updating the parameters of an agent's policy using gradient ascent on the estimated expected return, as opposed to estimating a direct value function. The candidate policy gradient methods for this project include Advantage Actor Critic (A2C) and Proximal Policy Optimisation (PPO).

Many studies have applied policy gradient methods to Poker. Deepholdem (Wang et al., 2022) successfully deployed deep CFR alongside PPO to NLTH. This considered, a research area appears in applying PPO, without deep CFR, to HULH. Expanding on this notion, a similar study by Lazaridis et al (2022) use PPO trained under self play on NLTH. They discover that the PPO based agent has a higher mean reward than both a random strategy opponent and a Monte Carlo search based 'hand evaluator, yet a negative mean reward when played against a static opponent who always checks or raised. The authors suggest instability to be the cause of this, to which the unstable learning curve of this PPO from Lazardis et al. (2022) is shown below.

Figure 24: Unstable PPO Learning, extracted from Lazardis et al.(2022)

In terms of explaining this relatively poor performance, the main limitation of policy gradient methods is that they face difficulty in the non-markovian and non-stationary setting of a multi-agent environment. Consequently, in the context of imperfect information games there are no convergence guarantees in finding NE strategy through self play (Fu et al., 2022). With this considered, the novel research question appears to see if the relatively poor performance exhibited by the PPO algorithm on the HUNLH (Lazardis et al., (2022)) also occurs for HULH.

In contrast to the finding of Lazaridis et al. (2022), Charlesworth (2018) applied an PPO trained through self play to a four player IIG called 'Big 2' and found the PPO was able to achieve better than human performance. Past the differences in the games, a possible explanation for this contrasting finding is that Charlesworth (2018) used parallel environments to create large batches in order to reduce the amount of variance. Fortunately the Pettingzoo API supports the use of parallel environments and this functionality will be implemented when deploying the PPO.

A similar study uses a self trained PPO for a four player imperfect information card game called Guandan

(Pan et al, 2022). The overall PPO neural network structure is identical to that of Charlesworth (2002), other than the latters use of RELU activation function rather than TANH. Naturally this provides possible hyperparameter options for this project. Additionally, it has been noted that little hyperparameter tuning was required to observe strong performance by the PPO model, and allows for action masking (Charlesworth, 2020; Pan et al., 2023).

Yang et al. (2023), applying PPO to the 'ticket to ride' game with a state space larger than chess, found that a PPO that undergoes self play was the highest performing agent against all the other RL opponents. This lends support to both the self play training schedule and choice of PPO as the learning algorithm for this project.

In terms of Advantage Actor Critic (A2C) learning algorithm of Mnih et al. (2016), this algorithm was applied by Srinivasan et al. (2018) to the domains of Leduc and Kuhn poker. Naturally another research area appears in testing the application of A2C to HULH and as such, this is chosen as an algorithm to compare the performance of the PPO against.

3.5 DQN

Deep Q Networks (DQN) are a common framework of reinforcement learning algorithms due to which their relative simplicity and easy incorporation of action masking make them an attractive learning algorithm to choose. Despite this, the standard DQN algorithm applied to HULH is unable to find a NE strategy (Heinrich and Silver, 2016). This is due to the limited stability of the DQN algorithm where the non-stationary nature of a two player game means the MDP property that conventional single agent RL algorithm exploit no longer holds. Considering the developments made to DQN's since, a possible research area arises of deploying an improved version of DQN - such as rainbow DQN - to investigate if there exists improved stability such that it can discover a NE strategy. Despite this, there is a notable absence of studies deploying DQN, and its variations, to imperfect information zero sum games. Therefore a DQN variant will not be chosen as the learning algorithm for this project.

3.6 Multi Agent Reinforcement Algorithms

In terms of the use of multi agent reinforcement algorithms (MARL), this can at the most basic level involve both agents using single agent RL methods. However, as mentioned by Zhong et al. (2020), these RL algorithms applied in the MARL environment will likely face difficulty in finding NE, as shown for DQN (Heinrich and Silver, 2016). MARL algorithms instead attempt to account for the non-stationary and non-markovian properties of the multi-agent environment. Within the MARL context, Deep CFR and NFSP have all been able to find NE for HULH. There remains a selection of possible MARL algorithms that can be applied to HULH, namely Policy Space Responce Oracles (PSRO) and Exploitability descent (ED). ED was successfully applied to Leduc and Kuhn Poker (Lockhart et al., 2019), yet there remains the research area of applying it to the more advanced HULH game. Due to constraints in time and expertise, MARL aglorithms will not be implemented in this project.

4 Development and Implementation Summary

4.1 Game Specification

Game objective: The objective of HULH is for a player to win all the available chips by having a better hand than the opponent or by forcing the opponent to fold.

Gameplay:

1. Each player receiving two private cards.
2. The first betting round starts with the player to the left of the dealer.
3. Each player chooses to call, raise, or fold.

4. Three public community cards are dealt.
5. Second betting round.
6. Fourth community card is dealt.
7. Third betting round.
8. Final community card is dealt.
9. Final betting round.
10. The player with the best hand wins the pot. The pot is split if it is a tie.

Hand Rankings: The hand rankings follow the same as in standard poker, with the ascending ranking being: Royal Flush, Straight Flush, Four of a Kind, Full House, Flush, Straight, Three of a Kind, Two Pair, One Pair, High Card.

Betting: The limit component of HULH prescribes the fixed structure for betting. Here, every round has a fixed small bet and a big bet to which the former is used in the first two betting rounds (pre-flop and flop) whilst the latter is used in the last two rounds (turn and river). For each betting round, a player can call (match the other players bet), raise (increase the bet), fold (give up) or or check (skip placing the bet).

4.1.1 Observation Space

The observation space that describes the state of the agent consists of three components: The agents cards, the community cards, and the chips raised by the players.

The community and agent cards are represented as a vector of 52 boolean intergers to which each set of 12 elements of the vector represents a suit, and the position of a 1 corresponds to a card. Each of the 4 betting rounds is a set of 5 boolean intergers which represent the number of chips raised, ranging from 0 to 4. The cards and chip vectors are joined to produce a overall vector of 72 boolean intergers shown in table 1. With perfect information, there are 3.16×10^{17} possible states the game can reach, whilst with imperfect information accounted for, there are "3.19×10^{14} decision points where a player is required to make a decision" (Bowling et al., 2015).

Index	Observation Component
0 - 12	Spades: 0: A, 1: 2, ..., 12: K
13 - 25	Hearts: 13: A, 14: 2, ..., 25: K
26 - 38	Diamonds: 26: A, 27: 2, ..., 38: K
39 - 51	Clubs: 39: A, 40: 2, ..., 51: K
52 - 56	Chips raised in Round 1: 52: 0, 53: 1, ..., 56: 4
57 - 61	Chips raised in Round 2: 57: 0, 58: 1, ..., 61: 4
62 - 66	Chips raised in Round 3: 62: 0, 63: 1, ..., 66: 4
67 - 71	Chips raised in Round 4: 67: 0, 68: 1, ..., 71: 4

Table 6: Composition of state observation

4.1.2 Action Space

The actions the agent can take during each betting round described previously are represented as a boolean interger vector consisting of 4 elements as shown in Table 2.

4.1.3 Reward Function

The reward function is a simple win-lose reward calculated as half of the chips cumulatively raised during the betting round. Since HULH is zero sum, these are equal and shown in Table 3.

Index	Action
0	Call
1	Raise
2	Fold
3	Check

Table 7: Discrete action space index

Win	Lose
+ raised chips / 2	- raised chips / 2

Table 8: Reward composition

4.2 Technical Setup

Python is chosen as the implementation language due to its primacy within the libraries that will be used and VScode is the chosen IDE to develop the project due to its integration with common frameworks like PyTorch and TensorFlow and for its debugging capabilities. The agent will be trained on a desktop computer with 6 cores, 16gb of RAM, GTX Titan GPU and running windows 10.

The HULH game is implemented using the Pettingzoo library; a multiagent version of the popular opneAI Gymnasium environment. It uses the *RLcard* library to deploy the core HULH game and implants it into an Agent Environment Cycle (AEC). The AEC steps the environment for each agent, returning agent specific observations, states, rewards and legal next actions. This is shown in Figure 2. For rendering and metadata storage purposes, this is passed into the 'raw environment' class and as such, the overall inheritance structure of this is shown in the UML diagram of Figure 4.

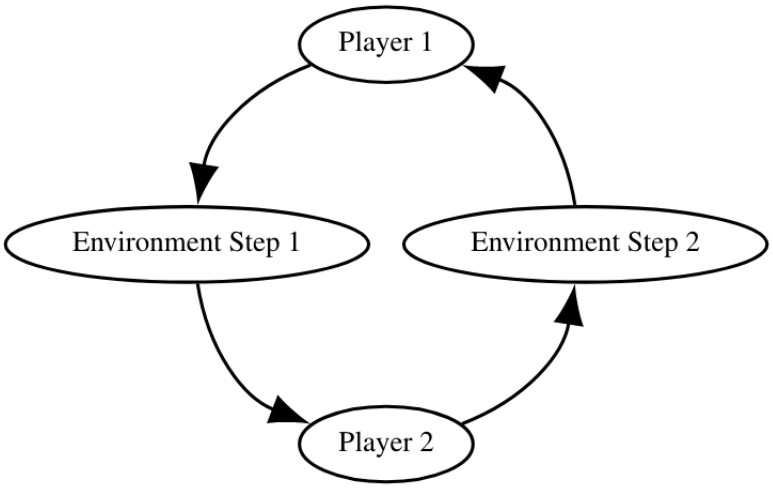


Figure 25: Agent Environment Cycle, extracted from Terry et al(2021)

This raw environment is passed through a series of functions that take as input an environment and outputs it with a wrapper applied. Each consecutive function adds a wrapper that provides functionality to impose the conditions of the game. The *terminate illegal* terminates the game if the agent chooses an illegal action, *out of bounds* ensures the action is in the action space and *order enforcing* forces the correct sequencing of environment stepping and resetting. This is shown in the sequence diagram of Figure 3.

4.2.1 Environment

Most notably, when the environment is stepped by the agent, the wrapper will then in-situ pass an observation to the opponent, take its chosen action, step the environment and then return the appropriate observation

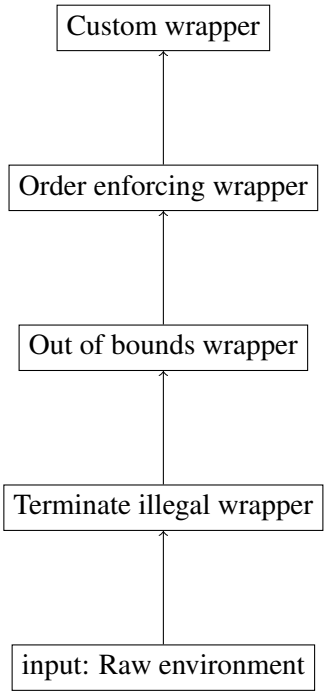


Figure 26: Sequence Diagram of environment wrappers

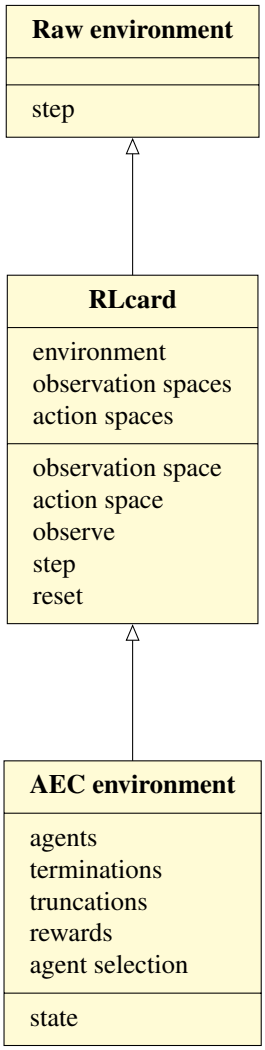


Figure 27: UML diagram showing inheritance

and reward back to the learning agent. This is equivalent to the automatic completion of the remainder of the AEC cycle shown in Figure 2, after the environment is stepped with the learning agents action.

4.2.2 Single agent learning algorithm

The key literature and background reading section informed the choice of Proximal Policy Optimisation (PPO) as the single agent learning algorithm that has not been applied to HULH yet, and that shows promise in finding a NE strategy.

The algorithm will be imported from the Stable baseline 3 (SB3) library since this was successfully used by Pan et al., (2020) and also due to the fact SB3 algorithms have proven compatibility with Pettingzoo environments (Terry et al., 2021).

The underlying neural network structure for implementing PPO consists of two main components: a policy network and a value network, sometimes called actor and critic networks (Charlesworth, 2018). These are shown in Figure 5 as the two neural networks third in from the left. The two outputs of this displayed structure is a softmax distribution of available actions, in addition to a value between -1 and 1 representing the calculated 'advantage' of the action.

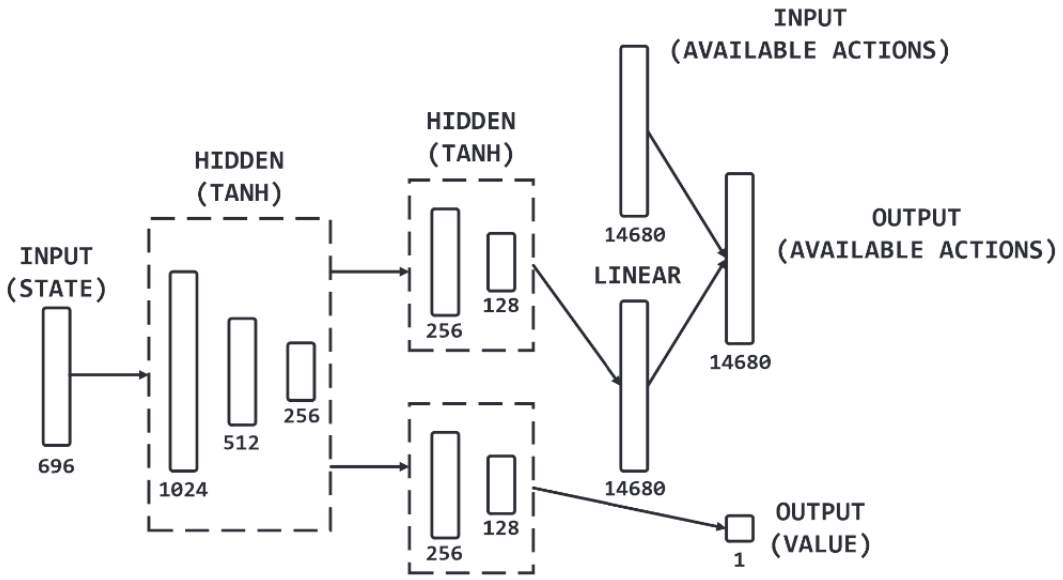


Figure 28: PPO Network Structure, extracted from Pan et al.(2022)

4.2.3 PPO Training Schedule

The PPO algorithm is trained by collecting data from the environment and updating the policy parameters shown in the below steps.

1. **Data Collection:** Trajectories of the agent executing the current policy are collected, recording the agents observed states, actions, rewards, and other relevant information. These trajectories are stored as mini batches.
2. **Surrogate Objective Function:** PPO employs a surrogate objective function that measures the quality of the current policy is measured using the surrogate objective function, usually a clipped variation of the policy's objective function.
3. **Policy Update:** The collected trajectories are then used to compute the surrogate objective to which the policy parameters are adjusted to according to gradient ascent.
4. **Value Function Update:** The value network is typically trained using a regression loss between the predicted values and the actual observed returns.

The training schedule is self play, a process used by Charlesworth (2018), Lazardis et al., (2022) and Pan et al. (2023). The opponent will start with a random policy, and after the agent has learnt a stable policy, this policy will then be transferred to the opponent. The agent will then undergo the next era of learning in which it is effectively playing against itself. This cycle of policy transfer and learning will repeat until the agent reaches a strategy that can be described as a Nash equilibrium. This is visually shown in Figure 6 below and it is important to note that this is different to the NFSP described in the key literature and background reading section.

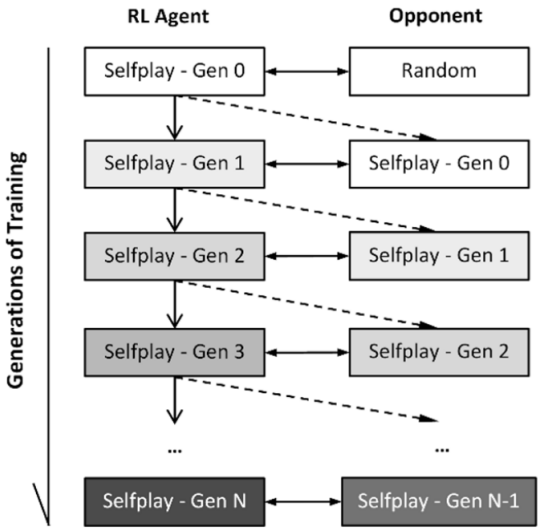


Figure 29: Self play training scheme, extracted from Yang et al.(2023)

4.2.4 PPO Hyperparameters

The table below lists the proposed hyperparameters to be tuned, along with the values used in the most relevant papers (Missing entry in Table 4 are due to not being specified by the authors).

	Author		
Hyperparameters	Charlesworth (2018)	Pan et al. (2022)	Yang et al. (2023)
Training time steps	150, 000, 000	50000	
Generation upgrade frequency / Mini batch size	960	128	
N training updates	156, 250		
activation function	RELU	TANH	
optimiser	SGD	ADAM	
Policy network n layers	1	2	2
Policy network length of layer	256	[256, 128]	[64, 64]
Value network n layers	1	2	2
Value network length of layer	256	[256, 128]	[64, 64]
Shared initial network n layers	1	3	
Shared initial network layers size	412	[1024, 512, 256]	
Learning rate	0.00025	0.00025	0.00040
Discount factor	0.995		0.99
Clip range		0.2	0.2
value function coefficient (α)	0.5	0.5	
entropy coefficient (β)	0.02	0.01	

Table 9: Hyperparameters

5 Data Sources

No public domain datasets will be accessed during the course of this project and as such, no permission is required from any person, company or institution.

6 Testing and Evaluation

Evaluating the performance of the self play trained PPO on the HULH game is done in line with the methods used in similar studies. Evaluation of the extent to which the agent has learnt some form of poker strategy will be done through plotting the loss against training epochs, as done in Pan et al.(2022). This will be further supplemented with a plot of win rate against training time steps as done in Yang et al. (2023).

In terms of evaluating the actual game performance of the agent, this will be measured by comparing the mean reward against a variety of opponents. This will include an opponent with a random policy, and one deploying a A2C algorithm policy.

To evaluate the extent to which a NE strategy has been found through, a plot will be created which shows how the mean reward of agent changes throughout the self play procedure. The expected result should be that throughout generations of training, the main reward between agent with Gen N agent and Gen n-1 agent in Figure 6 should reduce. If a Nash equilibrium strategy is discoverable by the PPO algorithm, there will be a point in the generations of self play training where the difference between in mean reward between Gen N agent and Gen n-1 agent is insignificant. A similar graph was produced by Charlesworth (2018) showing how the mean reward of the final trained agent compares to opponents using policies from earlier times in training. This is shown in figure 8 below.

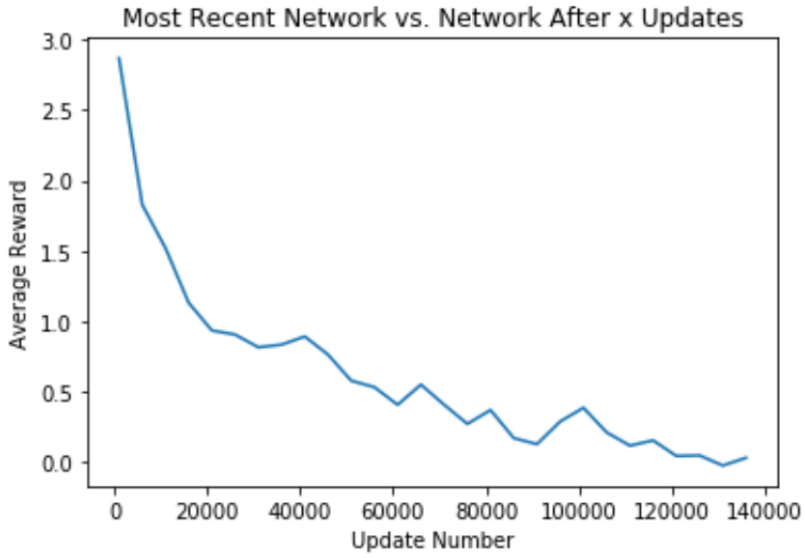


Figure 30: Final trained agent vs policies from previous generation, extracted from Charlesworth .(2018)

7 Ethical Considerations

Considering that this project involves no human participation, this project does not require any ethical approval. Additionally, there is no foreseeable use of artificially generated data such as probabilistic distributions within this project. However if it arises, the data generation process will be explicitly disclosed.

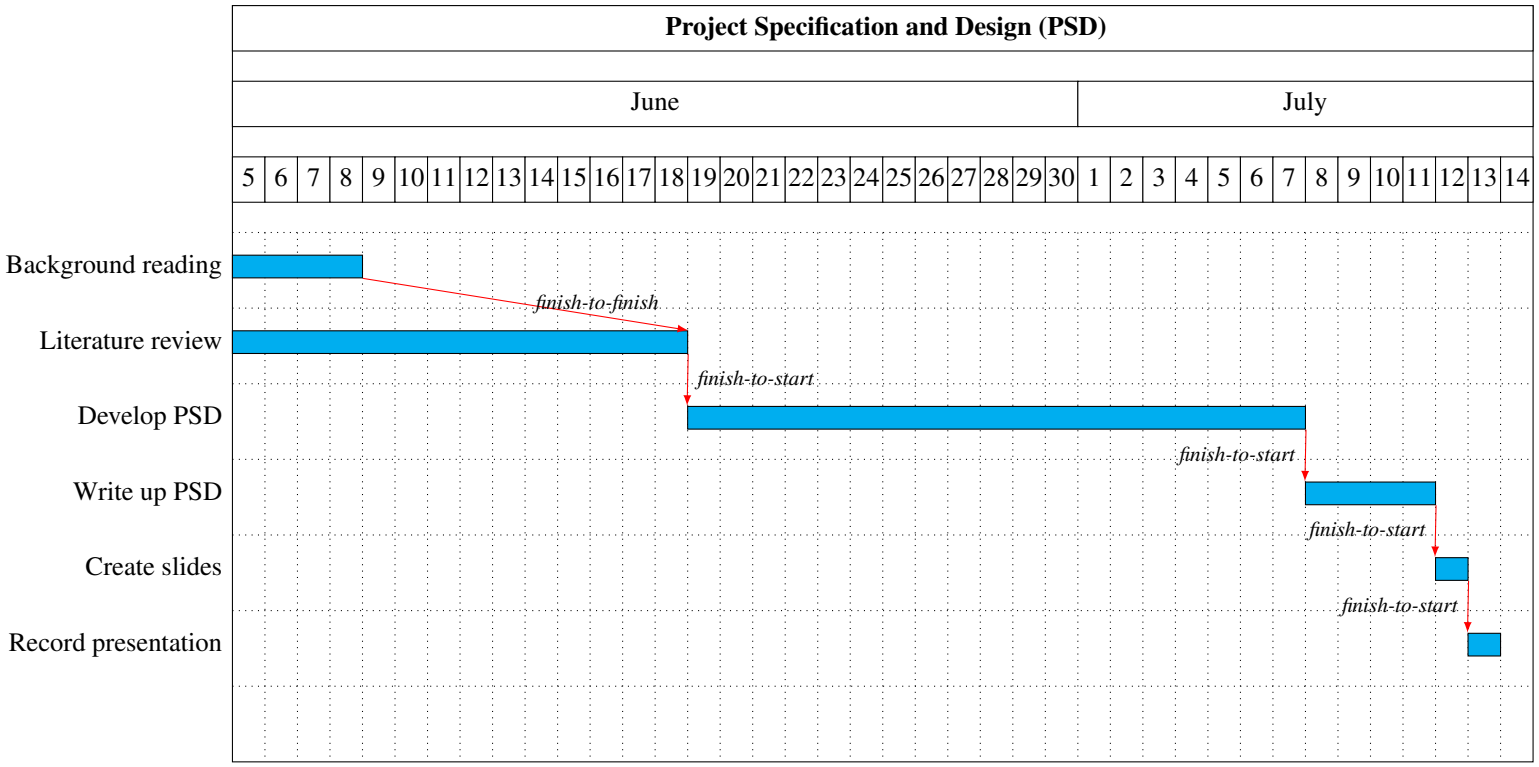
8 Project Plan

The below lists describe the detailed workflow to be implemented in order to realise this project, and is supplemented with a Gantt chart for each key stage of the overall project showing the expected duration of tasks and the dependencies.

The part of the project that is expected to be the most time consuming is the training and hyperparameter tuning of the PPO algorithm. As such, the Gantt chart for software implementation and testing reflects the longer duration of these tasks.

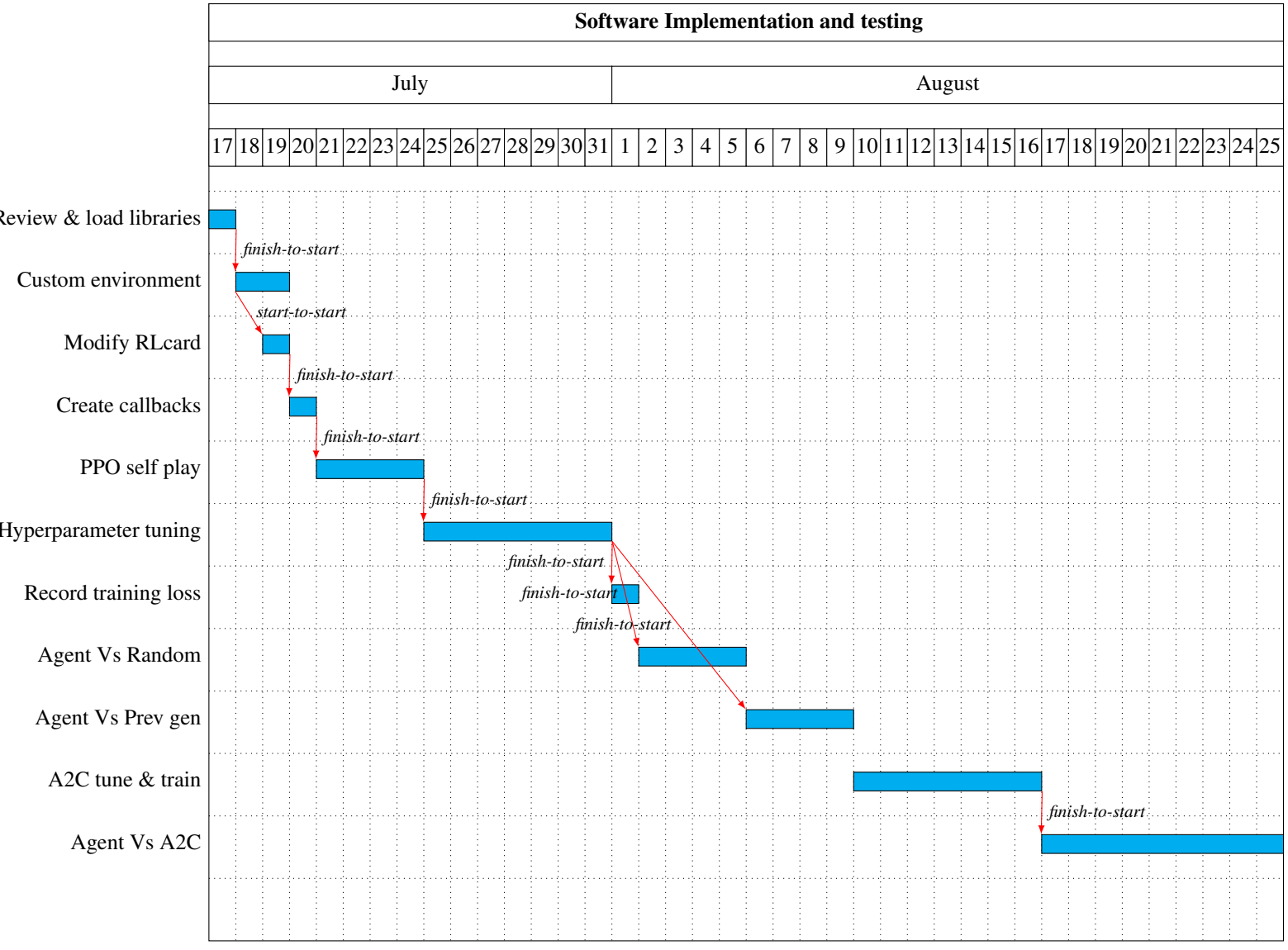
8.1 Project Specification and Design (PSD)

1. Background reading and literature review
2. Development of PSD
3. Writing up of PSD
4. Creation of slides
5. Recording of presentation



8.2 Software Implementation and testing

1. Set up VS code IDE and install Gymnasium, Pettingzoo, Stable baseline3, OpenSpiel and Tianshou libraries
2. Create custom environment wrapper for single agent learning algorithm compatibility
3. Modify RL card base 'player' class to specify which players are opponent and agent, and add functionality to allow internalisation of the relevant policy.
4. Test custom environment conforms to gymnasium format using environment checker function
5. Create custom call back class and evaluation environment to report on the learning progress of the agent
6. Implement the PPO self play training process outlined in section 4.2.3, saving the policy of previous generations.
7. Tune hyperparameters specified in Table 2.
8. Record loss curve for the final agent
9. Record mean reward of agent against random opponent and create graph
10. Record mean reward of agent against opponent using previous policies
11. Implement and briefly hyperparameter tune the A2C algorithm
12. Record mean reward of the final PPO agent against opponent trained using A2C algorithm and report on the comparison in performance.



8.3 Dissertation

First Draft

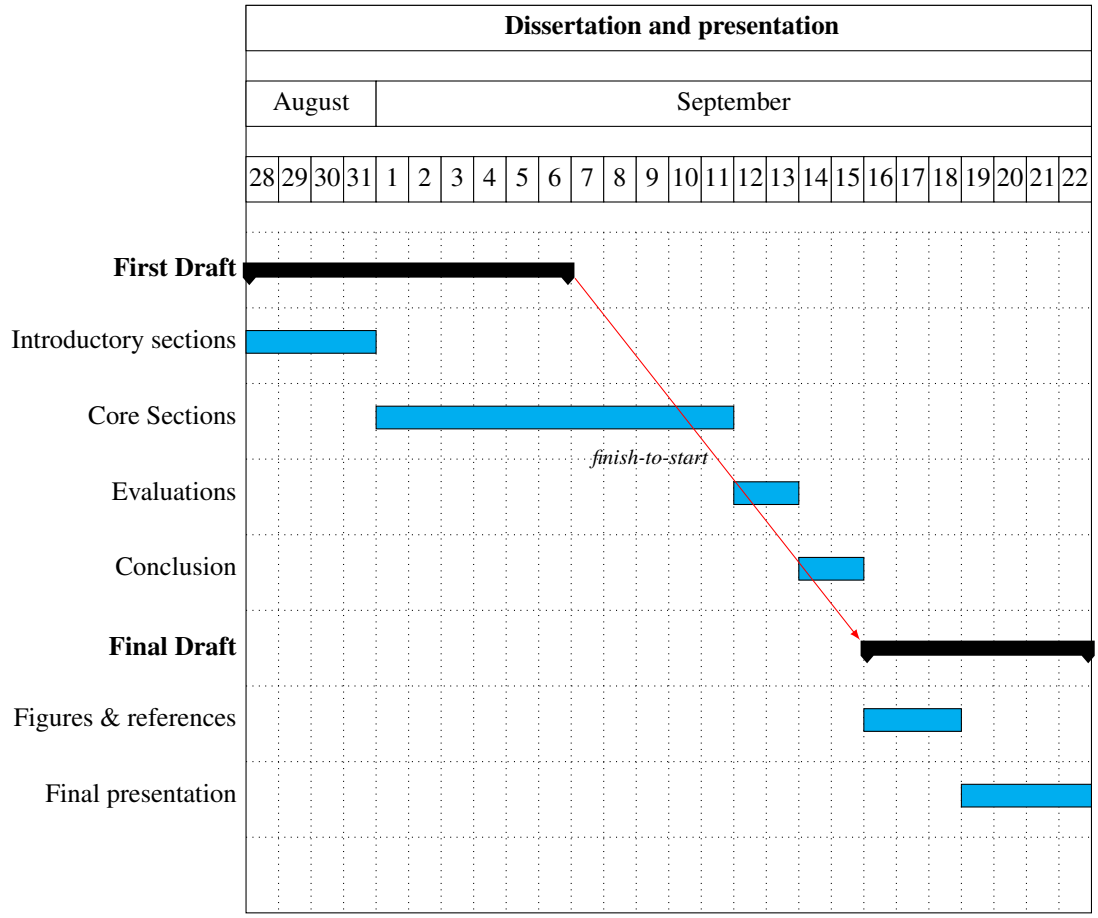
1. Introductory sections including title page, abstract, declaration, acknowledgements, table of contents.
2. Core sections including introduction, aims and objectives, background reading and literature review, design and implementation.
3. Evaluation, learning point, professional issues and conclusion.

Second Draft

1. Shortening main body and formatting the latex document.
2. Add appendix and bibliography.

Final Draft

1. Check figures and references.
2. Final presentation.



9 Risks and Contingency Plans

The below table outlines the risks associated with this project, including their likelihood, potential impact and contingent measures taken to mitigate the risks.

Table 10: Project Risks and Contingencies

Risks	**Contingencies**	**Likelihood**	**Impact**
Hardware failure	Regular backups to cloud storage	Low	Delays to workflow
Software failure	Spyder chosen as alternative IDE. Rlib, Openspiel and Tiansho selected as backup librarys for PPO algorithm provision.	Medium	Delays to workflow and potential changes to software implementation plan
Running out of time	Full project planned using Gantt charts with generous time allocation to tasks to accommodate unforeseen issues. Application of MARL removed from project to increase feasibility.	Medium	Likely to impact the evaluation and testing stage of software implementation
Programming problems	Debugging and compliance with gymnasium format	Medium	Delays to workflow