Department of Computer Science & Engineering
Independent University Bangladesh

Creating an Open-AI Gym like environment for Bangladeshi Game Shologuti using Unity3D and ML-Agents

Samin Bin Karim

Student Id: 1720049

Supervised by

Dr. Amin Ahsan Ali

January 2021

Report submitted in partial fulfilment for the degree of
Bachelor of Science in Computer Science

Department of Computer Science & Engineering
Independent University, Bangladesh

Abstract

Our initial goal for this project had been to implement and benchmark state of the art Reinforcement learning (RL) algorithms for the game Shologuti. However, there was no environment available for Shologuti that could be used to test RL algorithms in. Hence, our objective became to first implement a game environment for Shologuti that can easily be accessible for RL research. To make the environment easily accessible, the environment had to be cross platform and accessible using python programming language, as python is the de facto language favoured by researchers. We also wished to design a RL algorithm that could generalize to various games while consuming easily manageable resources, as the ultimate goal for developing RL algorithms is to eventually apply them to the real world.

To achieve these goals, we decided to create the game environment the world's most popular game engine Unity3D as that allowed us to export our game environment to all major platforms including Windows, Linux and macOS. We then used Unity ML-Agents Toolkit, that allows any Unity game environment or simulation to be turned into a RL environment where we can train intelligent agent. Unity ML-Agents also comes with support for a python Low-Level API which can communicate with the unity environment through ML-Agents external communication module. The python environment can receive observations from the environment and set actions and also receive rewards. Using a RL agent implemented in python we can easily train on Unity3D environment through Unity ML-Agents. As for the design of RL algorithm used, we decided to initially only try to implement a simple Artificial Neural Network that evaluates states, (like TD-Gammon (*Tesauro, Gerald (March 1995).Pdf*, n.d.)) because some of the more advanced algorithms are tricky to implement and taxing on hardware to run.

We managed to implement the environment as intended and also successfully connected our environment to external python scripts. While we did implement a RL agent, we were not able to fully train and test it due time constraint.

We however did manage to train two agents using the state-of-the-art implementations of Proximal Policy Optimization, PPO and Soft-Actor-Critic, SAC, algorithms provided by Unity3D Technologies. The agents were able to perform respectably against hardcoded symbolic Artificial intelligence of various degrees.

Attestation

I understand the nature of plagiarism, and I am aware of the University's policy on this. I certify that this is an original work by me during. However, following internationally accepted academic guideline of using others written work and / or software (in the form of code) in my university project is properly cited if used in any part of this work.

Signature:	Signature:
Name:	Name:

Evaluation Committee

Signature: ……

Name: ……

Supervisor: ……

Signature: ……

Name: ……

Internal Examiner: ……

Signature: ……

Name: ……

External Examiner: ……

Signature: ……

Name: ……

Convener: ……

Acknowledgements.

I would like to thank Dr. Amin Ahsan Ali for his enlightened help and guidance and inspiration throughout the project and for suggesting the idea of local Bangladeshi game Shologuti as a viable game for research in Artificial Intelligence. I would also like to thank Raihanul Bashar Fahim for his advice and oversight throughout the research process that was done for this project.

Table of Contents

List of Figures

List of Tables

1 Introduction

Shologuti or Sixteen Soldiers is a popular game in rural regions of Bangladesh and India. It is a board game like checkers with 2 players [Wikipedia]. In villages it may be played by drawing the board (Fig 1.6) on the ground or on paper and using stones or broken pieces of stick as gutis/pawns. Our goal was to implement state of the art Reinforcement learning (RL) algorithms on the game Shologuti. However, there was no environment available for Shologuti that could be used to test RL algorithms for the game. Hence, our objective became to implement a game environment for Shologuti and then develop RL agents to play the game against human users and other RL agents. In this chapter we discuss the history of Shologuti and related games, the project objectives and contributions, and the limitations and outline of the report.

1.1 Background and Context

■ Brief history of the Shologuti game

Shologuti is a part of a family of games like Draughts/Checkers that are all descendants of the middle eastern/Arabian game of Al Quirkat (Alquerque - the Spanish name of the game) figure 1.2. Earliest mention of it is in a book Kitab-al Aghani, the author of which died in 976 AD. There is some speculation that it may have originated in ancient Egypt (an unfinished Alquerque board is seen cut on roofing slabs in a temple at Kurna, which was built around 1400BC. But this may have been just a graffiti). It was brought to Spain by the Arabs, and mentioned in the Book of Games Manuscript which was written between 1251 - 1282 at the command of Alfonso X, King of Leon and Castile. It spread to England (the English Draughts game, first referenced in the 14th century) and France (where the game was called Jeu Force and Le Jeu Plaisant De Dames, first known reference found in the 15th century). These were played on chess boards. Games with similar boards to the original Al Quirkat one, is found and played in Africa as well (;ike Fanorona - Madagascar).

More board games with gameplay similar to Shologuti are found in South-Asia (for example Perali Kotuwa or the War Enclosure from Sri Lanks, figure 1.4). Bagh-bondi, Sher-Bakar, Jaguar and dogs (figure 1.1) are variants also played in Al Quirkat or Shologuti like boards, played all over the world from South America, Middle East, Africa, Russia, South-east Asia (Malaysia and Indonesia).

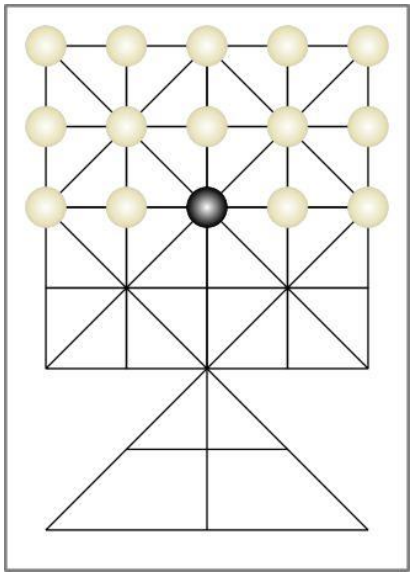

Figure 1.1 Adugo Jaguar and Dogs

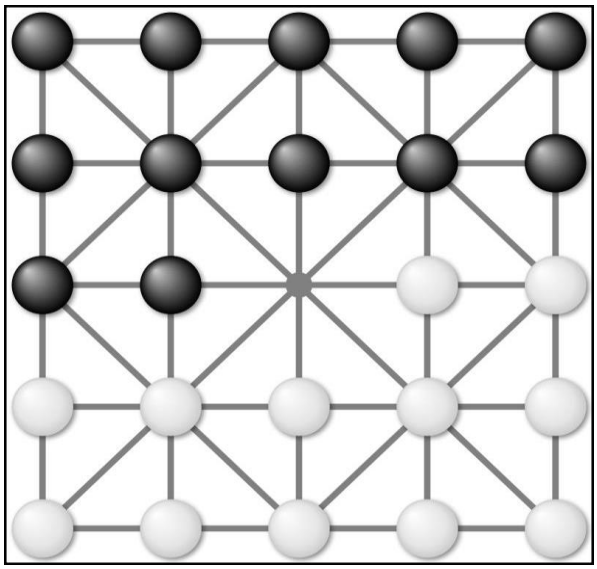

Figure 1.2 Al-Quirkat - Spain

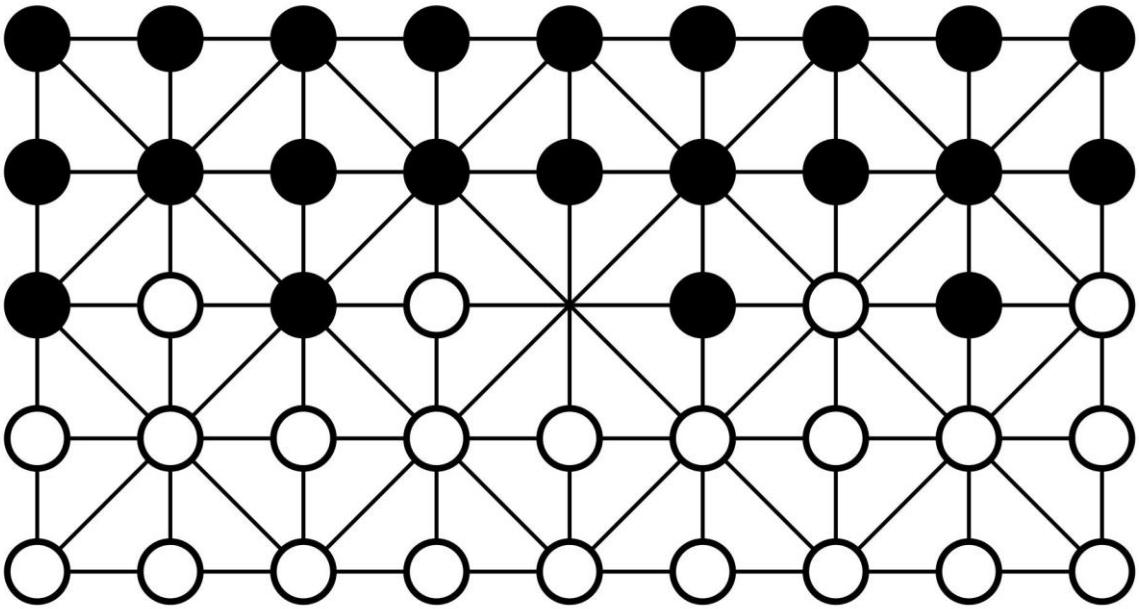

Figure 1.3 Fanorona-Madagascar

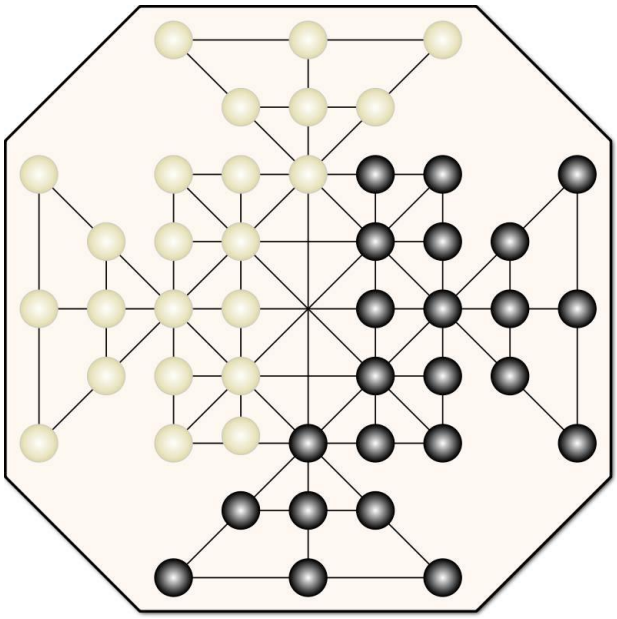

Figure 1.4 War Enclosure - Sri Lanka

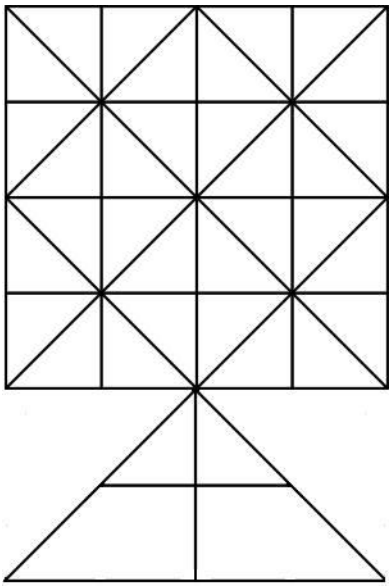

Figure 1.5 Tiger Game - Chile

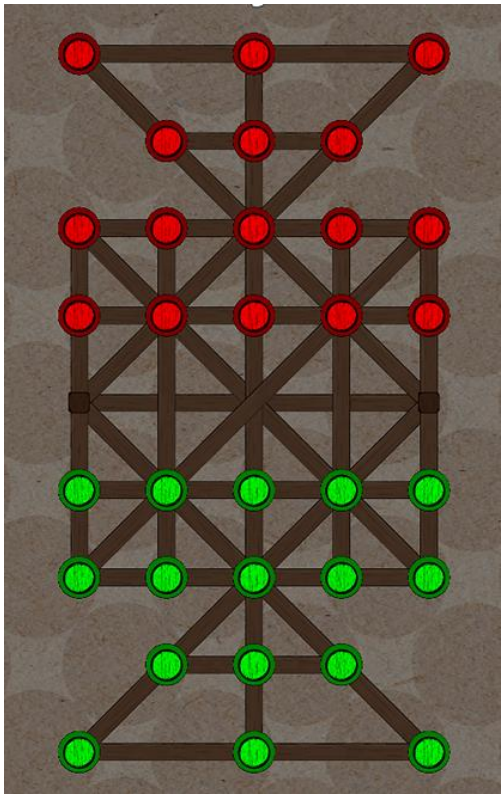

Figure 1.6 Shologuti - Bangladesh

■ Shologuti game rules

A traditional Shologuti board usually has 37 points joined by lines (Fig 1.6). Each player has sixteen pawns or guti. Players make moves in alternate turns. During their respective turns, players can move their guti along a line to a vacant adjacent point on the board or they can leap over to a vacant point behind an enemy guti along the same line. Leaping over an enemy guti results in it being captured. Capturing a guti earns 1 score point to the capturing player. Fig 1.7 and Fig 1.8 shows the board before and after a capture move. The capturing guti can perform more captures within the same turn if there are other such adjacent enemy guti along lines with empty points behind them along the same line. Whichever player captures all sixteen enemy guti first, wins the game. [Wikipedia]

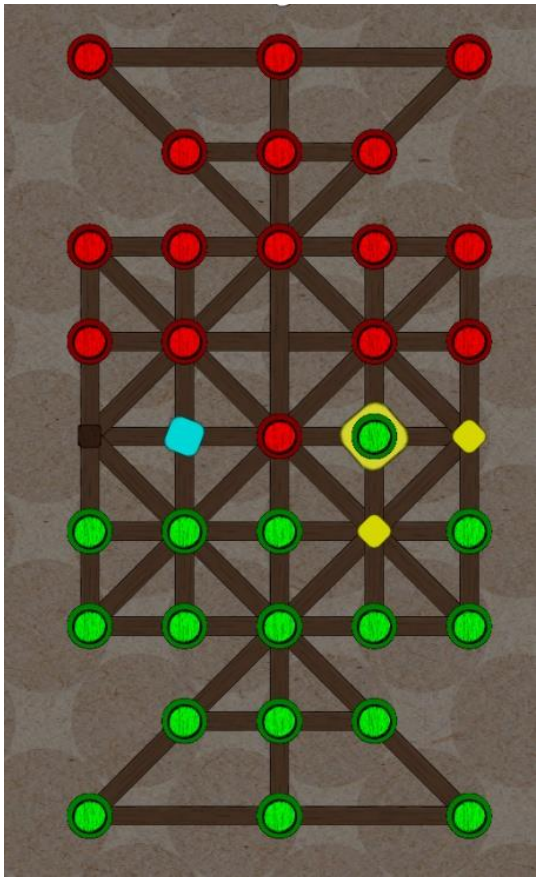

Figure 1.7 Before Capture

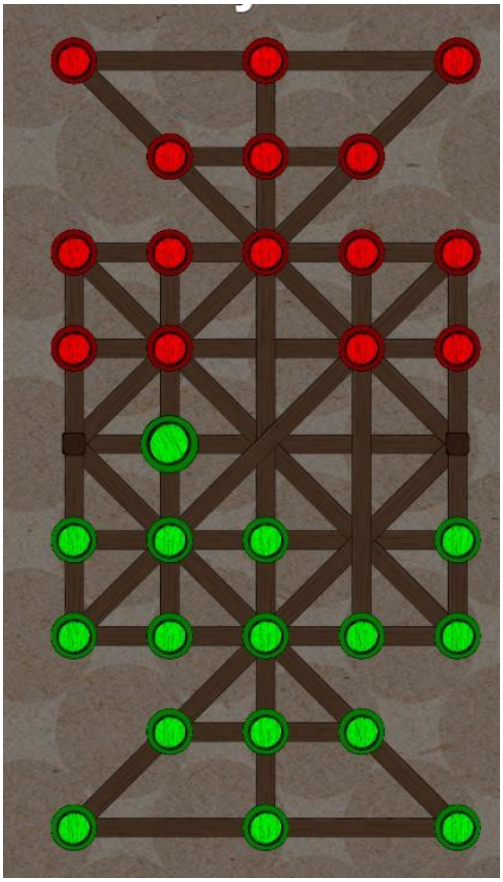

Figure 1.8 After Capture

■ Shologuti board game as an AI research environment

With a board size of 37 and with 2 players starting with 16 identical pawns each and five empty spaces, the state complexity of the game is roughly, 10^{18}, as show in the equation below [**Error! Reference source not found.**],thus m aking it comparable to that of popular game Connect 4 (Baby & Goswami, 2019) which has a state complexity of 10^{13} and English draughts (8 x 8 checkers) which has a state complexity of 10^{18}.

In the following equation,

$$\boldsymbol{x_1} = number\ of\ gutis\ owned\ by\ player\ 1$$

$$\boldsymbol{x_2} = number\ of\ gutis\ owned\ by\ player\ 2$$

$$\boldsymbol{x_3} = number\ of\ empty\ spaces\ on\ the\ board$$

$$\boldsymbol{x_3} = board\_size - x_1 - x_2$$

$$\textbf{board\_size} = 37$$

$$\log_{10}(\textbf{Statespacecomplexity}) = \log_{10} \sum_{x_1=0}^{16} \sum_{x_2=0}^{16} \frac{37!}{x_1!\, x_2!\, x_3!} = 17.58 \qquad 1.1$$

$$\log_{10}(\textbf{State space complexity}) \approx 18$$

The game also poses challenges for an Artificial Intelligence, such as valid move generation and searching and evaluation of optimal long term score maximizing moves. Shologuti has an empirical average **branching complexity** of **14**. (Empirical estimate from 10,000 games played on the final game made in Unity)

Hence, Shologuti may be considered as an interesting game environment that is worth exploring in the realm of Artificial Intelligence.

■ State of the art in Reinforcement Learning

Reinforcement Learning is a branch of Machine Learning that is concerned with learning to make decisions based on observations it receives from its environment, such that the rewards it receives from the environment are maximized. Agents learns to make optimal decisions for a given state of environment by trial and error. Different data structures may be used to map state/observation received from the environment to meaningful action/decisions taken in the environment (*Tesauro, Gerald (March 1995).Pdf,* n.d.). Neural networks are becoming increasingly popular as function approximators for this task as Deep Reinforcement Learning algorithms (Mnih et al., n.d.)have had ground-breaking achievements in recent years in multiple domains like Go game (Hölldobler et al., n.d.), Atari (Mnih et al., n.d.) games and StarCraft 2, AlphaStar (Arulkumaran et al., 2019).

To reduce entry to barrier and facilitate development and benchmarking of new RL algorithms, In 2016 OpenAI Gym (Brockman et al., 2016) was released as a robust platform for researchers to test and benchmark their Reinforcement Learning algorithms and agents in a standardized environment. Open AI Gym combined previously popular benchmark environments Arcade Learning Environment (ALE) (Bellemare et al., 2013) RLLab (Duan et al., 2016), and others into a software package that is easy to use and install. The primary interface for interacting with these environments was created in python, with easy platform independence and ease of understanding and use.

More recently, other OpenAI gym like environment have also been released RLCard(Zha et al., 2020) (multiagent environment for card games), Pettingzoo_Gym for multi-agent Reinforcement Learning environments (*Pettingzoo_gym_for_multi_agent_reinforcement_learning*, n.d.). Deep mind themselves released another library OpenSpiel (Lanctot et al., 2020) that contains environments like Chess and Poker and is adapted more for multi-agent learning.

■ Motivation for the project

Initially, the aim for this project was to benchmark state of the art Reinforcement algorithms on Bangladeshi games like Shologuti. However, despite some implementations of the game Shologuti existing (Nawshin & Saifuzzaman, 2018), their code base was not accessible and they could not be used as environments to test Reinforcement learning (RL) algorithms.

Due to the lack of availability of local games like Shologuti for easy benchmarking, combined with the popularity of Open-AI Gym like environments, the decision was made to create an Open-AI Gym like environment (Brockman et al., 2016) for Shologuti game to enable easy use in benchmarking and testing modern Reinforcement learning agents and techniques.

1.2 Objectives

Based on the previous discussion on the current state of research in RL methods for board games like Shologuti, we specify the following objectives of the project:

1) Design and implement a playable game environment for Shologuti using Unity 3D and C#
2) Design and implement symbolic Artificial Intelligence Minmax/Adversarial Search in the game so that Humans and Reinforcement Learning Agents have an opponent to play against.
3) Develop the environment for multiple platforms including Windows, Web and Linux.
4) Use Reinforcement Learning algorithms, PPO and SAC, implementations provided by Unity to train agents
5) Implement trained agents with different levels of experience to represent different difficulty levels for an agent trained with particular method.
6) Create python wrapper that allows users to communicate and exchange data with the Unity 3D environment, using easy to understand interface in python
7) Test that the python wrapper can successfully and efficiently communicate with the Unity 3D environment
8) Design and Implement a Reinforcement Learning Algorithm written in python to benchmark in the Shologuti Environment
9) Train and Test the Reinforcement learning Algorithm and produce Baselines

1.3 Challenges

Developing the Shologuti environment while meeting the objectives stated in section 1.2 poses some significant challenges. To develop the game environment a good understanding of Unity game engine and fundamental game development concepts is required. Since the project is sufficiently large, to develop a functional, extensible and easy to debug software, a good grasp of the SOLID principles and OOP is necessary. To develop the AI for Shologuti, which is an adversarial zero-sum board game, a solid foundation in game theory and classical AI is also required. Since the aim for the project is to ultimately make a test bench and test state of art RL algorithms in the Shologuti game environment, a thorough understanding of reinforcement learning theory was absolutely compulsory. The most difficult task for the project was understanding reinforcement learning from classical RL to state-of-the-art RL algorithms like SAC and PPO. Mastering the Unity toolkit ML-Agents, that was used to develop the RL agents for the game, was also a huge challenge as ML-Agents is not a widely used technology and also undergoes constant breaking changes.

1.4 Contributions

During the time of writing, no easily accessible Shologuti game environment exists that can be used as a test bench for training and benchmarking reinforcement learning algorithms and agents. Hence, this project may enable future Artificial Intelligence based research on Shologuti. The contributions of this project are as follows:

1) We have created fully playable game environment for a Bangladeshi traditional rural game Shologuti. The game has custom sprites, art and animations to make it more visually interesting
2) The Shologuti game and RL environment can be exported to almost any platform including Linux, Mac, Windows, Android and iOS with proper support for Windows and Web Platforms.
3) Because the environment is built in a C# engine, it will perform much faster than any python-based implementation of a simulator. The simulation speed will translate to faster training times.
4) The game environment has multiple implementations of AI. These are useful for enhancing gameplay and also as baselines for benchmarking or as opponents to train other RL agents against
    a. Agent using symbolic AI algorithm MinMax
    b. Trained Proximal Policy Optimization (PPO) agent (Can be controlled with external python scripts)
    c. Trained Soft-Actor-Critic (SAC) agent (Can be controlled with external python scripts)
    d. Temporal Difference, TD trainable agent (random agent if not controlled by external python scripts)
5) The difficulty of each of the AI can be controlled.
6) Easy to use GUI Settings page allows users to easily change the settings of the game environment.

7) The different AI can play against each other and the user can enable stepping mode to step through each move in the game. It can be useful for learning the game and also for analysing strategies used by the AI.

8) The trained PPO and SAC agents use neural networks that work natively in C# using Baracuda. The NN can be converted to other formats, enabling them to work natively on the Web and other supported platforms as well.

9) All of the RL agents are trainable and controllable using external python scripts by using the Unity ML-Agents low level python API

10) A python wrapper was built on top of the low-level python API that can communicate with, train and control TD agents active in a Shologuti environment. It provides an easier method for benchmarking and training agents in the Shologuti environment

11) PPO and SAC algorithms were benchmarked in the environment using a variety of different training parameters and methodologies. The results of the experiments are included in this report.

12) An effective novel reward system was found for training RL agents that we termed Intermediate Goal States

1.5 Limitations

Unity ML-Agents is a toolkit provided by Unity that is used widely in this project to connect the Unity environment to external trainers and python scripts. Many of the project's limitations revolve around Unity ML-Agents being unstable as breaking changes are made to its codebase with newer updates.

1) Once a version of the game is compiled and exported for a certain environment, more trained networks cannot be loaded into the environment to run natively. New implementations of RL algorithms can only be run through external python scripts.

2) The shape of the interfaces, sensors and actuators, that communicate with external scripts also become limited to the ones provided with the compiled environment.

3) New interfaces can only be added by extending the base C# code.

4) ML-Agents is a fairly new and uncommon technology which makes it difficult to learn and debug as many of the problems encountered may not have readily available solutions.

5) The project is also dependent on a specific version of ML-Agents (version 1.05) as latest version of ML-Agent has breaking changes that make it incompatible with the Shologuti project.

6) Due to dependence on code that maybe deprecated in the foreseeable future, extending this code base to add more functionality to the core environment may prove to be tedious or unsustainable.

7) Upgrading to the newest versions of ML-Agents may require a fair amount of code to be rewritten.

8) Reinforcement Algorithms are resource hungry and time consuming to train.

9) This also makes them quite difficult to debug and tune, as the result of a change in the hyper parameters or some aspect of the simulation environment or training algorithms takes some time to become evident.

10) The environment has no sound; hence a user can only use visual feedback to interact with the system

1.6 Outline of Report

In chapter 2 we provide a literature review. A brief history of RL and summary of the different papers that were read and used for this project are summarized. The summarized papers also include related research done on other local Asian games such as Carrom and Ludo. In chapter 3 and 4 we discuss the design and implementation of Shologuti game environment and the RL agent component. We discuss the training and benchmarking of popular RL algorithms such as PPO and SAC in chapter 5. Other results on performance analysis of the RL agents is also discussed. Chapter 6 concludes the report. We also discuss the limitations and future goals regarding Shologuti game environment and RL agent development as well as for similar games in this chapter.

2 Literature Review

2.1 An Overview of Reinforcement Learning

Reinforcement Learning is a branch of Machine learning where machine learning models are trained to take sequence of actions that lead to certain goal or goals.

Reinforcement learning agents must learn to take optimal actions in simple (for example, a simple 2D maze) or complex (for example Chess Game, Real World, etc) environments using only the parts they can observe. The parts the agent can observe are known as States. The sequence of actions the agent needs to take given its observations is known as the Policy. A state for a chess game agent may be the positions of the pieces on the board, while the policy is which move to make given the state.

To tell the agent if it is getting closer to its goal or goals, Reinforcement learning algorithms have a Reward Function. For example, a reward function for chess can give the Chess AI a positive reward for winning and a negative reward for losing. By optimizing its actions for all possible observable states, the agent's objective is to maximize its rewards and hence learn how to optimally solve a problem, which in case of chess is winning the game, without being explicitly taught how to do so.

Reinforcement Learning methods separate into two distinguishable paths, namely value-based learning methods and policy gradient methods. Value based methods learn the value of observable states and uses these values to decide which action to take to maximize the value / total reward it will receive in the future. The observed state can also be paired with the action taken in that state and an optimal value can instead be learned for the state action pair, known as a Q-Value while still being considered a value-based learning algorithm.

Policy Gradient Methods on the other had try to learn which action to take in any given observable state directly instead of learning the value function and then using that to plan. However, many more recent algorithms use a combination of Value-based and policy-gradient methods. Examples include Alpha Go Lee and its successor algorithms. [citation needed Alpha Go Lee].

2.2 Tree-Searching Algorithms for Game Solving

As games become more and more complex, they pose increasingly interesting challenges for Artificial Intelligence that are going beyond the realm of toy problems. Board games have been used as test beds Artificial Intelligence for a long time with the Deep Blue Chess machine (Feng-Hsiung Hsu, 1999) making headlines in 1997 by defeating the then World Champion in Chess Garry Kasparov. Deep Blue Chess machine had leveraged its available computation capacity to search the simulated Game Tree (Tree of all possible future board states) far into the future using MinMax Alpha-Beta Pruning Search. Pre-existing knowledge and expert tuning were used to create heuristic functions to evaluate board states when search could not be done to the end of the game. While Deep Blue had a profound impact on AI research, most of its expertise was chess specific. The Deep Blue Chess Machine was an entire machine dedicated to playing chess.

Besides that, MinMax algorithm and its derivates, are exponentially more computationally heavy during runtime when game states further into the future have to be simulated and searched. This is due to the fact that MinMax has to simulate all possible actions the current player takes and all possible actions the opposing player takes in response, resulting in many more future states, from each of which this recursive cycle will repeat for every level in the tree. Thus, the game tree tends to explode out to exponential complexity. Cutting off the simulation and search at shallow depths and using heuristics could result in sub-optimal performance as the AI does not have enough information to correctly weight its actions. The performance is also dependent on how good the heuristic function itself is in general.

MinMax was also not very effective in games where the game tree had high branching factors like the game of Go. To play games like Go to a more competent degree Monte Carlo Tree Searching (MCTS) algorithm proved more useful. In October 2015, Deepmind's Alpha Go became the first AI to beat a professional human Go player without handicaps on a full-sized 19x19 board, using MCTS combined with RL.

MCTS uses an expected outcome model instead of heuristic evaluations. Instead of simulating and searching from scratch every time, MCTS stores the probability of victory from a given state. During training runs, ff a state has never been encountered before, MCTS will add it to its growing tree and then record whether the AI won or lost by the end of the game into this new node. When used for playing against other players, MCTS exploits the knowledge it gained during its training runs to pick actions leading to future states that resulted in higher victories during its training iterations.

2.3 Reinforcement Learning Algorithms

■ Reinforcement Learning based Algorithms for Game Solving

Reinforcement learning gained wide audience in game solving with the release of TD-Gammon (*Tesauro, Gerald (March 1995)*, n.d.) a backgammon playing program that used a multi-layered perceptron (MLP) to approximate the value of the game state. The MLP was trained using Temporal Difference Learning, which is a rudimentary reinforcement learning algorithm. The program was trained via self-play (the AI playing against itself to learn) and

managed to achieve better performance than human players in the game backgammon. The algorithm worked by using the MLP as a heuristic function for a shallow (simulating only one or two levels into the future) MinMax tree search. While the training was quite computationally heavy and time consuming, running the trained AI was a far cheaper operation than brute force searching. Even though this was a major breakthrough, using similar structure and strategy in other games like chess did not reproduce the same level of success.

Deep Minds DQN, Deep Q-Network Architecture (Mnih et al., n.d.) which was based off of Q-Learning made a major breakthrough in 2013 by learning to play real-time Atari 2600 games from only direct visual inputs. The architecture uses a Convolutional Neural Network that takes visual input from the game screen and outputs into another Deep Neural network which eventually outputs the value of different actions in that given state. In essence, it takes visual input from the game screen and decides what action to take like a human. The CNN compresses the observable state space as it groups images that are similar together due its inherent feature extraction. This compression of similar states makes it viable to learn from visual input directly as otherwise, a single pixel change in the image would cause to be registered as a new state, thus increasing state space complexity enormously. DQN agents also trains using the sparse rewards from the environment (like score on the screen) like a human would as well. To increase exploration and sample efficiency, DQN stores gathered experience in a fixed size replay buffer and randomly batches experiences from it to train the network. The DQN learned to play 7 Atari 2600 games from the Arcade Learning Environment (ALE)(Bellemare et al., 2013). The agent was trained with self-play and had defeated all other previous solutions in benchmarks and even managed to defeat human experts in a few games.

In 2016, Deepmind's Alpha Go Lee (Hölldobler et al., n.d.), managed to defeat Mr. Lee Sedol, widely considered the best Go player, 4 to 1. Go is a board game that has a state complexity and branching factor even higher than that of chess. This makes the game very hard to solve using traditional Game Tree Searching methods. Prior to Alpha Go, this was considered impossible for state-of-the-art AI. Alpha Go Lee achieved this using a combination of Supervised-Learning to pre-train its Agent with prior domain knowledge and Reinforcement Learning to further improve upon its compressed domain knowledge. Alpha Go employed Monte Carlo Tree Search (MCTS) to explore the game tree and train its policy and value network. The tree searching module allows Alpha Go to explore the problem domain thoroughly an

A later iteration of Alpha Go, Alpha Go Zero (Silver et al., 2017), managed to defeat Alpha Go Lee, its predecessor, while only being trained using self-play. Thus, Alpha Go Zero removed the Supervised learning component. Alpha Go Zero achieved super human performance while learning Tabula Rasa (i.e., with no prior domain knowledge) using only self-play experience to train its neural networks.

A further iteration of Alpha Go Zero, known as MuZero (Schrittwieser et al., 2020) is a version of Alpha Go Zero that has been adapted for more general use instead of being fitted to the Go Game specifically. Mu Zero uses all the same basic architecture and principles but can be used for a wide range of tasks ranging from Chess to Robotics. Unlike the previous versions, MuZero does not expect a perfect simulator to be available on which it can perform

MCTS (Monte Carlo Tree Search). Instead, it uses a separate network to learn what the future state of the environment will be after taking an action

Besides the algorithms ranging from Alpha Go to Mu Zero, Deep is also behind the Policy Gradient and Value based reinforcement algorithm known as Proximal Policy Optimization, PPO (Schulman, Wolski, et al., 2017). PPO is a very popular Reinforcement learning algorithm for robotics and other continuous control problems. PPO is covered in more detail in the next section where we discuss the Reinforcement Learning algorithms that were actually used for the Shologuti Project.

More recently, in 2019, Deepmind's Open AI 5, a PPO (Berner et al., n.d.) based AI built to play Dota 2, managed to defeat World Champions in the game. Dota 2 is a very popular real time team based multiplayer game with 5 players on each side. The players must coordinate with their team to eventually destroy the opponents base in order to win the game. Additionally, each player must build their character by earning coin and buying equipment as well as levelling up from gathering experience. The map for each team is shrouded to what they and their team can see together. This allows the opponents to set up ambushes and perform many other complex tactics. The game Dota 2 poses a difficult problem for AI systems as each match can last over an hour, the action space is complex and continuous and the observable space is limited and imperfect. Dota 2 therefore poses a challenge that is comparable to challenges AI face in real life tasks.

The deep mind team adapted the PPO algorithm (Schulman, Wolski, et al., 2017) to learn from huge batches of data collected and used to training over large scale distributed systems. After 10 months of training, Open AI five achieved super human performance while only training against other scripted AI and itself.

■ Reinforcement Learning Algorithms used in the Project

Two state of art Reinforcement Learning Algorithms that are currently widely used are Soft Actor Critic (Haarnoja et al., 2019)(SAC) and Proximal Policy Optimization PPO (Schulman, Wolski, et al., 2017). Although these algorithms are more popular in use for robotics tasks, they can also be successfully applied to discrete environments such as board games (Baby & Goswami, 2019)

Proximal Policy Optimization, PPO is primarily an online policy gradient learning method. While PPO also learns a Value function, the Value function is only used to train the Policy network. Online Policy gradient learning methods explore using the same Policy that they update. Hence, they are typically very sensitive to update step size for the policy and can converge sub optimally and then be unable to recover. Due to this, limiting or optimizing the step size of the update is very important in policy gradient methods. To solve this issue, some algorithms implement a Trust region around the old policy to which the policy network can be updated confidently as the value network has real experience within the region.

Trust Region Policy Optimization, TRPO controls its step size by adding the KL Divergence constraint to its action policy update function (Schulman, Levine, et al., 2017). While updating the policy, TRPO computes the average KL-Divergence between the new policy and the old policy and ensures value of the KL-divergence is below a certain

threshold, hence ensuring the distance between the two policy is within a certain threshold and thereby keeping the size of the update within certain limits.

PPO uses this idea of constraint from TRPO, but instead of computing KL-Divergence to constraint the step size, PPO introduces a novel Clipped Objective Function that clips the update function values when they cross the trust region.

Soft-Actor-Critic, SAC (Haarnoja et al., 2019)is an offline Policy gradient method that uses both Q-Values and State-Values to estimate its Objective function. SAC also adds an entropy term to the Objective function in order to encourage exploration. Entropy is a measure of randomness. For policy network that outputs action probabilities given a state, the entropy will increase when the action probabilities are more evenly distributed (hence the agent will take more random actions / explore more) and decreases when the probability concentrates onto few actions (that is when the model has become more deterministic). The effect the entropy term has on value of the objective function is determined by the entropy coefficient, which slowly decays as the agent trains until a designated number of episodes. Soft-Actor-Critic also uses and experience buffer from which it samples experience tuples to train with. It is therefore not updating the same policy that it is using to train and does not need further systems to control step size as its entropy bonus system reduces risk of the Agent converging too fast and sub optimally.

2.4 Unity: A General Platform for Developing Intelligent Agents

Unity: A General Platform for Intelligent Agents (Juliani et al., 2020), paper discusses the state of the art of benchmarking environments available for reinforcement learning. It also argues that Modern game engines are uniquely suitable as general platforms for reinforcement learning as they enable the creation of learning platforms that are rich in visual, physical and task complexity. Using Unity3d game engine and Unity ML-Agents Toolkit, the paper surveys the research that is made possible by a flexible, tuneable and configurable environment, details of which can be adjusted easily if needed.

Unity and ML-Agents form a general platform that can be used to build any type of environment, whether it is a turn-based game or an accurate physics simulator or a robot in a realistic environment.

Unity and ML-Agents come with a comprehensive set of tools that lets users attach sensors to their agents they want to control in the game environment. The agents can be equipped with neural networks and trained using the provided python implementations Algorithms such as PPO (Schulman, Wolski, et al., 2017), SAC (Haarnoja et al., 2019) or imitation learning. The hyperparameters for these implemented algorithms can also be tuned.

If needed, Unity and ML-Agents also allows Unit3D environments to be accessed using python scripts in order to control agents with custom Machine Learning Algorithms implemented in Python.

2.5 Artificial Intelligence Research in Board Games from Bangladesh

■ Carrom

Multiplayer Carrom Board Game with an Ai Opponent, is a multiplayer Carrom made in Virtual Reality that has an Artificial Intelligence that uses a Decision Tree deducted from a database to take decisions. It is simple, easy and non-taxing on hardware and is overall a very interesting way to incorporate Machine Learning into the field of games. The game was made in Unity3D as Unity games can easily be ported to Virtual Reality. The environment was not unfortunately created to act as a test bench for 3rd party Artificial Intelligence (Nanaware et al., 2020). However, a very useful test bench for carom, implemented in python does exist https://github.com/samiranrl/Carrom_rl and it is used as a test bench for CS 747 - Foundations of Intelligent and Learning Agents, taught by Prof. Shivaram Kalyanakrishnan at IIT Bombay.

■ Ludo

Ludo is a game that has had more attention than games like Carrom and Shologuti. Complexity Analysis and Playing Strategies for Ludo and its Variant Race Games, (Alvi & Ahmed, 2011) analyses the state space complexity of the game ludo and suggests and analyses 4 different strategies that artificial intelligence may take to solve the game.

Artificial Intelligence: Game Techniques Ludo - A Case Study (Chhabra & Tomar, 2015) discusses implementing Q-Learning in a Ludo environment. TD(λ) and Q-Learning Based Ludo Players(Alhajry et al., 2012) implements a TD(λ) based player and a Q-learning based player, which after training, outperformed an expert player in the game.

Applied Reinforcement Learning with Use of Genetic Algorithms(Derakhshan, n.d.) uses Q-Learning methods using Artificial Neural Network to build ludo playing agents. Author introduces an idea called a parenting recipe. Parenting recipe indicates the reward policy for an agent when making moves. The author uses genetic algorithm to create new generations of Parenting recipes which then change how the agent is rewarded and hence how it learns to act. The author provides proof that his system of optimizing the reward policy (or Parenting Recipe) using genetic algorithm has the desired effect by comparing agents that trained with a fixed reward policy with an agent whose reward policy was optimized using genetic algorithm.

■ Shologuti

The game Shologuti has not been researched extensively in the field of Artificial Intelligence. The only prior work relating specifically to this game we found is "Role of Shologuti in Artificial Intelligence" (Nawshin & Saifuzzaman, 2018). The paper discusses the implementation of MinMax Adversarial Search Artificial Intelligence in the game Shologuti using .Net. It also argues why Shologuti is an interesting environment for Artificial Intelligence as it requires the Move Generation, Evaluation and Searching. Unfortunately, the game environment they created was not accessible and could be used for further research.

3 Design and Implementation of Shologuti game

In this chapter we describe the system overview, design and implementation details of Shologuti as a playable game and discuss the challenges that we faced while implementing the target features for the game.

3.1 Target Game Features

The features and deliverables that we aim to implement for Shologuti as a playable game are the following:

1. Working and fully playable cross platform Shologuti game. Unity game engine can export the game to almost all platforms including Linux, MacOS, Android and iOS. However, we only aim to export and properly test versions for Windows and the Web as subtle problems may arise in compiled versions of the game for every platform which can require any amount time to solve.
2. Give players the choice to select one of the multiple board game AI. We aim to implement MinMax AI along with two others trained RL Agents trained using SAC and PPO algorithms.
3. Adjustable Difficulty of all implemented AI. Both the MinMax AI and the RL Agents will have two different difficulty levels that can be selected.
4. Useful and user-friendly Heads-Up Display (HUD). HUD is an overlay over the main game that displays relevant information to the user like the score of individual players, their win counts, whose turn it is, what difficulty level is set for the enemy AI, who won the current match or if it was a draw, etc.
5. GUI Interface to interact with the game and to control in game options and settings.
6. Custom art assets and animations for varied and more interesting user experience and feedback.
7. Option for local multiplayer game play on the same screen.

3.2 Challenges

Given the game feature mentioned above, there were a number of challenges that were needed to overcome to development the game. The different challenges faced in developing the game and RL agents are described below:

- **Learning game development with Unity game engine**

Shologuti was the first larger scale game project that I had attempted with Unity and hence many concepts involved were still new for me. As the Shologuti project used an alpha version feature of Unity, namely ML-Agents, I had to spend a good amount of time understanding the internal class system of Unity in C# and exactly how objects are loaded into memory and rendered on the screen and in which order the internal scripts are called. Concepts like state machines, game states and animation states also had to be learned but these we relatively easier to grasp.

- **Learning AI Agents for games and Reinforcement Learning**

The basics of game theory is necessary in order to make an AI for a zero-sum adversarial board game. This was relatively the easiest part of the project. Heuristic MinMax search is a simple to learn and implement yet a powerful algorithm in use. The biggest challenge here was making a lightweight simulator for MinMax to use for simulating future states without corrupting the main board states.

A considerable amount of time had to spent to learn and understand the fundamental concepts of RL. Starting from classical RL to having a good grasp on modern state of the art Deep RL algorithms like PPO, SAC and Alpha Zero. It was a difficult and arduous journey for me. Resources for learning RL from scratch are not as available as others and beginning with classical RL is especially complicated by the fact that it requires a good grasp of concepts from Control Theory like Markov Chains and Markov Decision Process, along with a solid understanding of basic statistics and probability.

- **Struggles with implementing RL-Agents in Unity3D**

  - ML-Agent is still a relatively new machine learning toolkit built for the world's most popular game engine Unity3D. Due to this, ML-Agent undergoes breaking changes from one version to the next and also has scarce and lacklustre learning materials and documentation. Hence, some of the finer details of using the ML-Agents toolkit had to be learned by trial and error and by making extensive use of the debugger built into Rider IDE.
  - Two programming languages had to be used while developing this environment, python and C#. Both these languages have drastically different design philosophies and switching between them was a bit cumbersome. In the past we had tried to use TensorFlow Sharp (A 3rd party C# wrapper for TensorFlow) while using an older build of ML-Agents, but support for TensorFlow Sharp has since been dropped and newer versions of ML-Agents had begun using PyTorch instead of TensorFlow.
  - During the initial stage of the project, it was quite confusing and difficult to install ML-Agents as, the code for it had to be fetched from GitHub and built using Command Prompt and had multiple 3rd party

dependencies. However, newer versions of ML-Agents are available to install easily from the package manager inside the Unity3D game engine.

- Unity3D is a very heavy Game Engine and cannot not be run on lower-end hardware. Hence, we were limited by the hardware we could use for the project. We struggled to run Unity3D on most of the computers we owned ourselves.

■ **Learning and applying SOLID principles**

Due to the increasing size of the code base, concepts from Software Engineering became more and more important for the project. Eventually, I had to learn about SOLID (Single-responsibility; Open-closed principle; Liskov Substitution; Interface Segregation; Dependency Inversion) principles to maintain a readable, reusable and easier to debug piece of software. Partially enforcing SOLID principles to the project reduced tightly coupled classes and made making extensions to the code base much easier. The amount of redundancy in the code and clutter also decreased drastically.

3.3 System Design Overview

■ System Overview Diagram

Figure 3.1 illustrates the basic components of the Shologuti game system. The components and their functions are as below:

1. **GameManager:** The GameManager forms the central hub that controls and enables communication between all other components of the system. All other components are sub-components that help the GameManager manage the environment.
2. **GameState:** This component controls the state the game is in. The game can be in 5 different states, In-Play, Paused, Draw, Player-1 Win and Player-2 Win.
3. **UIManager:** This component is responsible for managing the Graphical User-Interface (GUI) of the system. It controls the animation states of the objects (like the gutis on the board) and buttons that the users interact with. It has a subcomponent that handles the input from the users through the interface. The UIManager can also send information to the Settings component through the GameManager if the user changes the settings using the GUI.
    a. **InputHandler:** Handles input events generated by the user interacting with the game environment.
4. **Settings:** Settings component is responsible for maintaining and updating the various settings that are controllable and affect different aspects of the game environment. The settings can be updated by using information sent by UIManager. Controllable settings include, player type and difficulty on each side, animations on or off, auto restart after a match ends and whether the game has to be manually stepped through after every move.
5. **BoardManager:** It controls and updates the main board used to play the game. It is also responsible for maintaining a simulator that can be used to simulate future states of the game without affecting the main board.
6. **Player:** Each player is responsible for requesting for permission to act from the GameManager and also deciding which action it will take. The Player gets the current main board state from the Board component and sends an action which results in a change to a new main board state. All the communication is done through the GameManager. Three different player types can be spawned. Each type of player has different properties that affect the method in which they choose their actions.
    a. **Human:** Human player types simply waits for the real human controller to input their action through the GUI and UIManager.
    b. **MinMaxAI:** MinMaxAI computes the next action to take using the simulator and the current main board state that it fetches from the BoardManager.

c. **RLAgent:** The RLAgent sends the main game state it receives to an external python script or to a Baracuda NN (a C# readable format for neural networks) and waits to receive an action. The communication is enabled through a Sensor Actuator System that is covered in detail chapter 4. It also has Reward System that is useful for training and is also discussed in chapter 4.

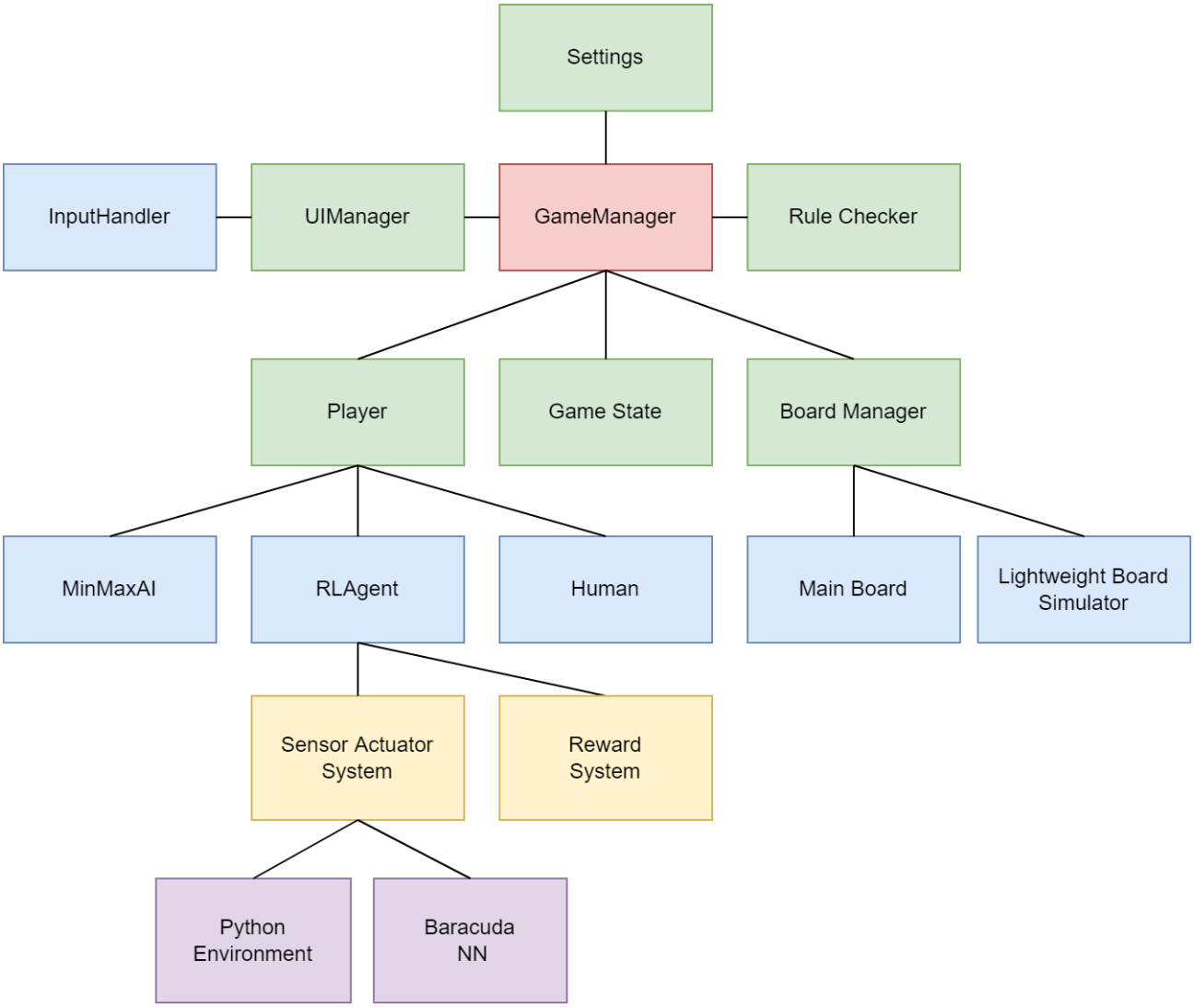

Figure 3.1 System Overview Diagram

■ **Basic Process of Taking Action**

When the request for an action is received, the system checks what type of player is requesting the action. If acting player is a human, then the system waits to receive an action from human and then executes it. If the acting player is an agent using MinMax searching algorithm, then an optimal action is computed using heuristic MinMax algorithm and executed. If the acting player is a RL agent then it sends the current state of the Shologuti board as an observation to a Baracuda NN or an external python script. The agent then waits to receive an action and then executes it.

The following diagram figure 3.2 shows Basic Process of taking an action by the different types of players:

Figure 3.2 Basic Process of an Action

■ Move made by human

When a move is requested from a Player Entity that has type Human, it waits for input from the GUI interface and then executes the move on the Board once the Action is Received. Figure 3.3 shows a Game Tree which shows all possible outcomes of the game along with the unique path that leads to that outcome. Each player takes actions in turn and create a sequence of board states that eventually result in a Red Pawn Player victory or a Green Pawn Player Victory.

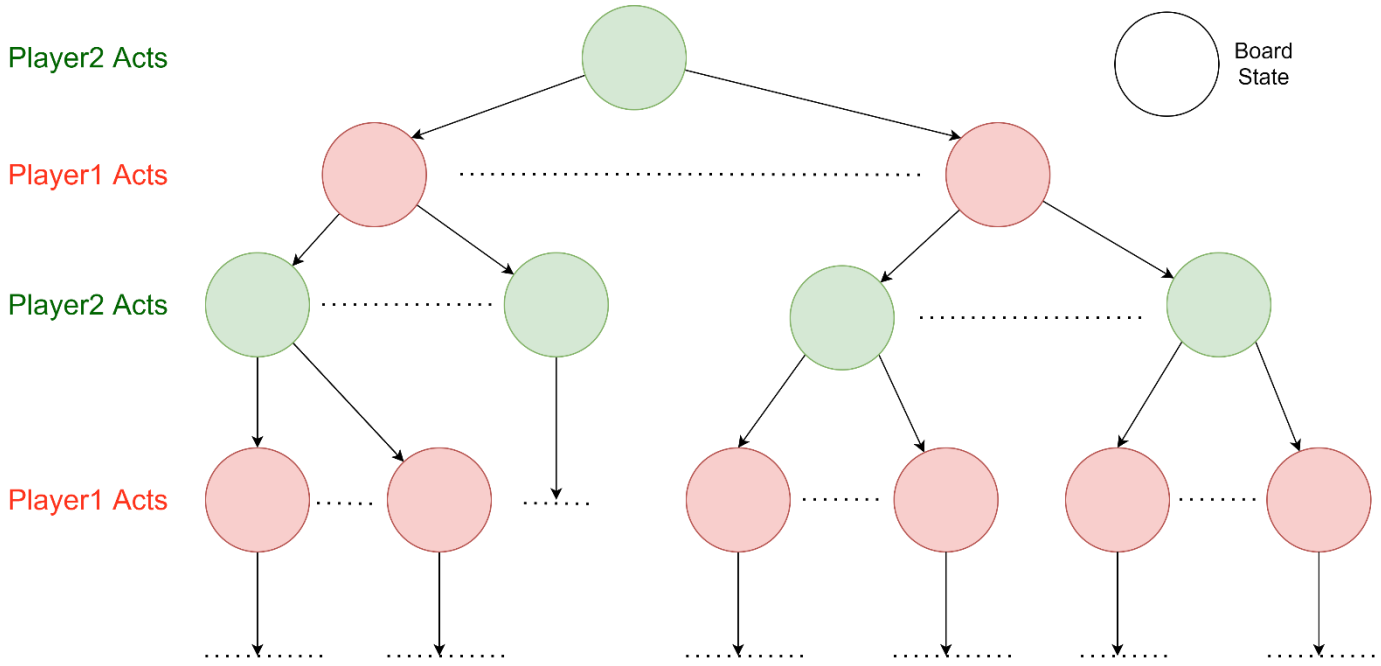

Figure 3.3 Game Tree

■ Move made by artificial intelligence

There are two other types players besides human, as shown in figure 3.1 and 3.2. These are controlled by either an agent using a hard coded symbolic AI algorithm like MinMax or an agent using a trainable AI trained using a reinforcement learning algorithm like SAC or PPO. The player types and the subcategory of agents they encapsulate are listed below:

1. Player type: MinMaxAI
    a. Agent using MinMax algorithm
2. Player type: RLAgent
    a. Agent using Neural Networks trained using SAC RL algorithm
    b. Agent using Neural Networks trained using PPO RL algorithm

Both these types of players and the different agents controlling them are covered in chapter 4.

3.4 Hardware/Software Specifications

This section discusses the hardware and software requirements for developing the Shologuti Environment

- **Software Specifications**

  - Operating System: Windows 8 or later 64-bit versions only, macOS 10.12: Sierra or later 64-bit versions only, Linux (Ubuntu 16.04 or later, CentOS 7 or later) 64-bit versions only
  - Unity 3d Game Engine, versions 2019 to 2020.1
  - Tools: Unity ML-Agents version 1.05 only and exactly
  - Nvidia CUDA version 10.1
  - Python 3.0 or later
  - .Net Framework 2.0 or .Net Framework 4.0.

- **Hardware Specifications**

  - CPU: Minimum Quad Core Processor x86 Architecture
  - GPU: Graphics card with DX10 (shader model 4.0) capabilities.
  - RAM: Minimum 8 Gigabyte

3.5 Tools Used

List of tools used in making this game environment.

- **Unity3D**

Unity3D is a C++ game engine that uses C# as its scripting interface. It allows us to create our game environment on the windows platform and export to multiple platforms including Windows, Linux, MacOS, Web, Android, iOS, etc. This ensures that our environment will be platform independent. It is more versatile than other development environments like Mujoco and does not require a paid license in order to use. It is also easier to learn and use.

- **ML-Agents**

Unity ML-Agents is an open-source machine learning toolkit for the world's most popular game engine Unity3D. It allows games and simulations to serve as environments for training intelligent agents. The toolkit comes with state-of-the art built in machine learning algorithms like PPO (Schulman, Wolski, et al., 2017) and SAC (Haarnoja et al., 2019) to enable users to easily train their agents It provides a simple to use python API that can be used to communicate with the unity3D environment from a python environment in order train test and use custom algorithms.

- **Baracuda library**

Baracuda is a library provided by Unity that can covert neural networks created using PyTorch or Tensorflow to. nn files that can be run and understood by Unity game engine. These. nn files can also be converted to any relevant format using Baracuda including Open Neural Network Exchange, (ONNX) to enable the neural networks to be used in Web, Windows, Linux, MacOS or Android platforms.

- **.Net 2.0**

Unity3D is built on a backbone of Microsoft .Net Framework. Unity allows the use of up to version .Net version 4.0. However, to ensure maximum compatibility we chose 2.0.

- **IDE Rider**

Rider is the integrated Development environment we used to build the C# side of the project. Rider comes with a slew of features like always on Intellisense which smartly suggests Code completions.

- **Visual Studio Code**

Visual Studio Code is a text/code editor made by Microsoft with a lot features and 3rd party plugins which allows it support a wide range of programming languages, with code suggestion, code completion and syntax highlighting. It a robust and versatile tool and is very useful when the code base is not huge. It pairs very well with the python programming language.

- **TensorFlow**

Initially ML-Agents used to provide implementations of reinforcement learning algorithms made on a Tensorflow Sharp backend. Tensorflow Sharp was in turn a C# wrapper over the TensorFlow library. However, as of version 1.05, Unity ML-Agents now uses PyTorch Based implementations

- **PyTorch**

Unity ML-Agents machine learning algorithms are built on top of PyTorch library. PyTorch is an open-source machine learning framework from Facebook's AI Research lab (FAIR).

- **WebGL, Open GL, Direct X, Metal,**

Unity3D being a cross-platform game engine, can build WebGL applications (Web), Direct X applications (Windows), OpenGL applications (Windows/Linux/Android) and Metal applications (iOS, macOS). Which means we do not have to worry about the different graphics API implementations for individual platforms ourselves.

3.6 Implementation of Core Game Systems

We implemented a playable Shologuti Game in C# using Unity3D game engine. The game is playable across all platforms including Web, Windows, Mac and Linux. The game has not been tested or optimised for mobile platforms and may run into unseen issues.

■ Board setup

The Shologuti board is represented in logic as a doubly connected searchable graph. Each node in the graph signifies a position on the board. Each node stores which neighbours it is connected and their address. Each node also tracks which type of guti (red/green or empty) currently occupies its position. The following figure 3.4 illustrates a doubly connected graph representing a Shologuti board. the circles are nodes and the purple lines are edges connecting them to their neighbouring nodes.

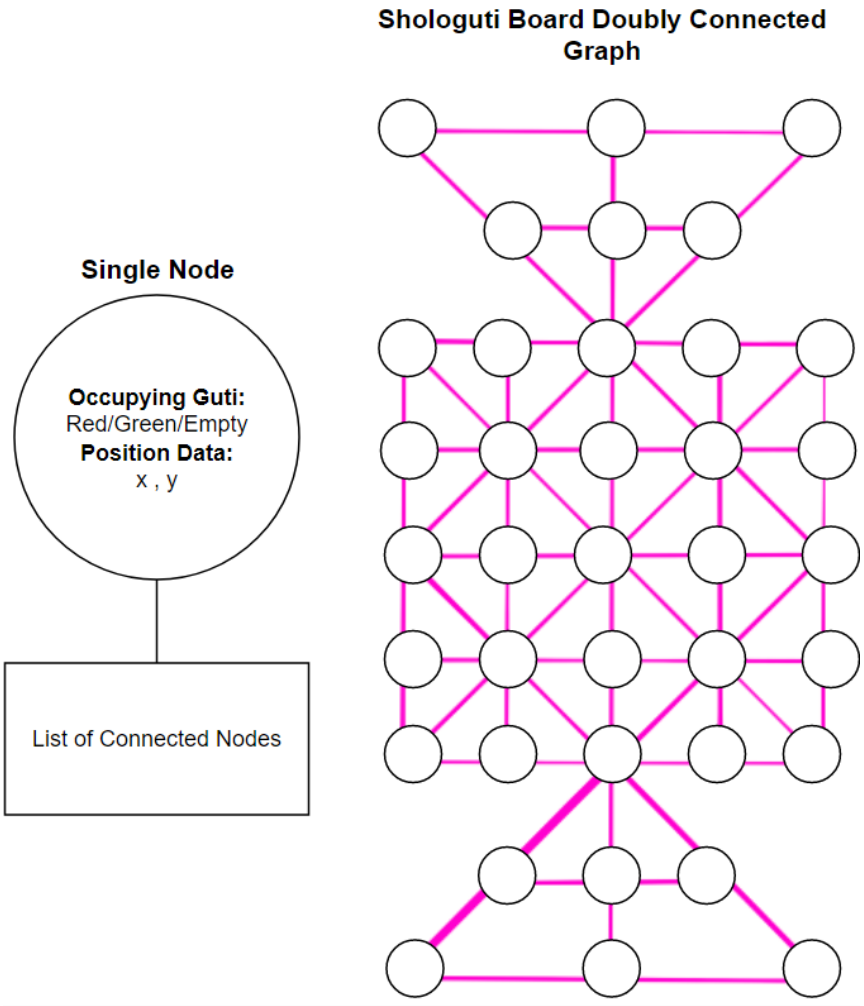

Figure 3.4 Shologuti board as a doubly connected graph

The initial layout of the board for Shologuti game is loaded from a JSON file. The JSON file contains information of the board layout, the initial positions of the Red and Green Pawns/Gutis, and information about which board position is connected to which other board positions. Since the board layout and the connections between the board positions are loaded from a JSON file, in the future, if we want to implement other variations of Shologuti game like Peralikatuma [https://en.wikipedia.org/wiki/Peralikatuma] which follows the same game mechanics as Shologuti but with each player starting with 24 pawns / gutis and a slightly different, larger board layout. The board is stored as a graph using a dictionary which uses cartesian coordinates as keys and stores board node objects. Board node objects store the connections they have to their neighbouring nodes along with which type of guti is currently occupying the node, Red/Green/None. The dictionary allows for much faster access to board nodes and speeds up searching operations performed on the board graph.

- **Creation of Guti / Pawns**

The visual objects that are seen on the board and spawned in into locations specified by the JSON initialization file. The guti objects are created using the Unity3D prefab system, where one pawn was created with no colour and only a sprite and a behaviour script that is needed for the guti to function inside the game. The colour of the guti can be selected while spawning, so in the future we may add option to let players change the colour of their pawns/gutis.

- **Move Generation**

When a pawn / guti is clicked, a number of Highlights are spawn on positions on the board to which the clicked guti can move to. In order to find these next possible positions for the selected guti, the rule checking system is used to find legal moves. Adjacent connected locations to the selected guti are searched for empty positions. If the adjacent connected position is held by a friendly guti, then the position is marked occupied. However, if an enemy guti sits on an adjacent connected position, the direction between the enemy guti and the selected guti is determined by subtracting the two positions (the position class is designed to behave like vectors) and then an empty position adjacent to the enemy guti is searched for along the same direction. If there is indeed an empty position behind the enemy guti, then that position is also marked as walkable and highlighted. For AI, move generation works the same, only instead of highlighting moves for a single guti, for MinMax all possible moves are generated for a given state as it is needed to simulate future board states.

- **Rule Checking System**

A dedicated rule checking system was made to check for game end states like draw conditions being met or victory conditions being met. The draw conditions apply when both players have equal scores and the maximum number of moves per match has been reached. The victory condition applies when one player reaches a maximum score of 16 or the maximum number of moves for the match have been reached and the winning player has a higher score. The rule system is called to check for game end conditions at the end of every move. The rules system is also used to filter legal from illegal moves during move generation. The rules system also has logic to determine which player's

turn it is next as some moves can give a player an extra turn and hence this logic has to also be checked every move. This module is also used by the action filters used by the RL agents.

- **Scoring system**

Game stores and updates scores for each player per match. A point is awarded for each enemy player's guti that is successfully captured. Along with the match scores, the win count for each player is also tracked and updated.

- **Multiple game agents**

Multiple types of Board Game AI have been implemented and tested for the Shologuti game. The Human Player can play against these AI or watch them play against each other to learn how to play. A lightweight simulator is created implemented using a copy of the board graph being used in the game. This simulator can be used to predict future states by all types of AI, without corrupting the original board state. The following board game AI have been implemented in game.

1. AI using MinMax adversarial search with rule based heuristic function and variable search depth was implemented. It uses the simulator object to simulate future states.
2. AI utilizing a trained PPO Agent that has an action policy network size of 64 x 64. AI uses AC-2 architecture as shown in figure 4.7.
3. AI utilizing a trained SAC Agent that has an action policy network size of 64 x 64. AI uses AC-2 architecture as shown in figure 4.7.
4. TD AI using NN architecture shown in figure 4.5. The TD agent does not have a pre trained neural network. Instead, it waits for connection to an external python script. If an external python environment is not found, then after a while, TD AI acts randomly.

■ **Adjustable Difficulty of AI**

The Difficulty level or the skill level for both the Tree Searching hardcoded AI and the Trained Reinforcement Learning Agents can be adjusted. For the Tree Searching agents the search depth is varied and for the reinforcement learning agents the experience level of the agent is change by swapping Neural Networks trained for longer periods with lesser trained Neural Networks.

■ **Local multiplayer**

The Game supports local multiplayer by allowing two human players to play against each other on the same board. The implementation of a fully functional game with a scoring and a rule system controlling the turns allows two players to play the turn-based game on the same device.

3.7 Implementation of User Interface

A good and intuitive interface is a major part of the user's experience. Hence, some care was taken when designing and implementing the user interface and creating the graphics and animations for it.

■ **User-Friendly Heads-Up Display.**

Information about the game state is presented to the user through rendered scoreboards and other rendered text on-screen that are updated live. These systems were made using Unity Canvas 2d system which allows 2d text and GUI interface to be made and rendered as an overlay over the main game being rendered by the Camera component of Unity. The information displayed include:

1. Score of each player
2. Which player's turn it is to move.
3. The type of player playing on each side, Human, AI, RL Agent
4. Difficulty of the AI controlling an opposing player.
5. If a player is controlled by an RL Agent, the type of algorithm used to train it is also displayed
6. Whether the game is Paused, In-Play or in a Victory Condition
7. The number of victories for each player

■ GUI Interface

The Shologuti game has colourful GUI interface. This was also made with the help of Canvas 2d. The GUI interface includes the following features:

1. Buttons allow users to pause the game, replay or navigate to and from the settings menu.
2. Clicking on a guti or pawn highlights it and all its possible moves. Clicking on the highlighted locations allows players to move their guti.
3. The GUI settings menu allows users change various settings like enemy AI and difficulty.
4. All GUI features are augmented with animations.

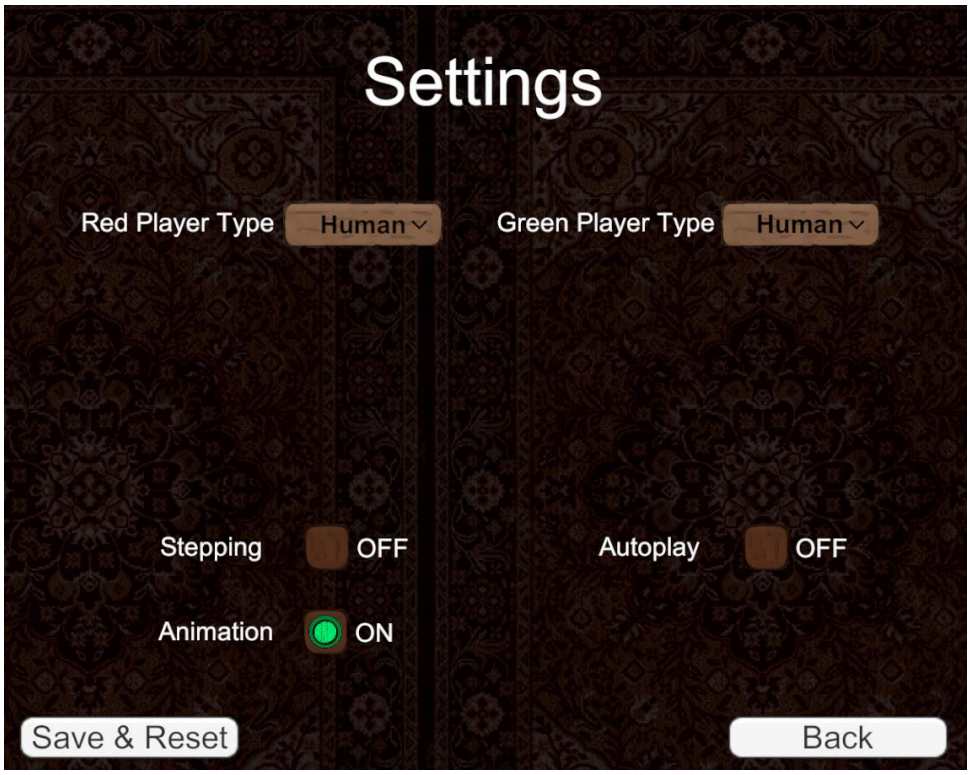

Figure 3.5 GUI settings interface

■ **Custom art assets and animations for feedback**

Custom sprites and images and animations were made for the board, backgrounds, pawns/gutis and the buttons to provide the user with a rich and visually interesting experience.

Highlights marking the selected guti and the locations to which it can be moved are all animated to add more visual flair. Hovering over guti or highlighted move locations triggers animations as well. All player moves are animated with the help of additional highlight sprites indicating source and destination positions along with special animation if a guti is being captured with the current move.

All animations can be skippable by clicking anywhere on the screen. Animations can be turned off entirely from the settings menu.

The animations were made using Unity 2d animations library. The custom graphics and art were made using Adobe Photoshop 2020.

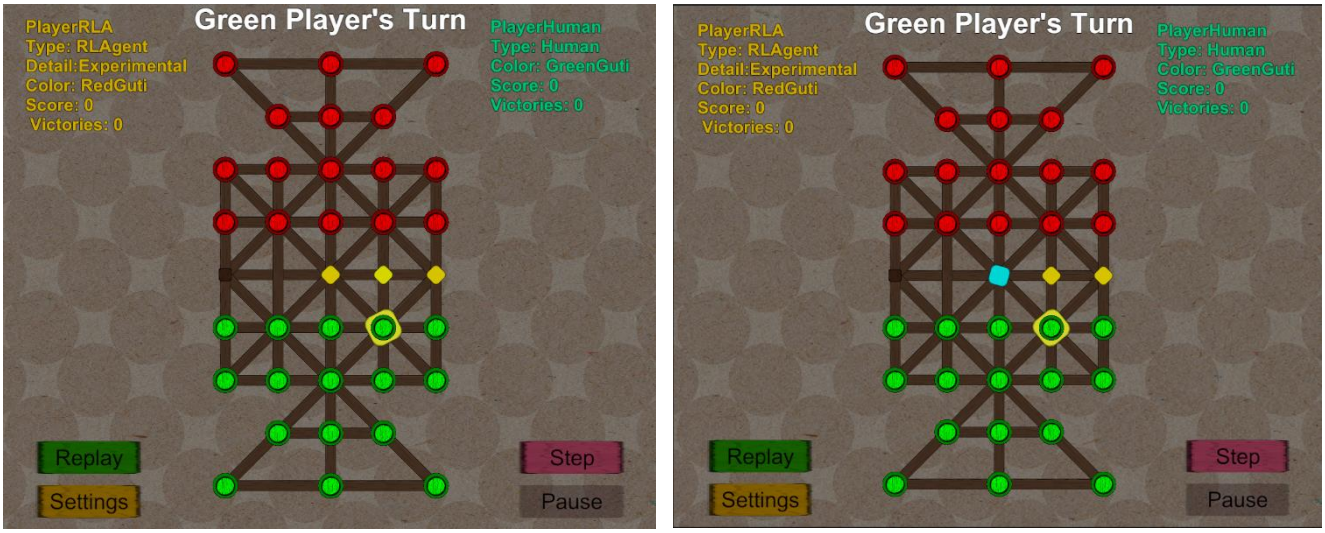

Figure 3.6 Animations and Graphics

4 Design and Implementation of RL Component

In this chapter we discuss the target features, and the design and implementation of those features for the RL component of the Shologuti project.

4.1 Learning Environment Features

In this section we discuss the features we aim to implement for Shologuti as learning and benchmarking environment for RL algorithms and agents.

1. Create a Custom Sensor Actuator system to enable communication of RL agents with external python scripts
2. Allow Benchmarking and train against MinMax AI with variable search depths
3. Enable Self-Play training of RL agents
4. Benchmark and train against pre-trained agents trained with PPO and SAC RL algorithms
5. Have AI vs AI game modes for strategy comparison between them.
6. Have a stepping mode to allow users to step through a game one action at a time. This is useful mainly for AI vs AI mode.
7. Allow Humans to play against RL agents at any time, for more detailed investigation of strategies being employed
8. Change settings of the learning/benchmarking environment using GUI
9. Create a cross platform environment that can communicate with accompanying Python environment

4.2 MinMax Adversarial Search based Agents

MinMax AI, on receiving a request for a move from the Game Manager, it calculates its Next Action using MinMax and then executes the move action. MinMax Algorithm, explores the game tree for a certain depth, simulating all possible combinations of board states. The following figure 4.1 MinMax Tree shows an exploration / simulation up to depth 3 of the Game Tree. When search depth reaches the 3rd level, it uses a heuristic function to estimate the score. The heuristic function is the score difference of the two opposing players in the simulated moves. The green arrows point up represent a Max node, which takes the values of the board states below it and only passes up the board state with the maximum value. The Red Arrow pointing downward is a Min node and it only passes up board state with the minimum value. Min and Max node layers alternate as the turns of the players alternate in the simulation of the game. with one Max node on top the player that calls the MinMax Algorithm gets the action leading to one board state at the end of computing that has the highest possible score for all-possible simulated board states for the next 3 turns.

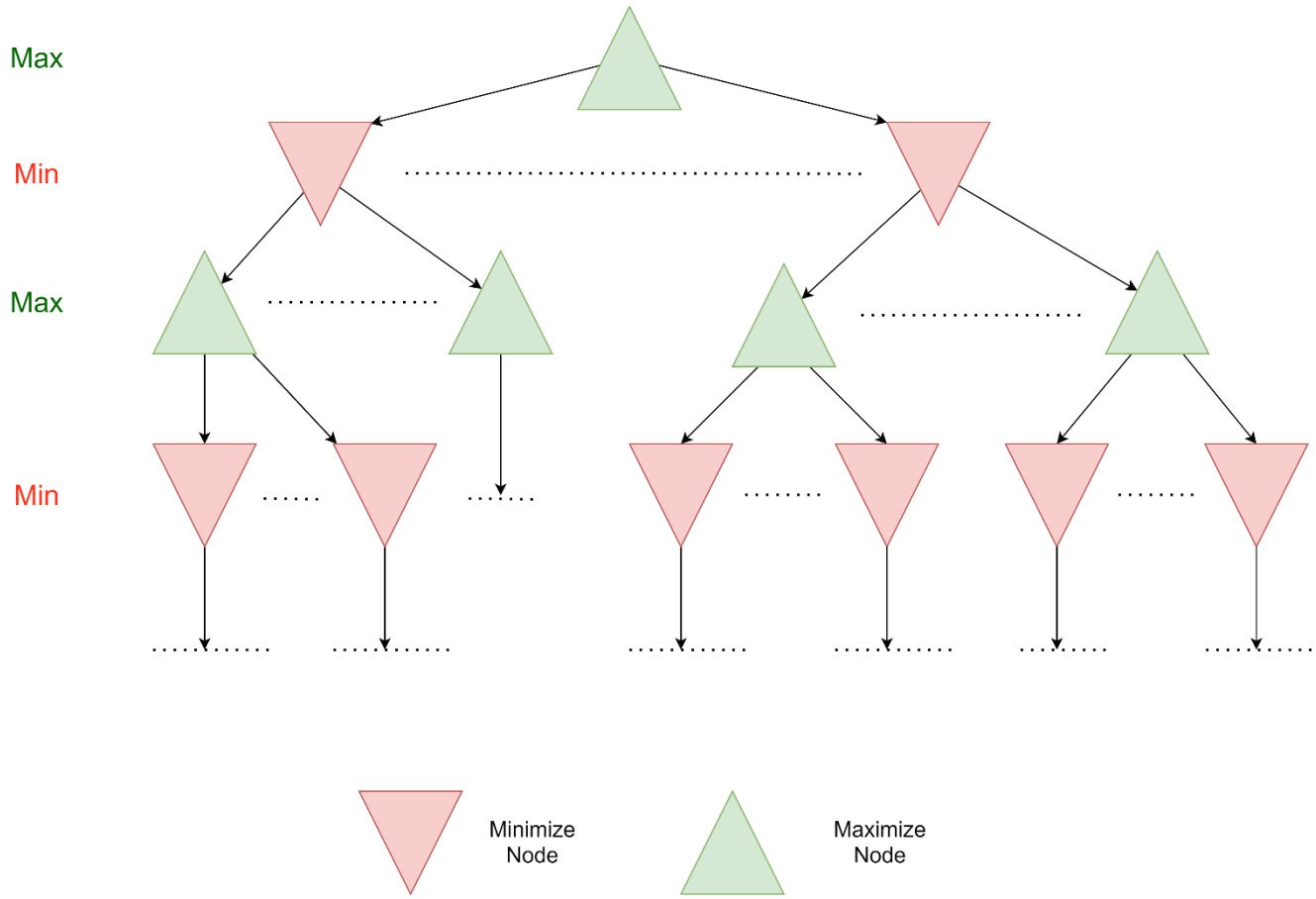

Figure 4.1 MinMax Game Tree

4.3 Design of RL Component in Unity

■ Temporal Difference (TD) Agent Decision/Inference Diagram

When a TD Agent receives request for a move, it collects observation for the current state of the board and then simulates the state of the board immediately after taking each of the possible actions it can take. Then all the possible states after taking one move, are passed as future state observations using the Sensor component to the Python environment or to a trained Baracuda NN model. Then the Agent waits for the state observations to be processed and returned through the Actuator as float values between 0 to 16. The agent then executes the action that produces the board state with the highest value. Figure 4.2 illustrates this whole process.

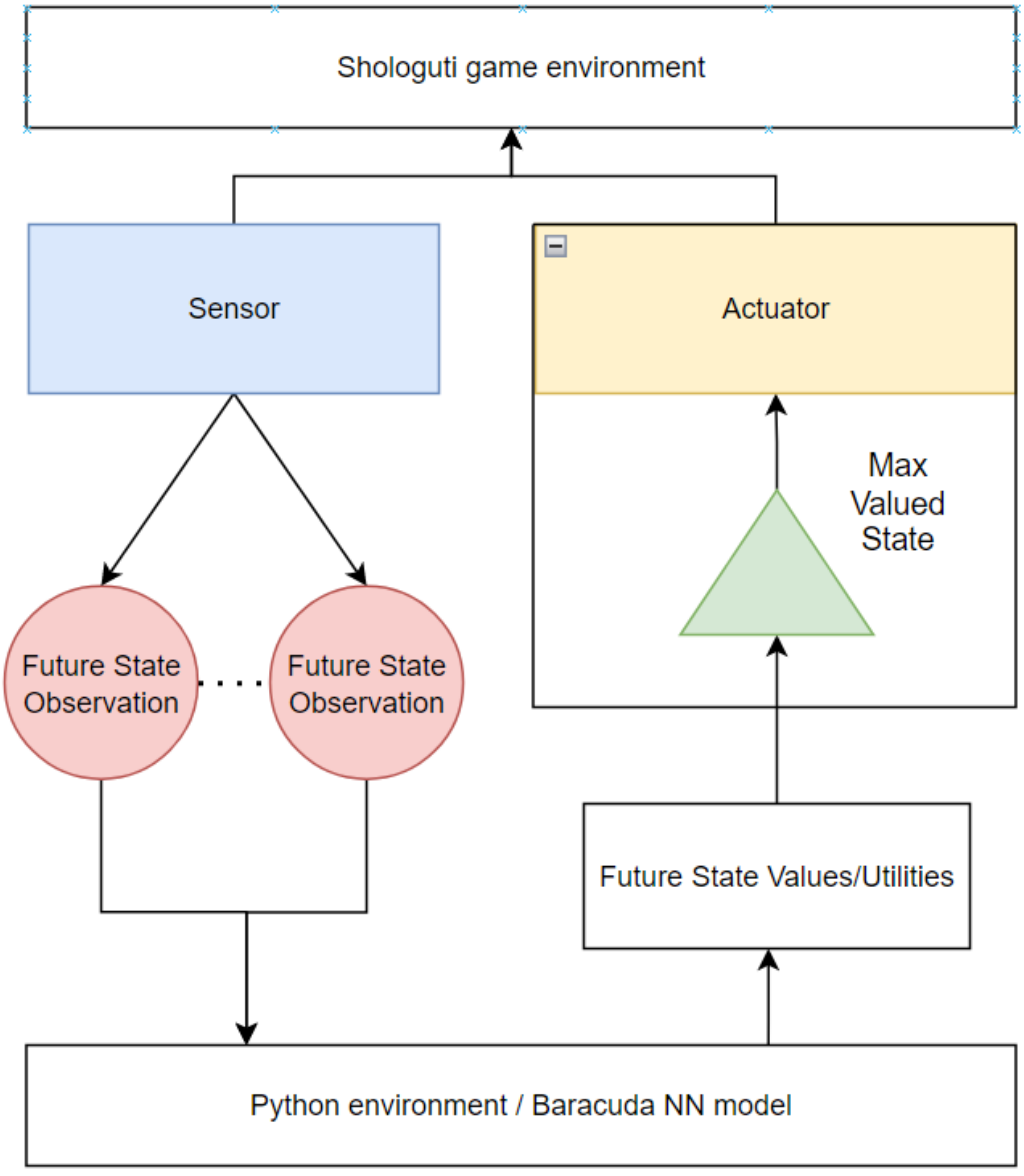

Figure 4.2 TD-Agent Decision Process

■ **Actor Critic (AC) Agent Decision/Inference Diagram.**

When an AC agent receives a request for a move, it collects observation for its current state and sends the observation to the Python environment/ Baracuda NN model. The observation is processed and a vector with the probabilities of each possible action and also illegal actions are sent back to the Actuator. The Actuator filters the illegal actions (all illegal actions for AC agents using a NN with AC-1 architecture as in figure 4.6 cannot be filtered) and also converts the vector output into a single action that is then carried out by the AC agent. Figure 4.3 illustrates this whole process.

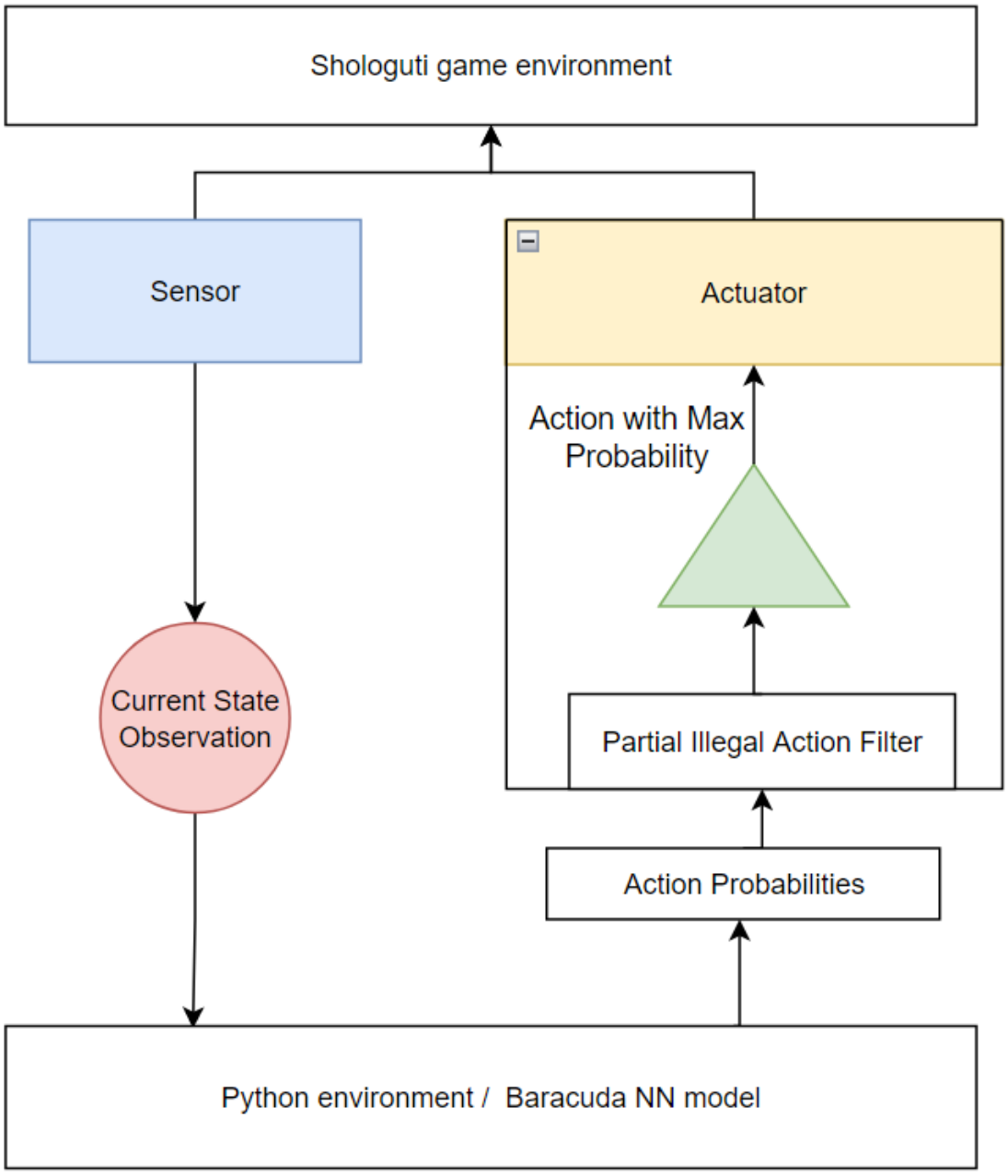

Figure 4.3 AC-Agent Decision Process

■ Observation Representation

Initial board state, where 1 represents green guti/pawn, 2 represents red guti/pawn and 0 indicates an empty node on the board. The following table 4.1 and figure 4.4 shows how the board state is represented and to the neural network. The result is a 1D vector of length 37.

1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	0	0	0	0
0	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2			

Table 4.1 Initial board state observation

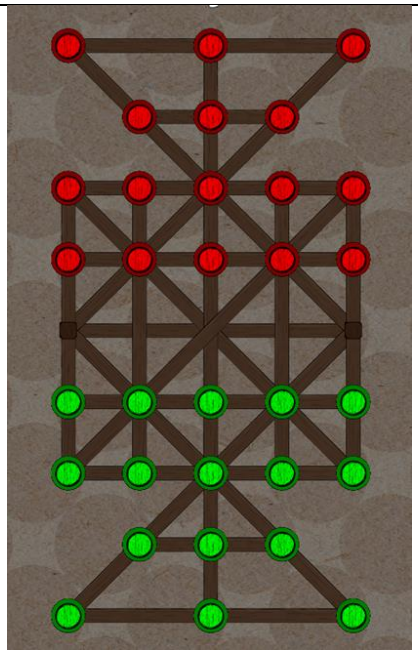

Figure 4.4 Initial board state

4.4 Neural Network Design

Three different Neural Network (NN) architectures were designed for this project that further widely categorize into Temporal Difference (TD) NN and Actor Critic (AC) NN architectures.

The figures showing the AC architectures only show the Actor or the Policy NN. These architectures all also have a Critic / Value NN like the one in figure 4.5 that is used while training and is trained alongside the Policy NN.

■ Temporal Difference (TD) Neural Network architecture

TD NN architecture was the first architecture created for the entire project. It takes a vector observation of length 37 represent a state and evaluates with a float value between 0 to 16. The NN alone is not enough to take actions and can be used as a heuristic for searching algorithms like MinMax. Figure 4.5 shows the TD NN architecture. Figure 4.2 shows how this architecture is used for decision making in the game environment.

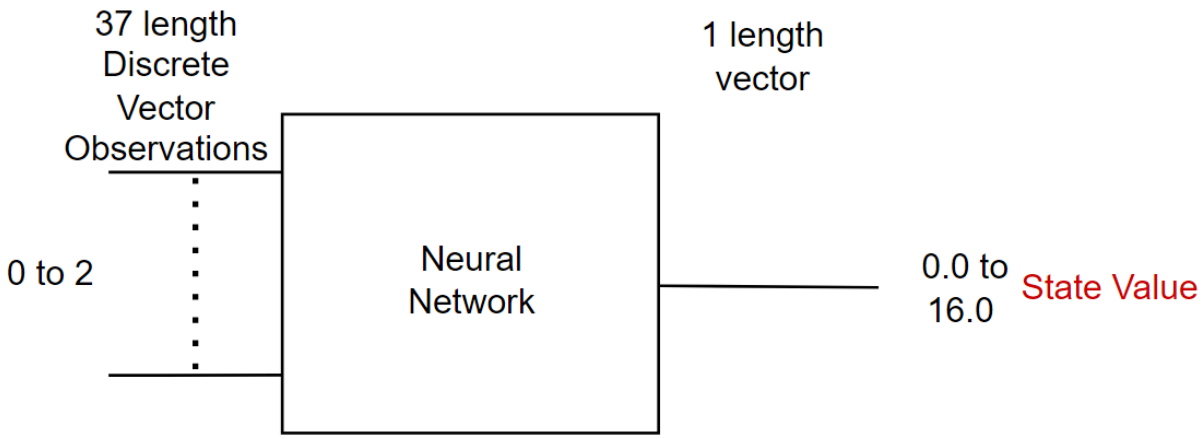

Figure 4.5 TD NN architecture

■ AC-1 architecture

AC-1 architecture was the first AC architecture made for this project. It takes a vector of length 37 as input and outputs a vector of length 74, 37 + 37, with the first 37 elements adding up to one and next 37 elements also adding up to 1 separately. Hence, the first 37 output heads give the probability of choosing between 37 possible selectable source nodes and the next 37 output heads give us the probability of the selected target node where the guti on the source node should be moved. Note, this architecture allows for many illegal moves, and hence the network output is filtered by the action filter to limit the number of illegal combinations that can be produced by the NN. The action filter also converts the output into two discrete variables ranging from 0 to 36. These are used as indexes to directly to actions in the Shologuti environment. Despite this, some illegal combinations of actions are still possible as even after filtering illegal selectable nodes and illegal target nodes, the NN can produce illegal permutations by selecting a valid selectable node but mapping it to an illegal target node. Figure 4.6 illustrates the AC-1 architecture.

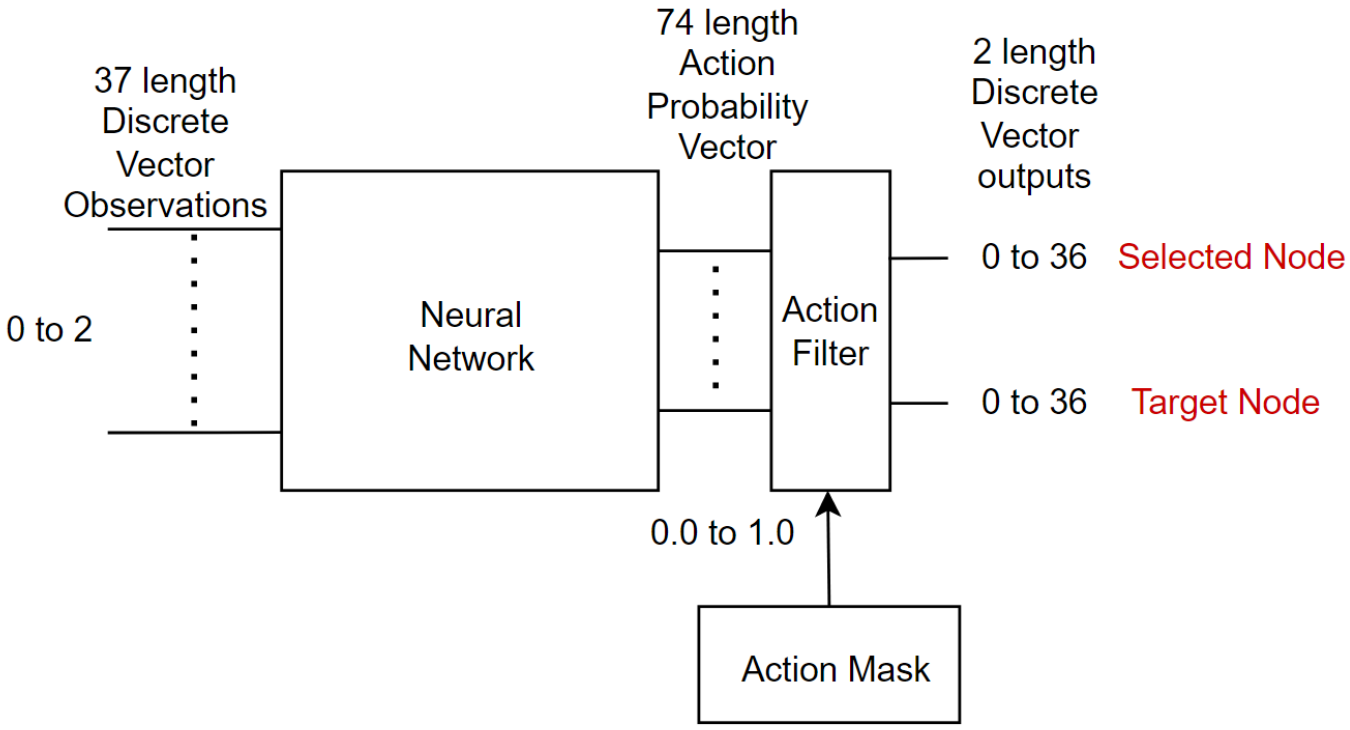

Figure 4.6 AC-1 NN architecture

■ AC-2 architecture

AC-2 architecture is almost identical to AC-1, covered in section 4.4 previously, except for its output vector. AC-2 outputs a probability vector of length 24, 16 + 8, where the first 16 add up to 1 and the next 8 add up to one. The first 16 represent the selected guti (the game starts with 16 for each side). The next 8 represent the possible moves the guti can make (8 is the maximum possible moves that can be taken by a guti). The action filter filters out missing gutis or gutis with no available moves. The NN can still map to illegal moves as 8 moves are always not available to each guti. To remedy this, a mod operator is run on the move selection output with the number of legal actions for the selected guti, after it has been converted to a discrete variable by the action filter. This allows illegal actions to be mapped to legal ones. This will however add bias to certain actions when the move count is not factor of the max move count 8, as certain actions will be mapped to from more output heads than others. Figure 4.7 illustrates the AC-2 architecture.

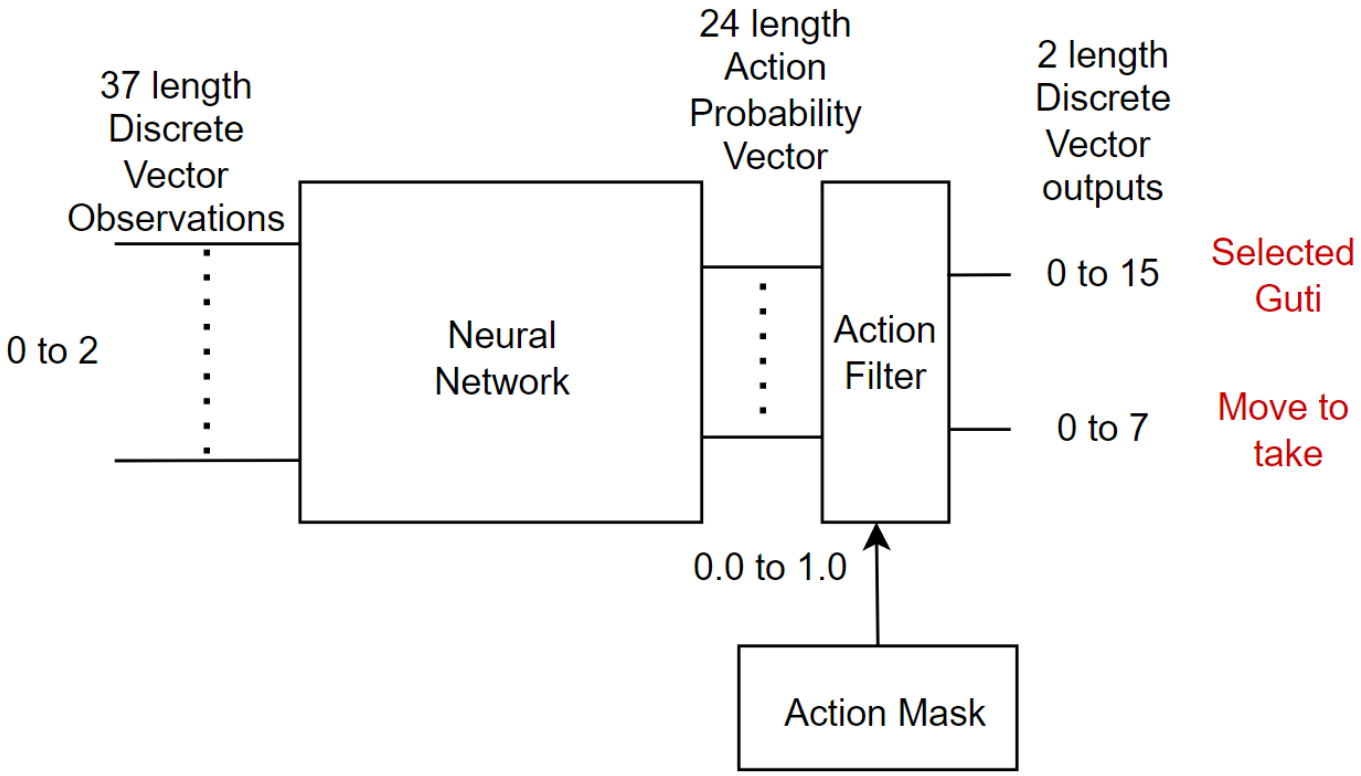

Figure 4.7 AC-2 NN architecture

4.5 Agent Training Process Design

■ Agent Training Process using Unity ML-Agents Trainer

Figure 4.8 shows the process of training an agent using the trainer provided by Unity ML-Agents. The ML-Agent trainer also makes use of the Baracuda library to convert the NN trained in PyTorch into a format that can be used directly by the Unity Game Engine. The. nn file generated by Baracuda can be converted to (ONNX) and many other formats, allowing it to be used natively in Windows, Web, MacOS, Linux and Android. Any number of RL agents can be trained at the same time, as each agent is tagged with a team number and receives actions and rewards according to its team tag. All the deployed agents update the same neural network. The RL algorithms use the observations to produce actions and then use the rewards sent in response to the action to train.

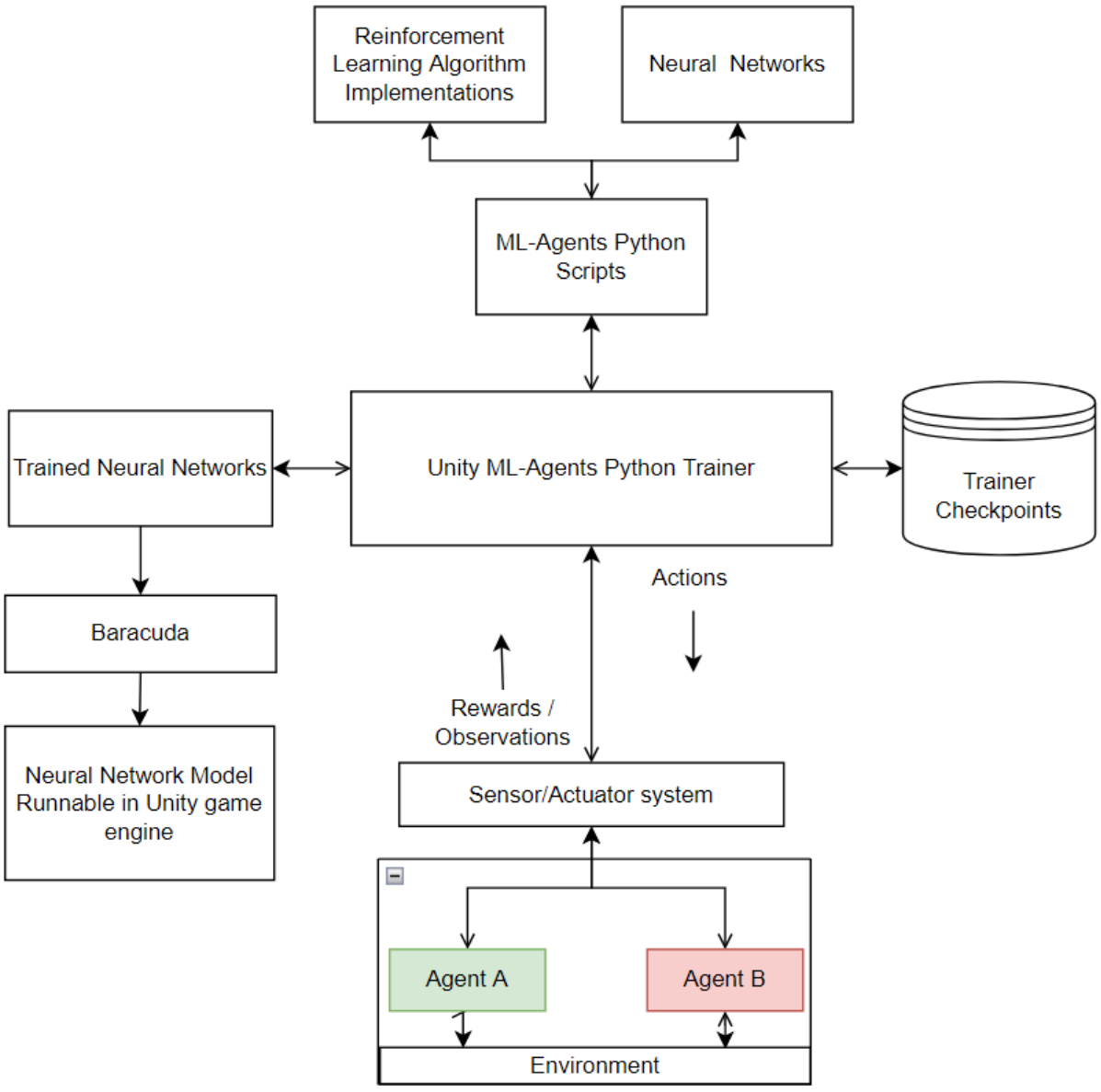

Figure 4.8 Agent Training Process

■ Agent Training Process with Custom RL Algorithms

Figure 4.9 shows the process of training an agent with custom python scripts and custom RL algorithm implementations. Baracuda was not used in the custom trainers implemented for this project, hence the trained neural networks can only be run using the python interface provided with this project.

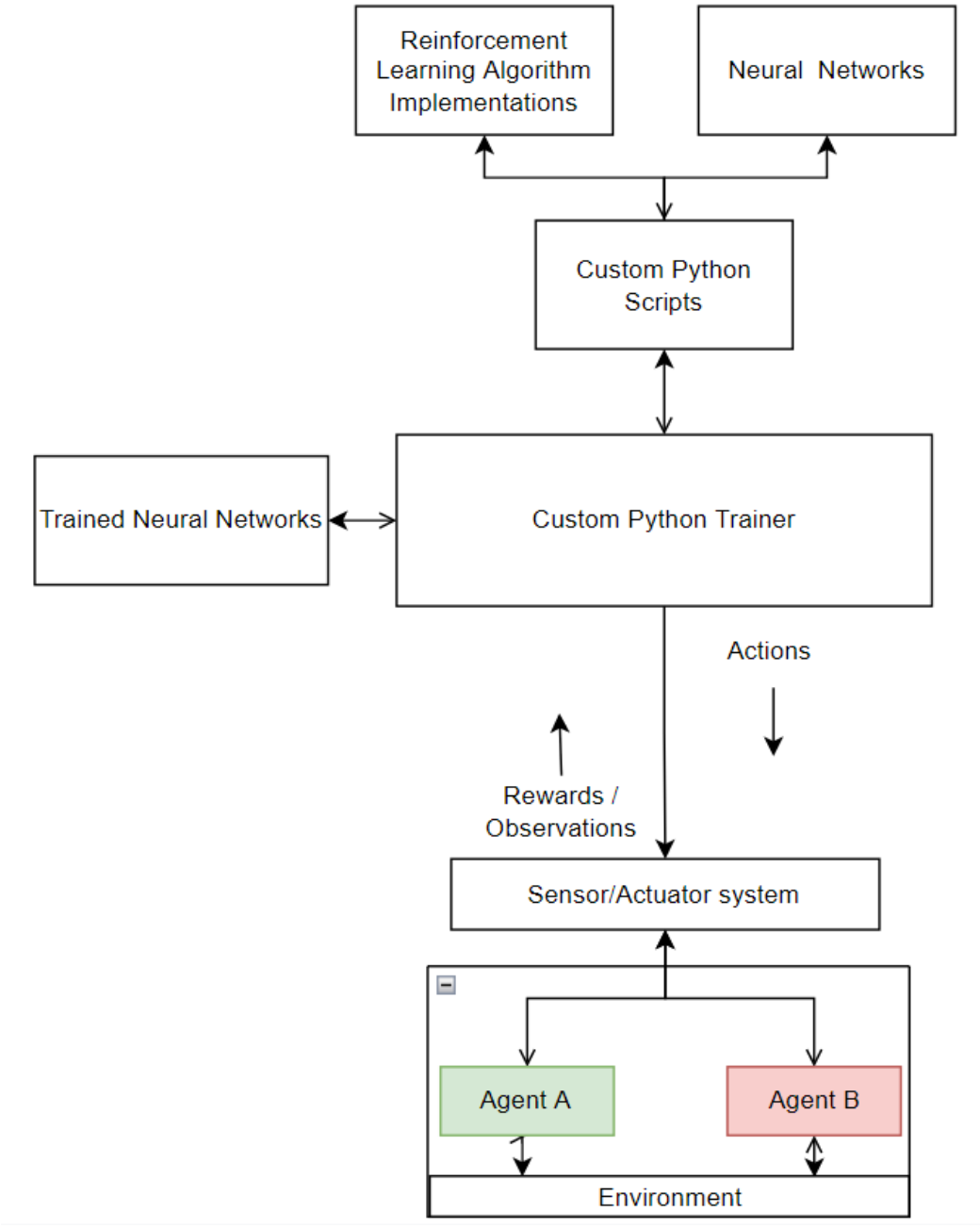

Figure 4.9 Agent Training Process with Custom RL Algorithms

■ State and Action mirroring to facilitate Self-Play

To enable the RL agents to play on any side of the board while also learning, a mirroring module was added to all the actuator and the sensor components of the system. The sensor mirroring module converts observations from both player 1 and player 2 into player 1 perspectives. Thus, the sensor always sends observations to the NN from the player 1 perspective. The actuator mirroring component always receives actions from player 1 perspective but can them to player 2 perspective if the agent requesting the decision was player 2. Thus, no destructive relearning needs to be done when switching player perspectives and all the training and knowledge gained as player 1 can be used as player 2. This enables us to also train using self-play while constantly switching the player that is training.

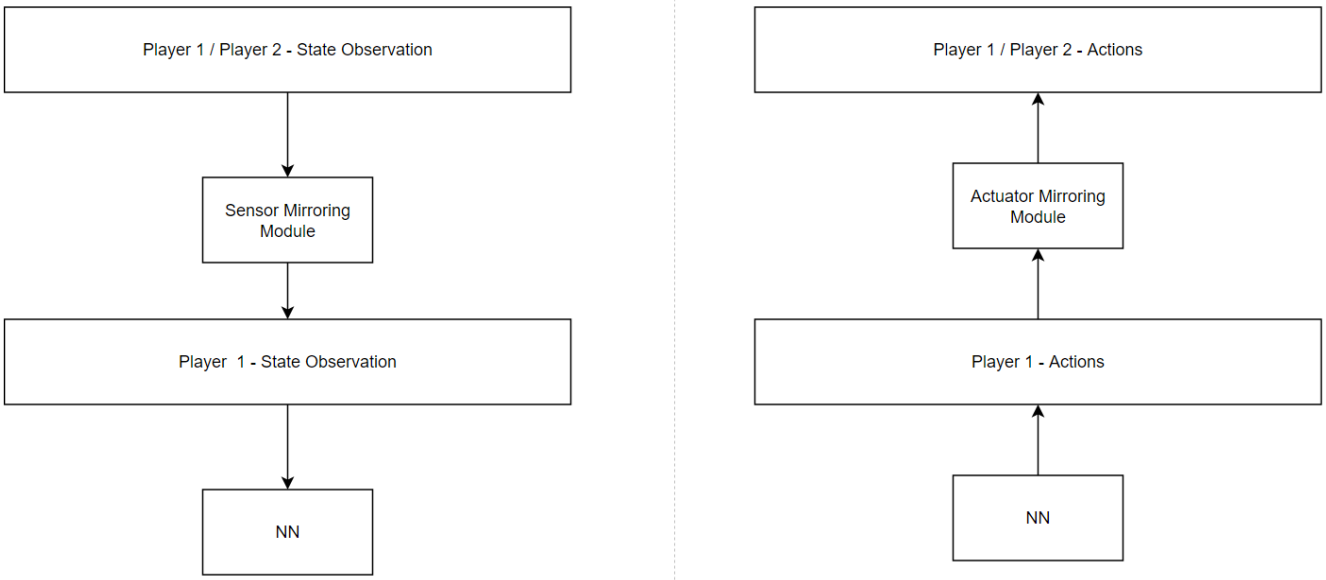

Figure 4.10 Observation and Action mirroring

■ **Generalized RL Algorithm Diagram**

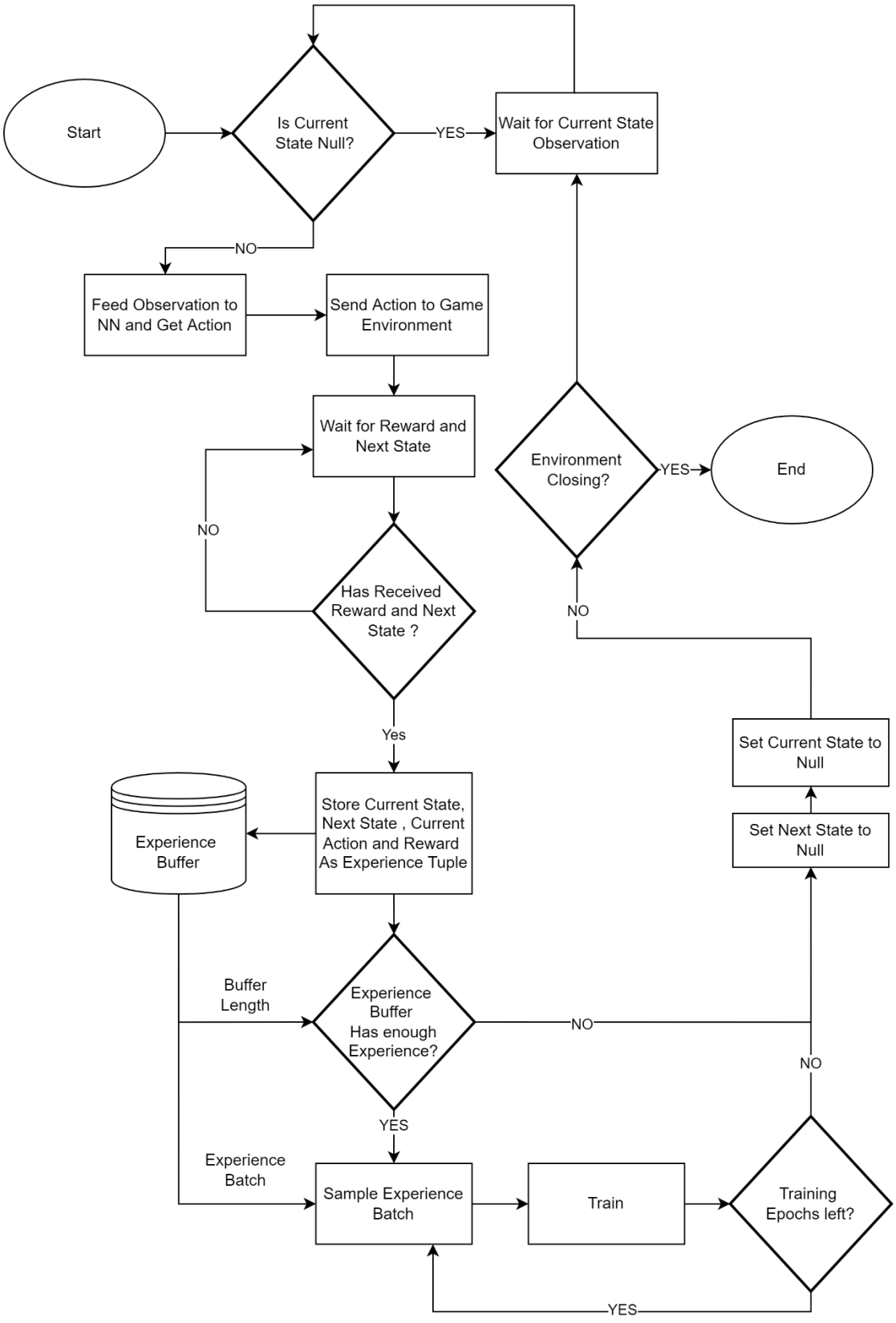

Figure 4.11 General RL algorithm diagram

4.6 Learning Environment Feature Implementation

■ **Custom Sensor system**

The collection of sensors provided by Unity ML-Agents did not have a method for collecting data from turn-based games. Hence, the sensor interface provided by Unity ML-Agents was extended to allow for use in a board game environment. In total 3 types of sensor components were made to provide 3 different types of outputs to external python scripts. All sensors send both observations and rewards after the agent performs an action to a NN or to a python environment. The rewards are sent the same way for most sensors but the observation representation sent changes.

1. Actor-Critic sensor: A sensor that simply observes and coverts the board state to a list and sends a list of floats representing different state of each node on the board. As there are 37 nodes in a Shologuti board, the sensor observes and sends a list of 37 floats. This sensor is used by the agent's labelled SAC and PPO in the environment figure 4.12.
2. Experimental Actor-Critic sensor: Another sensor sends one hot vectors for each node state. The nodes can be occupied by player 1 or player 2 or it could be empty. Hence, a board node can have 3 states and the sensor therefore sends 3 x 37 = 111 length list of floats. This is used by the Experimental agents in the environment labelled Expr. Fig 4.12. This sensor system is not properly tested but, it can be accessed and controlled and trained using external scripts. It is adding to the array of observation types the Shologuti environment offers and was made for use in future for custom Alpha Zero based RL algorithm implementations.
3. TD sensor: A third sensor component observes all possible immediate future steps (like a 1-ply search in MinMax), and sends them to a neural network or an external python script for evaluation. The actuator component waits for every state that was sent in turn to be evaluated. The actuator translates this information into possible future actions with potential future scores that can be used to act optimally. The rewards sent back by the sensor in the case of a TD agent is used to update the values it produces for every observation sent. The agent labelled TD in the environment makes use of TD sensor as shown in figure 4.12

All sensor components allow for communication with python interfaces using ML-Agent's python module. However, a further wrapper to make the interface Open AI gym-like is only available for the 3rd sensor type, TD from the list above.

Figure 4.12 Trainable/RL Agent types

■ **Custom Actuator system**

Three different actuator components were made to interpret information sent back by 3 different types of network architectures namely AC-1, AC-2 and TD figure 4.6 and 4.7 and 4.5. The actuators convert the probabilities output by the agent into discrete integer variables that can be easily mapped to meaningful actions in the environment. Actuators also have an action filter to filter out as many illegal actions as possible.

1. AC-1 actuator covered already in this chapter previously in section 4.4. Converts action probabilities with size 74 into two discrete integers with values ranging from 0 to 36 each. Before conversion some illegal actions are filtered from the 74-length vector. If the agent still manages to takes an illegal action after filtration, the action is discarded and another action is requested for the same state. The two discrete integer values are converted to an action and the action is taken in the environment.
2. AC-2 actuator covered already in section 4.4. Converts action probabilities with size 24 into two discrete integers with values ranging from 0 to 16 and 0 to 7. Before conversion all illegal actions are filtered. The two discrete integer values are converted to an action and the action is taken in the environment.
3. TD actuator covered already in section 4.4. The actuator receives values for all the future states that were sent as observations and then takes the action that resulted in the states with the highest value.

■ **Benchmark and train against variable MinMax Depths**

The environment has AI implemented using MinMax with variable search depths and competent heuristic functions. These can be used as opponents to train against or benchmark against by pre-trained or currently training RL agents.

■ **Enable Self-Play Training**

Self-play was enabled with some clever mirroring of the observations and actions which allows the agent to be trained on one side of the board and generalise for both sides. The neural network being trained always receives observations as player 1 and sends actions as player 1. If the agent requesting a decision was player 2, then the sensor and actuator mirror the observations and actions which allows player 2 to use a NN trained by player 1.

Hence, no destructive relearning needs to take place and the agent can play against itself safely. This has already been discussed in section 4.5 and figure 4.22.

- **Benchmark and train against agents pre-trained with PPO and SAC**

Pre-trained RL agents trained using SAC and PPO algorithms are provided with the simulator to use for benchmarking and training other RL agents. This training process has already been covered in section 4.5 and figures 4.23 and 4.20. The NNs were trained using the trainer script and implementations of SAC and PPO provided Unity ML-Agents. The NNs that were trained used the AC-2 network architecture figure 4.7.

- **AI vs AI game modes for strategy comparison**

The different types of AI provided by the environment can be set to play against each other to gauge their performance by simple observation and seeing the victory counters. Their actions can also be observed to see what strategy each agent employs or compare the symbolic MinMax AI against the RL agents. RL agents trained with different strategies can be pitted against each other as well.

- **Stepping mode**

The simulator has a stepping mode that can be enabled from the settings menu that allows users to control the progression of the game manually using the step button (figure 4.13). This is useful specially when viewing a match between two AI, as the simulator stops after every action and the user can view and analyse the move and strategies being used by the AI thoroughly.

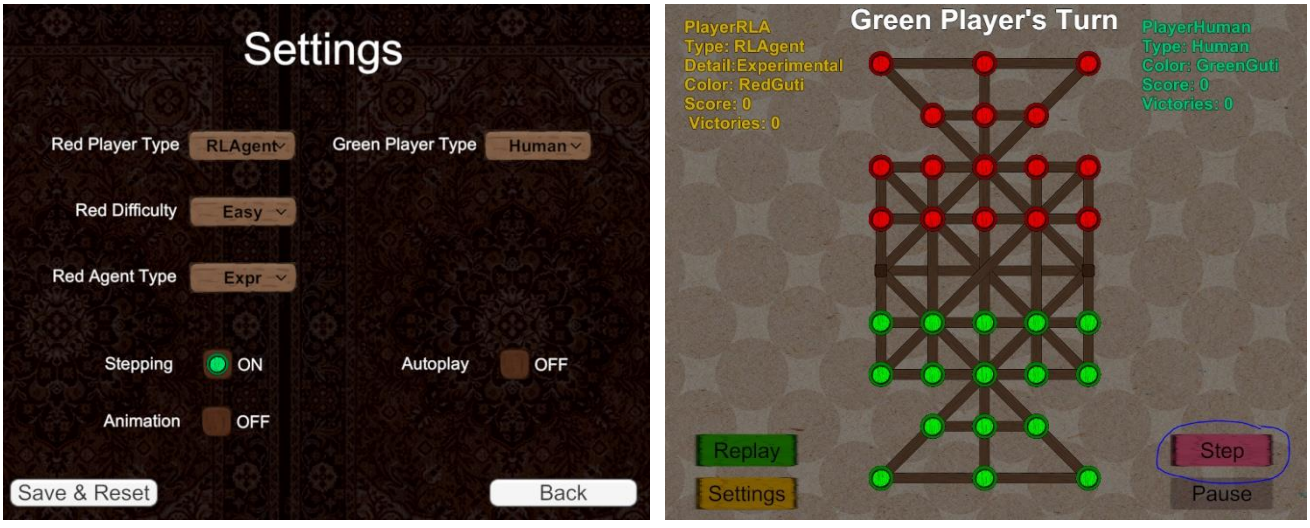

Figure 4.13 Stepping mode

■ **Evaluation of AI with human as opponent**

Humans can play against the pre-trained AI, the MinMax AI and also agents connected to the environment using the python wrapper. This allows users to evaluate the agents and study them by playing against them directly. This can also be used to benchmark the AI against human players in the future.

■ **Change settings of the environment using GUI**

All the relevant settings of the environment can be changed using the Settings GUI page. Figure 4.14 shows all the settings in the environment that can be controlled using the GUI interface.

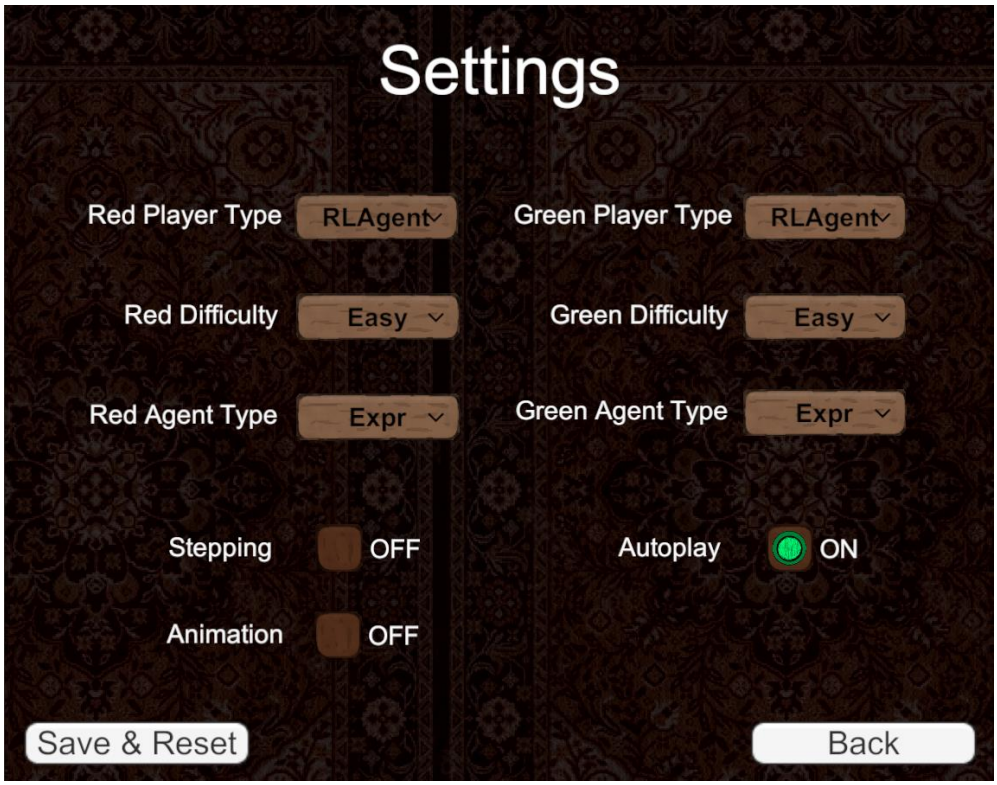

Figure 4.14 Settings Interface

4.7 Training RL Agents using Unity ML-Agents

■ Training Agents using default ML-Agents scripts

To train using ML-Agents scripts, first ML-Agents python package was installed into a virtual environment from their git-hub page https://github.com/Unity-Technologies/ml-agents. After installation a running unity environment with an RL agent waiting for input from an external script can be trained using the following console command. The conception diagram for this is found in figure 4.8.

```
mlagents-learn <trainer-config-file> --run-id=<run-identifier>
```

Agents can also be trained using a compiled executable version of Shologuti environment by specifying the full path of the executable using the –env argument.

```
mlagents-learn <trainer-config-file> --env=<env_name> --run-id=<run-identifier>
```

- **<trainer-config-file>** is the file path of a trainer configuration YAML. The YAML file contains hyperparameters for the trainer including which algorithm to use to train.
- **<env_name>(**Optional) is the full path including name of the Shologuti executable containing the agents to be trained.
- **<run-identifier>** is a unique name that is used to differentiate one training run from another

```
mlagents-learn <trainer-config-file> --run-id=<run-identifier> --resume
mlagents-learn <trainer-config-file> --run-id=<run-identifier> --force
```

A previous training run can also be resumed using --resume or overridden using –force.

The following figure – shows a trainer-config YAML file that needs to be configured to use Unity ML-Agents default trainer.

```yaml
behaviors:
  GutiBehaviour:
    trainer_type: ppo
    hyperparameters:
      batch_size: 256
      buffer_size: 2048
      learning_rate: 3e-4
      epsilon: 0.3
      lambd: 0.99
      num_epoch: 8
      learning_rate_schedule: constant
    network_settings:
      normalize: false
      hidden_units: 64
      num_layers: 2
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0

    init_path: C:/Users/samin/Unity/ml-agents/results/Shologuti PPONewIL3/GutiBehaviour
    checkpoint_interval: 50000
    keep_checkpoints: 5
    max_steps: 10000000
    time_horizon: 32
    summary_freq: 1000
    threaded: true
```

4.8 Cross platform Python accessible environment

A game environment that can be accessed using Python for testing and use in Windows, Linux and MacOS has been built. This is done with the help of ML Agents that provides methods to connect a C# environment to a python environment. Figure 4.9 shows the overview of how an external python script can access, control, interact with and train agents in the Shologuti environment.

■ Connecting to Python Environment

RL Agents can optionally connect to a python environment using ML-Agents low level python API in order to send its observations and receive actions. The python Environment may use a NN to control these decisions. It can also the resulting rewards the environment sends back to train the NN. To use the low level python API , the ML-Agents python package must be installed.

```
from mlagents_envs.environment import UnityEnvironment
env = UnityEnvironment(file_name='SholoGuti', seed=1, side_channels=[])
env.reset()
behavior_name = list(env.behavior_specs)[0]
print(f"Name of the behavior : {behavior_name}")
spec = env.behavior_specs[behavior_name]
print("Number of observations : ", len(spec.observation_shapes))
print("Observation vector shape: ", spec.observation_shapes)
decision_steps, terminal_steps = env.get_steps(behavior_name)
```

Figure 4.15 Connecting to Unity3D environment from python

■ Python environment Commands

- **Reset:** env.reset() Sends a signal to reset the environment. Returns None.
- **Step:** env.step() Sends a signal to step the environment. Returns None. Note that a "step" for Python does not correspond to either Unity Update nor FixedUpdate. When step() or reset() is called, the Unity simulation will move forward until an Agent in the simulation needs a input from Python to act.
- Close: env.close() Sends a shutdown signal to the environment and terminates the communication.
- **Behaviour Specs:** env.behavior_specs Returns a Mapping of BehaviorName to BehaviorSpec objects (read only). A BehaviorSpec contains the observation shapes and the ActionSpec (which defines the action shape). Note that the BehaviorSpec for a specific group is fixed throughout the simulation. The number of entries in the Mapping can change over time in the simulation if new Agent behaviors are created in the simulation.
- **Get Steps:** env.get_steps(behavior_name: str) Returns a tuple DecisionSteps, TerminalSteps corresponding to the behavior_name given as input. The DecisionSteps contains information about the state of the agents that need an action this step and have the behavior behavior_name. The TerminalSteps contains information about the state of the agents whose episode ended and have the

behavior behavior_name. Both DecisionSteps and TerminalSteps contain information such as the observations, the rewards and the agent identifiers. DecisionSteps also contains action masks for the next action while TerminalSteps contains the reason for termination (did the Agent reach its maximum step and was interrupted). The data is in np.array of which the first dimension is always the number of agents note that the number of agents is not guaranteed to remain constant during the simulation and it is not unusual to have either DecisionSteps or TerminalSteps contain no Agents at all.

- **Set Actions:** env.set_actions(behavior_name: str, action: ActionTuple) Sets the actions for a whole agent group. action is an ActionTuple, which is made up of a 2D np.array of dtype=np.int32 for discrete actions, and dtype=np.float32 for continuous actions. The first dimension of np.array in the tuple is the number of agents that requested a decision since the last call to env.step(). The second dimension is the number of discrete or continuous actions for the corresponding array.
- **Set Action for Agent:** env.set_action_for_agent(agent_group: str, agent_id: int, action: ActionTuple) Sets the action for a specific Agent in an agent group. agent_group is the name of the group the Agent belongs to and agent_id is the integer identifier of the Agent. action is an ActionTuple as described above. Note: If no action is provided for an agent group between two calls to env.step() then the default action will be all zeros.

■ **Training Agent in the Python Environment**

To train the agent we can fetch DecisionSteps using: env.get_steps(behavior_name: str)

A **DecisionSteps** has the following fields:

- **obs** is a list of numpy arrays observations collected by the group of agents. The first dimension of the array corresponds to the batch size of the group (number of agents requesting a decision since the last call to env.step()).
- reward is a float vector of length batch size. Corresponds to the rewards collected by each agent since the last simulation step.
- **agent_id** is an int vector of length batch size containing unique identifier for the corresponding Agent. This is used to track Agents across simulation steps.
- **action_mask** is an optional list of two-dimensional arrays of booleans which is only available when using multi-discrete actions. Each array corresponds to an action branch. The first dimension of each array is the batch size and the second contains a mask for each action of the branch. If true, the action is not available for the agent during this simulation step.

Rewards extracted from the DecisionSteps list can be used for training the agent implemented in python.

4.9 Connecting to Unity using Python Wrapper for TD Agent

This is a python wrapper built over the low-level python API provided by Unity ML-Agents. It was created for ease of communication with the TD Agent in the Shologuti environment. As the TD agent sends a variable length series of observations representing all the immediate future states, it can be difficult to use as it is hard to tell when one environment step ended and another began. The **ShologuUnityWrapper** for python encapsulates the tedious work and instead provides two functions to easily interact with the TD Agent. The **gather_experience()** function gathers experience tuples of (previous observation, next observation , reward). These experience tuples can be stored in an experience buffer and sampled to train a Temporal Difference (TD) learning agent. The other function **run_agent()** simply runs the agent.

Sample Experience tuple:

```
Experience Tuple:
prev_obs: tensor ([ 2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  1.,  0.,  0.,  2.,
2.,  2.,  2.,  2.,  2.,  0.,  2.,  2.,  0.,  2.,  0.,  2.,
2.,  2.,  2.,  2.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
0.]),
next_obs: tensor ([ 2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  1.,  0.,  0.,  2.,
2.,  2.,  2.,  2.,  2.,  2.,  0.,  2.,  0.,  2.,  0.,  2.,
2.,  2.,  2.,  2.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
0.]),
reward: -13.0)
```

Sample Code:

```
from typing import List
import torch
import SholoGutiUnityWrapper
from experience import Buffer
from model import NN

def main():
    # Launch and store connection to environment
    env = SholoGutiUnityWrapper.SholoGutiUnityWrapper(path=
                'C:/Users/samin/OneDrive/Desktop/SholoGutiWindowsBuild/SholoGuti',
                time_out_wait=120,  no_graphics = True)
    print("Environment Created")
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    nn = NN(38, 32, 1).to(device)
    for i in range(0, NUM_TRAINING_STEPS):
        try:
            # run_agent(env, nn)
            gather_experience(env, nn, experience_buffer)
            env.reset_env()
        except Exception as e:
            print(e)
            env.close_env()
            return
    env.close_env()
```

5 Training and Benchmarking with RL Algorithms PPO and SAC

In this section we discuss the benchmarks and evaluations we wish to produce for two different RL algorithms Proximal Policy Optimization (PPO) and Soft Actor Critic (SAC) using implementations of them provided by Unity ML-Agents toolkit.

7 different training experiments were conducted where some of the 4 experiment variables shown in figure 5.1 were changed. The training experiments were divided into 4 different training setups. The training setups are discussed in detail in section 5.1 and 5.3. The 4 variables (summarized in figure 5.1) tweaked during the experiments are as follows:

1. **NN architectures**: There are two types of NN architectures used for the experiment AC-1 and AC-2, both of which have been covered in section 4.4 before.
    a. **AC-1** NN architecture, figure 4.6.
    b. **AC-2** NN architecture, figure 4.7, was made to eliminate illegal actions after the observation that AC-1 NN architecture allowed many illegal actions to be taken, which led to agents wasting time trying them during training.
2. **Training methods:** The two types of training structures include training against agents using MinMax search algorithm or training using self-play.
    a. **Training against agent using MinMax:** The MinMax using agents can search to a depth of 1 or 2 and the search depth is accounted for in the training variable as shown in figure.
        i. MinMax with search depth of **1**
        ii. MinMax with search depth of **2**
    b. **Training with self-play:** The RL agent trains by playing against older and more recent copies of itself.
3. **Reward structures:** 3 different reward structures were designed to train agents with. (Covered in section 5.2).
    a. **R1** is a traditional reward structure with rewards for winning or capturing enemy guti and penalties for enemy achieving the same things or taking illegal actions. The rewards have high magnitude, to encourage faster learning and convergence to maxima even if it is suboptimal which is useful for slow learning environments.
    b. **R2** maintains the ratio of the rewards in R1 but scales everything down to within 1 for better stability over longer (higher number of moves/steps) training sessions which is possible in faster learning environments. R2 also adds rewards for reaching **intermediate goal states** and also gives $1/8^{th}$ the reward for winning if an agent wins without capturing all 16 enemy gutis.

c. **R3** is copy of R2 but instead of extra rewards for reaching intermediate goal states, the agent is given extra rewards by a reward generation system called Curiosity which gives rewards for finding new and unseen states.

4. **Number of parallel learning environments:** Initially the Shologuti environment did not have the feature to run multiple parallel learning environments together, hence the first training setups TR1 and TR2 only use 1 parallel learning environment. Even then, the number of parallel learning environments is still limited to 1 if the agents are training against MinMax opponents. Higher number of parallel learning environments drastically increase the speed of training steps taken and make long training sessions more viable. The Number of parallel learning environment variable takes one of the following values.

   a. **1** learning environment

   b. **4** parallel learning environments

   c. **8** parallel learning environments

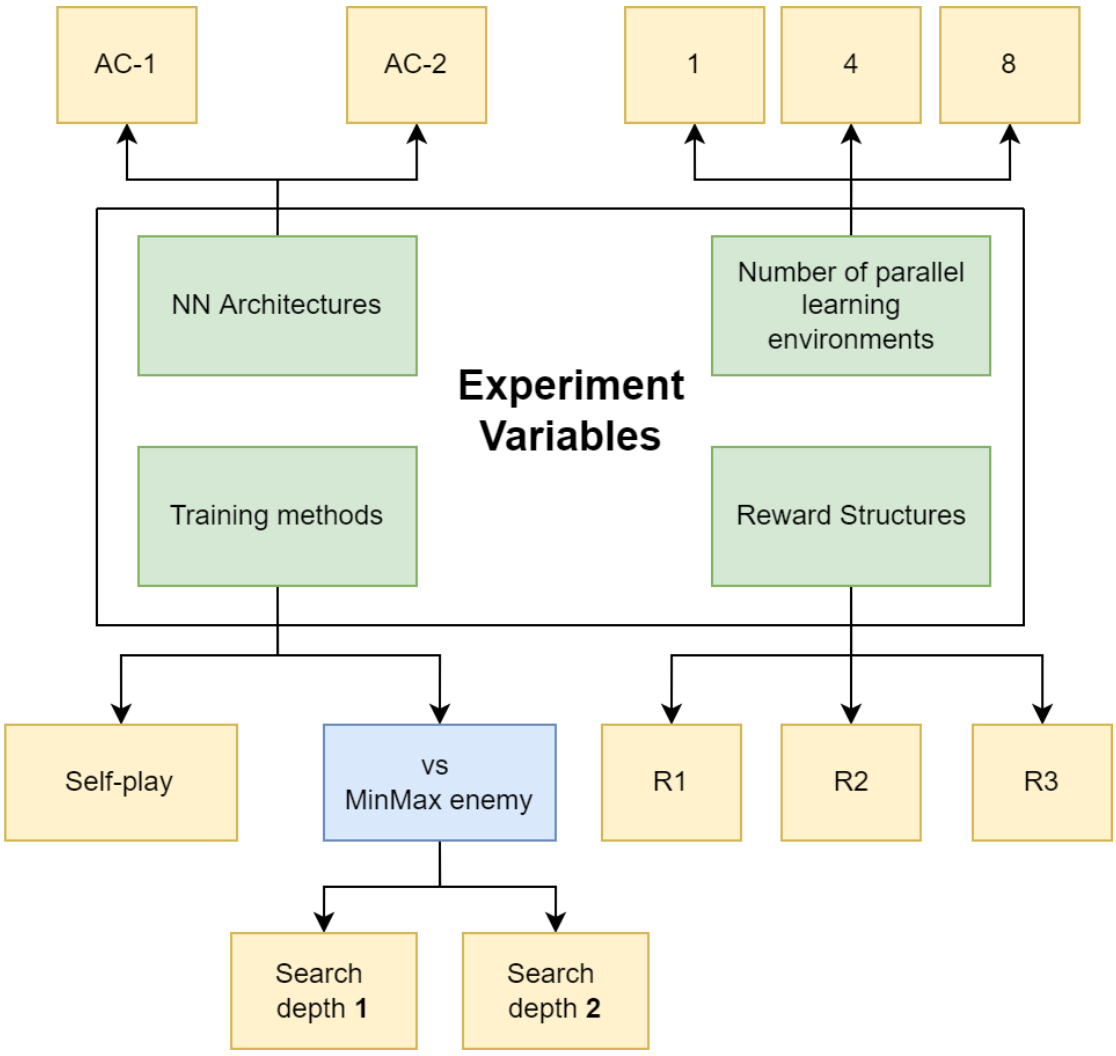

Figure 5.1 Training experiment variables

5.1 Training Setups

The 7 training experiments are separated using training setups that share an episode/match/game length of 100 moves and experiment variables. In each setup, certain environment and training parameters were tweaked while others were kept constant in order to attempt to solve problems encountered in the previous setups. All possible combinations of the variables were not covered in every training setup. The reasoning is covered in section 5.3.

■ Training Setup 1 (TS1)

Trains Action Policy NN with the AC-1 NN architecture, figure 4.6, section 4.4, using policy reward policy R1 as shown in table 5.5 with 1 learning environment. This training setup was used for the following training experiments.

1. **Training setup 1 experiment 1 (TS1AC1Exp1):** SAC and PPO agent training against agent using MinMax searching algorithm with search depth 1
2. **Training setup 1 experiment 2 (TS1AC1Exp2):** SAC and PPO agent training with self-play then tested against MinMax search depth 1

Experiment name	NN architecture	Training method	Reward structure	Number of parallel learning environments
TS1AC1Exp1	AC-1	Vs **MinMax**, search depth **1**	R1	1
TS1AC1Exp2	AC-1	**Self-play**	R1	1

Table 5.1 Training Setup 1

■ Training Setup 2 (TS2)

Trains Action Policy NN using the newer AC-2 NN architecture, figure 4.7, using reward policy R1 as shown in table 5.5 with 1 learning environment. This training setup was used for the following training experiments.

1. **Training setup 2 experiment 1 (TS2AC2Exp1):** SAC and PPO training with AC-2 architecture against agent using MinMax searching algorithm with depth 1
2. **Training setup 2 experiment 2 (TS2AC2Exp2):** SAC and PPO training with AC-2 architecture against MinMax searching algorithm with depth 2

Experiment name	**NN architecture**	Training method	Reward structure	Number of parallel learning environments
TS2AC2Exp1	***AC-2***	Vs **MinMax,** search depth **1**	R1	1
TS2AC2Exp2	***AC-2***	Vs **MinMax**, search depth **2**	R1	1

Table 5.2 Training Setup 2

■ Training Setup 3 (TS3)

Trains an Action Policy NN using the newer AC-2 architecture, figure 4.7, using reward policy R2, as shown in table 5.6. The agents training in this setup used intermediate goal states which is covered in more detail in the reward policy section 5.2. The experiments conducted under this setup used multiple parallel training environments. This training setup was used for the following training experiments:

1. **Training setup 3 experiment 1 (TS3AC2Exp1):** Training agents using intermediate goal states with 4 parallel learning environments and self-play
2. **Training setup 3 experiment 2 (TS3AC2Exp1):** Training agents using intermediate goal states with 8 parallel learning environments and self-play

Experiment name	NN architecture	**Training method**	**Reward structure**	Number of parallel learning environments
TS3AC2Exp1	AC-2	***Self-play***	***R2***	**4**
TS3AC2Exp2	AC-2	***Self-play***	***R2***	**8**

Table 5.3 Training Setup 3

■ Training Setup 4 (TS4)

Trains an Action Policy NN using the newer AC-2 architecture, figure 4.7, using reward policy R3, as shown in table 5.7. A reward generator is used in addition to the rewards given by the environment to give bonus to the agent for finding new and unknown states. The intrinsic reward generation system used is called Curiosity, and it is covered in more detail in the Reward Policies section. This training setup was used for the following training experiments:

1. **Training setup 4 experiment 1 (TS1AC2Exp1):** Training agents with Curiosity rewards with 4 parallel workers and self-play

Experiment	NN architecture	Training method	**Reward structure**	Number of parallel learning environments
TS4AC2Exp1	AC-2	Self-play	***R3***	4

Table 5.4 Training Setup 4

5.2 Reward Policies

This section covers the 3 different reward policies used for training in the 7 training experiments.

■ Reward Policy 1 (R1)

The reward policy R1 used for the training setup 1 and 2 is shown in the following table 5.5. The rewards given range from -16 to +16. R1 is detailed in the points below.

1. The maximum reward of +16 is awarded for winning the game in anyway, which includes getting a small lead in score and then stalling till the maximum number of moves are reached.
2. The minimum reward of -16 is given for losing a game.
3. Additionally, the agent gets a +1 reward for capturing any enemy guti and a -1 for losing a guti.
    - These rewards are both $1/16^{th}$ of the maximum/minimum reward that the agent will get if they win or lose.
    - They were added so that the agent has more frequent real rewards to learn from.
4. A small penalty of −0.2 ($1/5^{th}$ of that for losing a guti) was also given to the agent for taking any action.
    - This is so that agent has incentive to try to minimize the number of moves it takes to end a game.
    - If an agent takes a large number of useless actions, then it can get up to a -20 total cumulative reward.
5. A penalty of -16 is also given for taking illegal actions.
    - The sharp penalty is to heavily discourage the agent from exploring illegal actions as it wastes training and computation time.

The rewards chosen were larger in magnitude to encourage the training agents to learn and converge faster as the larger contrast in magnitude will adjust the action policies more drastically. This was done intentionally to encourage faster convergence, even to sub-optimal maxima, to make R1 more useful for slow training environment.

Reward / Penalty Condition	Reward	Ratio to maximum reward
Reward for wining a match	16	1
Reward per enemy guti captured from enemy	1	1/16th
Reward for drawing a match	0	0
Penalty for losing a match	-16	-1
Penalty per guti lost to enemy	-1	-1/16th
Penalty per legal move	-0.2	1/80th
Penalty per illegal move	-16	-1

Table 5.5 Reward Policy R1

■ Reward Policy 2 Intermediate Goal States (R2)

The reward policy R2 used for the training setup 3 is shown in the following table 5.6. The rewards given range from -1 to +1. R2 is detailed in the points below

1. The reward proportions are the same as the proportions used in R1 table 5.5.
2. The magnitude of the rewards was reduced because
   - We could use multiple training worker agents
   - This allowed much faster training
   - Hence, it was more advantageous to reduce contrast between rewards to allow the agent to explore more and converge more slowly with better results.
3. The winning reward was differentiated between stalling win reward and a real win reward.
   - When the agent wins without capturing all the enemy gutis, it means that the agent probably wasted moves or chose to play defensively after gaining a lead in score. This was termed a stalling win.
   - Stalling wins were given lower rewards (1/8th of that for a real win) to encourage the agent to play more aggressively.
4. Additionally, the agent using reward policy R2 is rewarded for reaching generated **intermediate goal states**.
   - The intermediate goal states are desirables states between the start and finish of a match that we want the agent to reach.

- The intermediate goal states used to reward the agent were generated by an agent using MinMax searching algorithm with a search depth of 3 that generated desirables states from mid and late stages of a Shologuti match.
- The generated states were stored in a dictionary which was then serialized to file to be fetched for use in training.
- When training an agent using reward policy R2, the generated intermediate states are also loaded
- The compressed representation of the states themselves are used as keys in the dictionary to allow for instantaneous access during training.
- If the access attempt is successful (meaning an intermediate goal state has been reached) then the agent gets a bonus reward as shown in table 5.6 and otherwise nothing.

Reward / Penalty Condition	Reward	Ratio to maximum reward
Reward for wining a match after capturing 16 enemy guti	1	1
Reward for wining a match by stalling the game after reaching a higher score than enemy till move limit is reached.	0.125	1/8th
Reward per enemy guti captured from enemy	0.0625	1/16th
Reward for reaching intermediate goal states	**0.0625**	**1/16th**
Reward for drawing a match	0	0
Penalty for losing a match	-1	-1
Penalty per guti lost to enemy	-0.0625	-1/16th
Penalty per legal move	-0.02	-1/50th
Penalty per illegal move	-1	-1

Table 5.6 Reward Policy R2 with Intermediate goal states

■ Reward Policy 3 Curiosity Rewards (R3)

The reward policy R3 used for the training setup 4 is shown in the following table 5.7. The rewards given range from -1 to +2. R3 is detailed in the points below.

1. Mostly the same rewards as R2
2. But with extra rewards generated by Curiosity algorithm instead of the intermediate goal states.
3. Curiosity is an intrinsic reward generation system that trains its own generative NN that tries to predict future states
    - Instead of training on the reward, Curiosity uses the current and future state observations sent by the environment to train its NN in a supervised learning manner,
    - It tries to predict what the next state of the game board will be given a current board state.
    - The higher the prediction error (meaning the state is less visited or unknown), the more reward it will give the agent.
    - Thus, the agent is incentivised to find new unseen states as the Curiosity NN will be unable to accurately predict states it has not been trained for and will subsequently give the agent a bonus besides the normal rewards it gets from the environment.

Reward / Penalty Condition	Reward	Ratio to maximum reward
Reward for wining a match after capturing 16 enemy guti	1	1
Reward for wining a match by stalling the game after reaching a higher score than enemy till move limit is reached.	0.125	1/8th
Reward per enemy guti captured from enemy	0.0625	$1/16^{th}$
Reward generated by curiosity module	**Range (0 to 2)**	**0 to 2**
Reward for drawing a match	0	0
Penalty for losing a match	-1	-1
Penalty per guti lost to enemy	-0.0625	$-1/16^{th}$
Penalty per legal move	-0.002	$-1/50^{th}$
Penalty per illegal move	-1	-1

Table 5.7 Reward Policy R3 with Curiosity rewards

5.3 Results and Analysis

In this section we present the results for different strategies that we employed to train agents using RL algorithms SAC and PPO. We also analyse the results produced and provide explanations for the decisions made for the later training experiments.

- **Results for Training Setup 1**
  - **Observations and analysis from TS1AC1Exp1**

In the first experiment we discuss the results of training two agents with RL algorithms PPO and SAC against an opposing agent using symbolic hard coded MinMax AI. The search depth for the MinMax AI was limited to only a depth of 1. The observations from this experiment are as follows:

1. **Faster convergence of SAC algorithm as compared to PPO:** Figure 5.2 and 5.3 show the results of training agents using SAC and PPO respectively to train against an opposing agent using MinMax searching algorithm. The curve produced by SAC converges faster than that produced by PPO as shown in figure 5.4.
2. **Convergence to a negative cumulative reward:** Both the curves in figure 5.2 and 5.3 reach a maximum cumulative reward of -100 and then stabilize.

Training against agent using MinMax searching algorithm

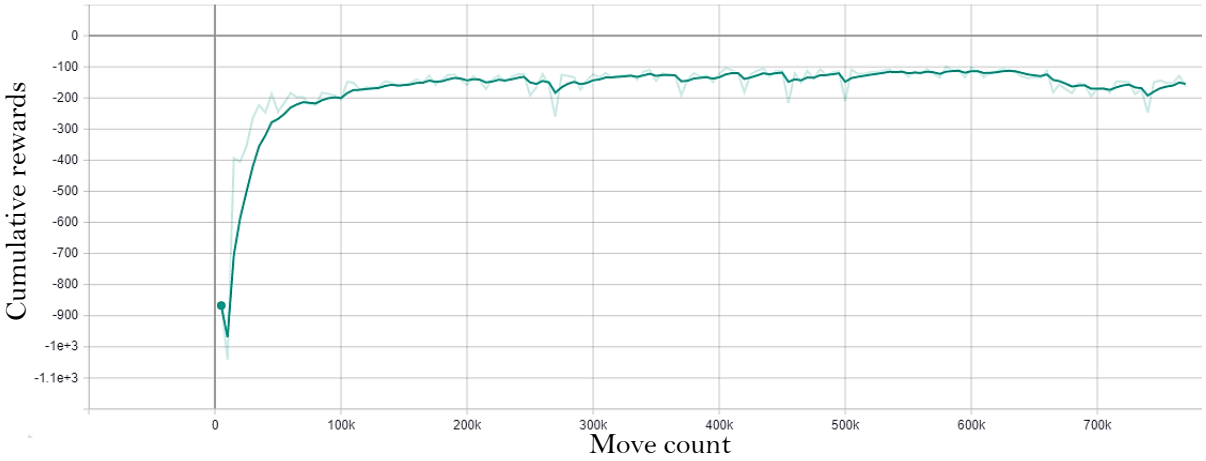

Figure 5.2Training with SAC vs MinMax

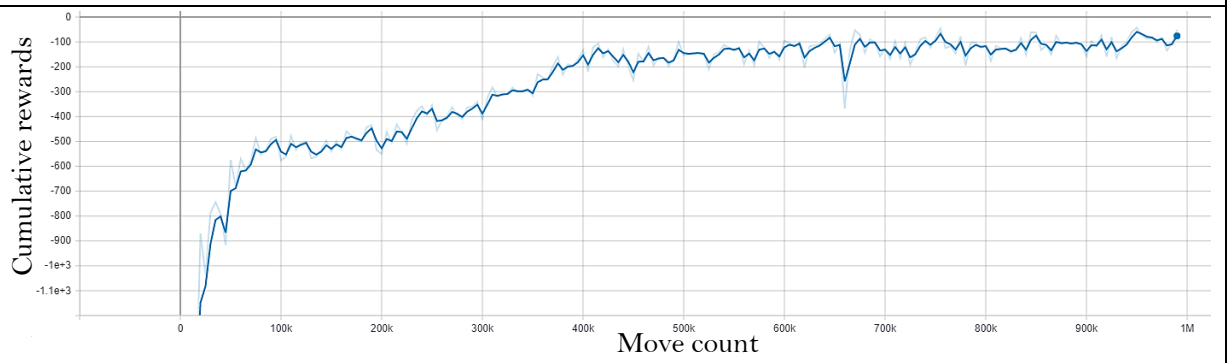

Figure 5.3 Training with PPO vs MinMax

SAC vs PPO trained against MinMax

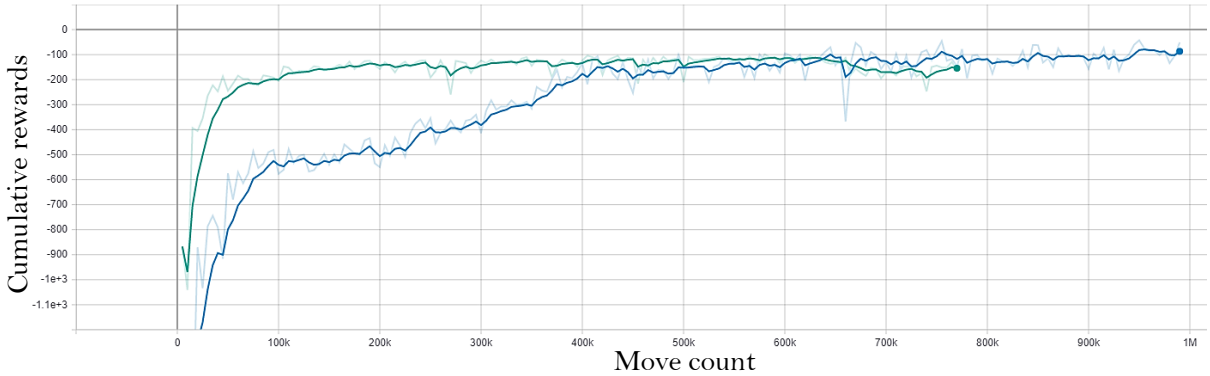

Figure 5.4 SAC and PPO comparison

- **Observations and analysis from TS1AC1Exp2**

In this experiment we discuss the results of training two agents with RL algorithms PPO and SAC using self-play. The observations from this experiment are as follows:

1. **Convergence to a lower negative cumulative reward:** Figure 5.5 and 5.6 show the results after training agents with SAC and PPO using self-play. Both curves eventually reach a maximum of around -20 in cumulative rewards.
2. **Faster convergence of SAC algorithm as compared to PPO:** The SAC curve again converges a bit faster than the PPO curve as shown in figure 5.7.
3. **Poor performance against agents using MinMax search depth 1:** Figure 5.5 shows the SAC curve crashing at around 600 thousand steps and figure 5.6 shows the PPO curve crashing (the cumulative reward falling to very low values) at around 900 thousand steps. The crashes were caused when the training agents were switched from self-play to a number of test matches against an agent using MinMax search with a search depth of 1 to test their performance.

Training with self-play

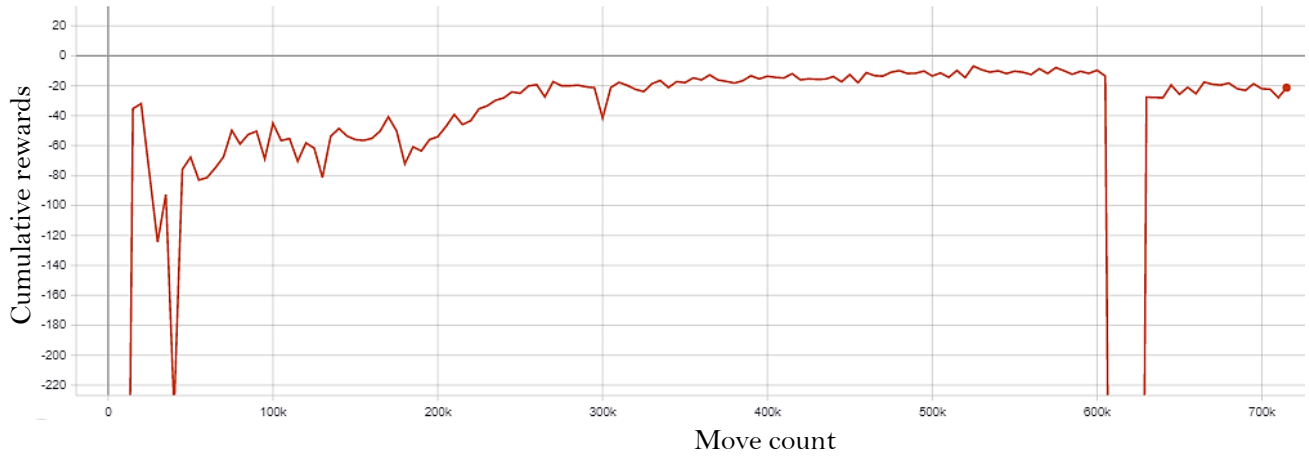

Figure 5.5 Training with SAC using self-play

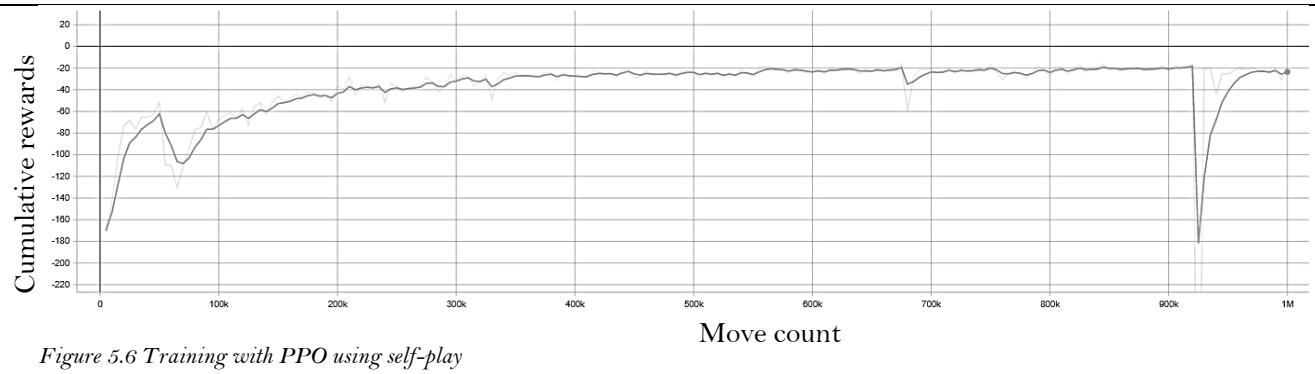

Figure 5.6 Training with PPO using self-play

SAC vs PPO trained with self-play

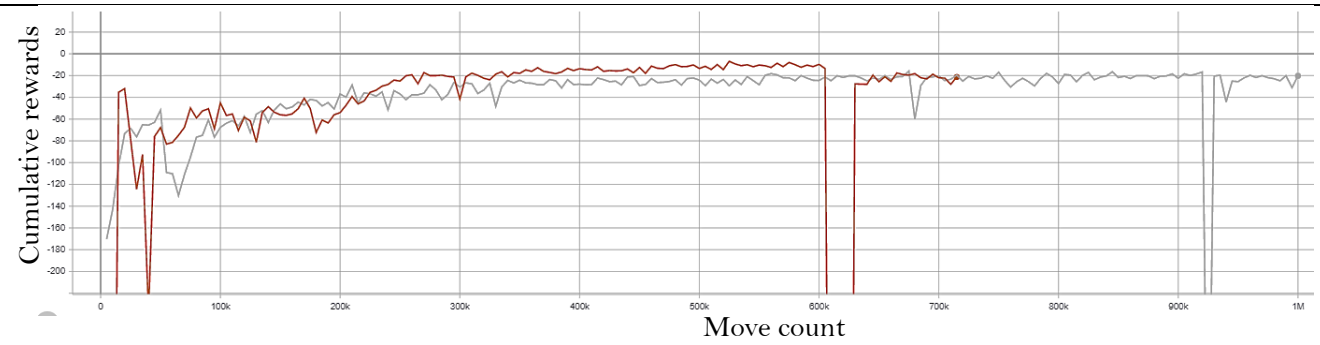

Figure 5.7 SAC and PPO comparison

- **Combined observation and analysis from TS1AC1Exp1 and TS1AC1Exp2**

  From the previous 2 experiments under TS1 and the trained agents they produced; we make the following observations:

  1. **SAC converges faster than PPO:** From the curves in figures 5.3 and 5.6 we can conclude that SAC generally converges faster than PPO, hence making it a more sample efficient sample efficient algorithm.
  2. **Overestimation of rewards by self-play agents:** Figure 5.8 shows that while the self-play trained agents look quite good in terms of cumulative rewards, with a maximum reward of -20 vs the -100 produced by the agents training against MinMax, they do not actually perform that well in practice.
  3. **Agents trained directly against MinMax search depth 1 perform better than self-play trained agents:** Figure 5.9 shows the victory count of an SAC agent trained against MinMax while figure 5.10 shows the victory count of an SAC agent trained with self-play. Both agents were trained for the same number of steps with the same RL algorithm. However, the one trained against MinMax directly has a win ration of 49:1 while the one trained with self-play loses every single match it plays against the same agent using MinMax with search depth of 1.
  4. **High probability of illegal moves:** To investigate why the reward for the agents trained against MinMax was so negative, despite having a high win rate, we ran another training run using the agent trained with PPO in figure 5.3. We also recorded the illegal move count in this training run. Figure 5.12 shows the result of training the agent from figure 5.3 a further 1 million steps. The cumulative reward seemed to have increased to -50 and gotten stuck again. However, taking a look at figure 5.13 reveals that the agent is taking quite a high number of illegal actions. The agent takes an illegal action with the probability of around 0.2 even after 2 million training steps/moves. So, 20% of all moves are illegal even though for every illegal move, the agent gets a sharp additive penalty.
  5. **Trained agents perform poorly against agents using MinMax search depth 2:** Figure 5.11 shows an SAC agent trained against MinMax using search depth of 1 against an agent using a MinMax with search depth of 2. The trained agent loses every single match.

Training with self-play VS training against MinMax

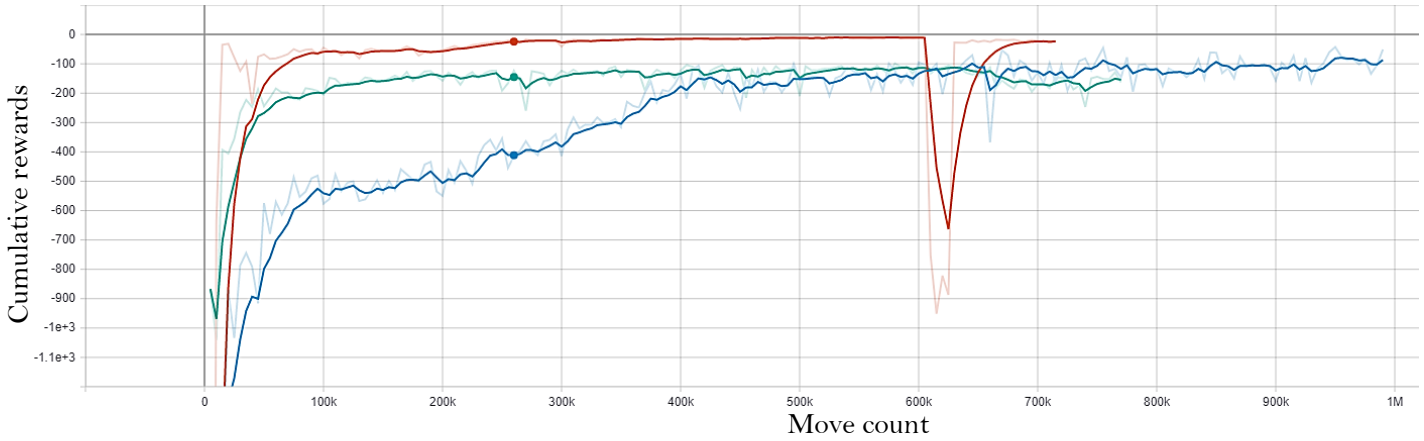

Figure 5.8 Comparison of self-play training vs training against MinMax

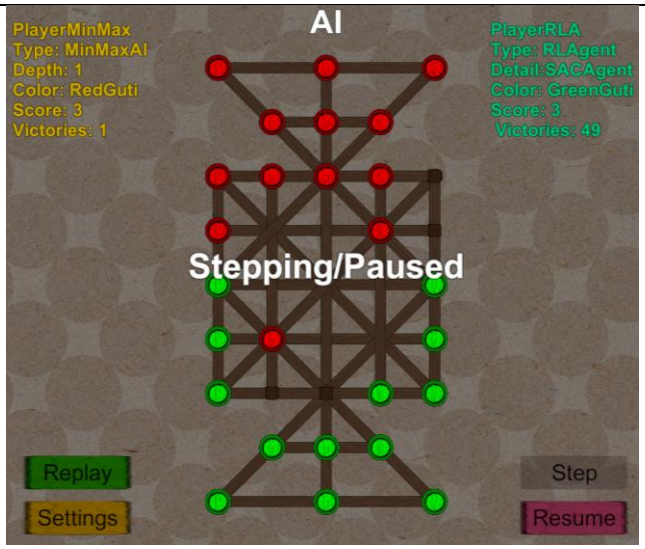

Figure 5.9 Agent using MinMax depth one vs SAC agent trained against MinMax depth one

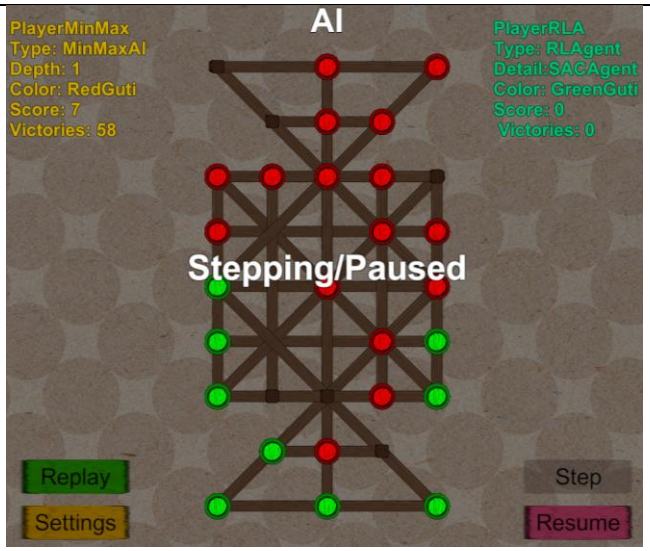

Figure 5.10 Agent using MinMax Depth 1 vs SAC agent trained using self-play

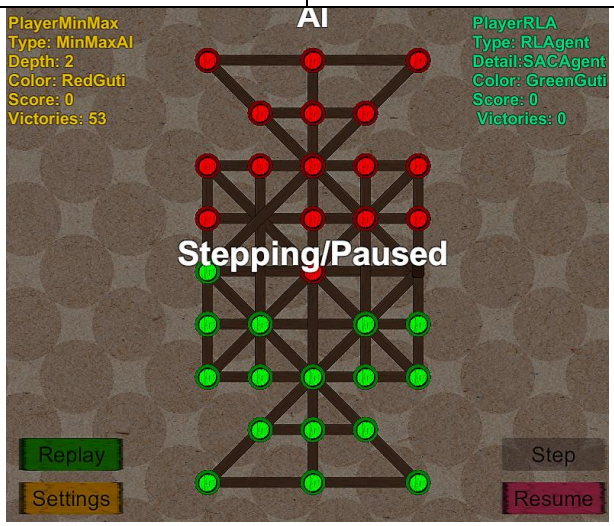

Figure 5.11 Agent using MinMax depth 2 vs SAC agent trained against MinMax depth 1

Illegal move probability

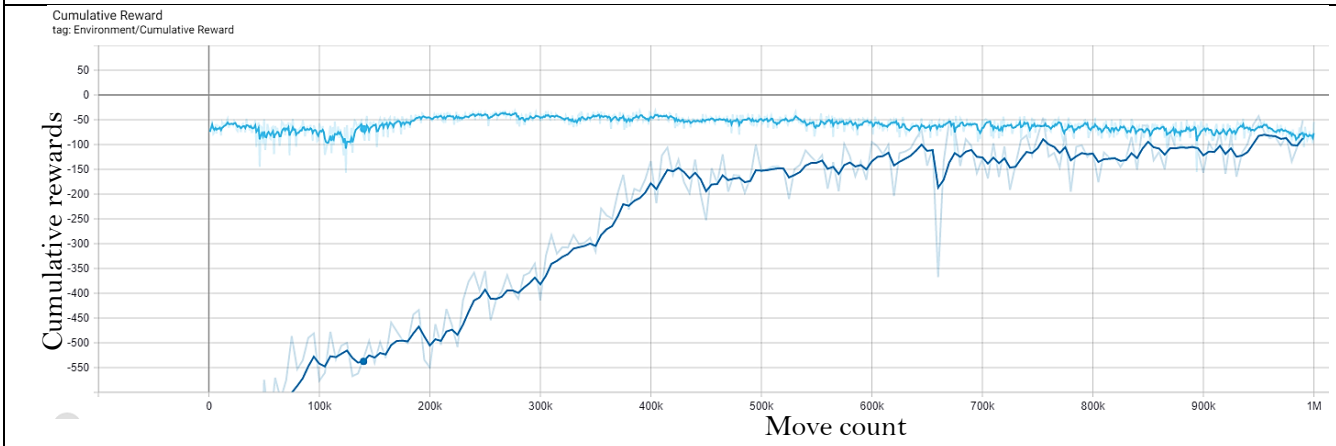

Figure 5.12 Training PPO agent trained against MinMax a further 1 million steps

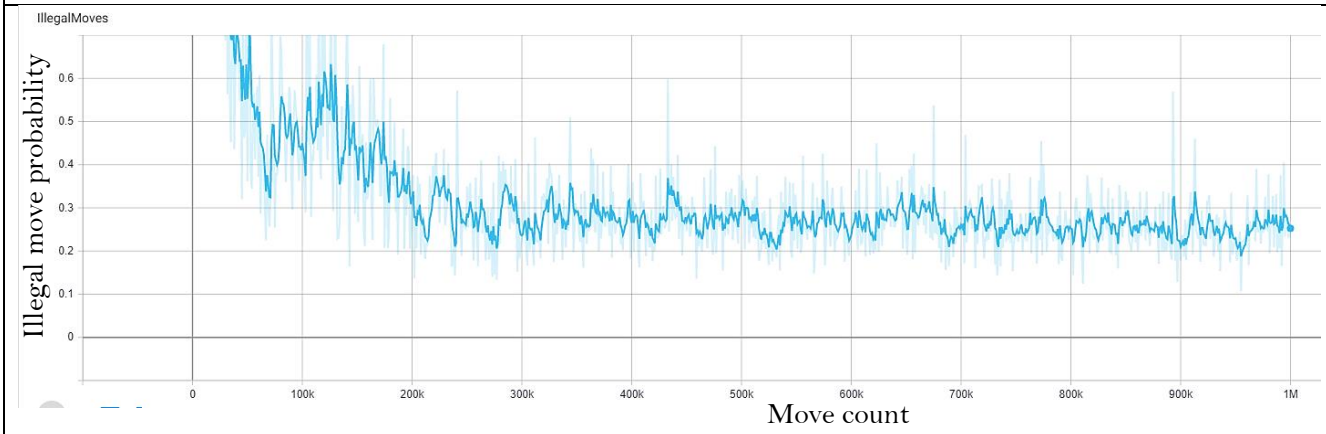

Figure 5.13 Illegal move probability curve

■ Results for Training Setup 2

With the observation of high number of illegal actions being taken by the previous NN architecture, we decided to make changes to the AC-1 NN architecture to limit the number of illegal moves that are possible. In fact, the new NN architecture AC-2, takes zero illegal moves.

▪ Observations and analysis from TS2AC2Exp1

In this experiment we discuss the results of training two agents with RL algorithms PPO and SAC against MinMax search depth 2 with AC-2 NN architecture. The observations from this experiment are as follows:

1. **AC-2 NN architecture is faster to train and produces higher cumulative rewards than AC-1 NN architecture:** Figure 5.14 shows the training run with SAC and PPO against MinMax with search depth of 1 on the new AC-2 architecture. Figure 5.15 compares the training runs with AC-1 architecture and with that of the ones produced by AC-2 architecture. The AC-2 NN learned much faster and produced cumulative rewards near zero instead of the -100 produced by AC-1 NN.

SAC and PPO training with AC-2 architecture

Figure 5.14 SAC and PPO training agents using AC-2 NN architecture

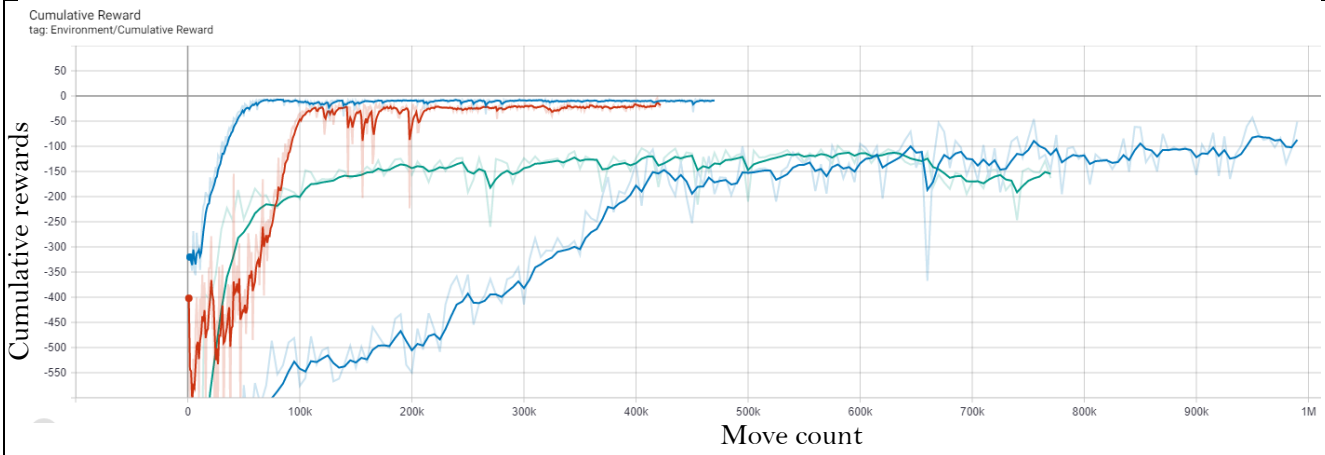

Figure 5.15 AC-1 NN architecture vs AC-2 NN architecture performance

- **Observations and analysis from TS2AC2Exp2**

Further training runs were run using AC-2 NN architecture training against MinMax with search depth 2 directly. We began plotting the win rate instead of the cumulative rewards as they were better estimators of performance.

1. **Increasing draw count with increasing cumulative rewards:** The agent optimized for drawing matches instead of trying to win against MinMax using search depth of 2 on multiple training runs. Figure 5.17 and 5.18 shows the increasing draw count with the increasing cumulative reward collect by an agent training using SAC RL algorithm.
2. **High draw probability vs MinMax search depth 2 even after multiple training runs:** Figure 5.19, 5.20 and 5.21 illustrates how the agent is reducing the victory count of MinMax (figure 5.20) using search depth of 2 and increasing draw counts (figure 5.19) while, its own victories are not increasing (figure 5.21).
3. **Poor performance of trained agents against agent using MinMax search depth 2:** Figure 5.15 shows lacklustre performance of the trained agents against MinMax depth 2 even after multiple training runs.

High Draw Count of RL trained agents Vs MinMax Search Depth 2

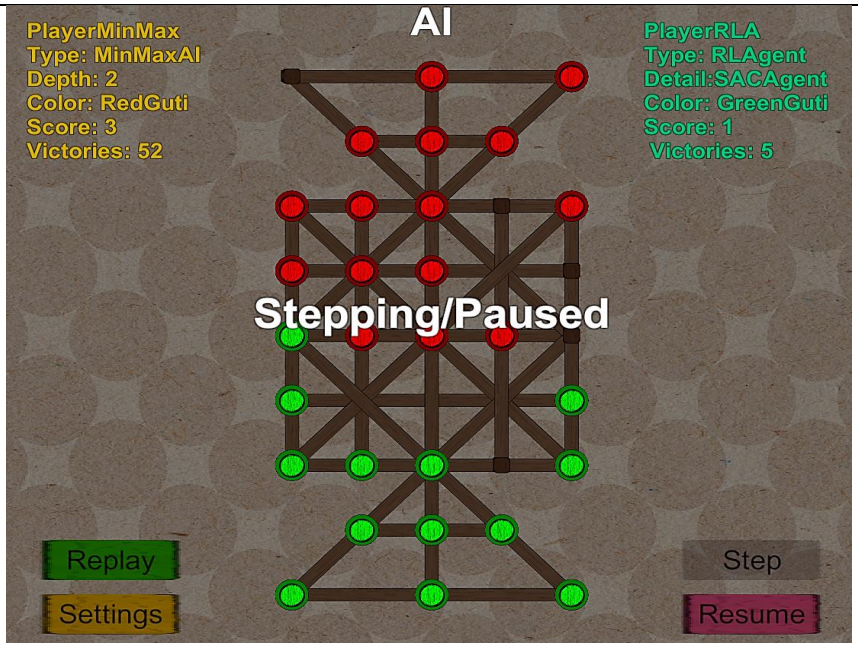

Figure 5.16 SAC agent using AC-2 NN architecture against agent using MinMax with search depth of 2

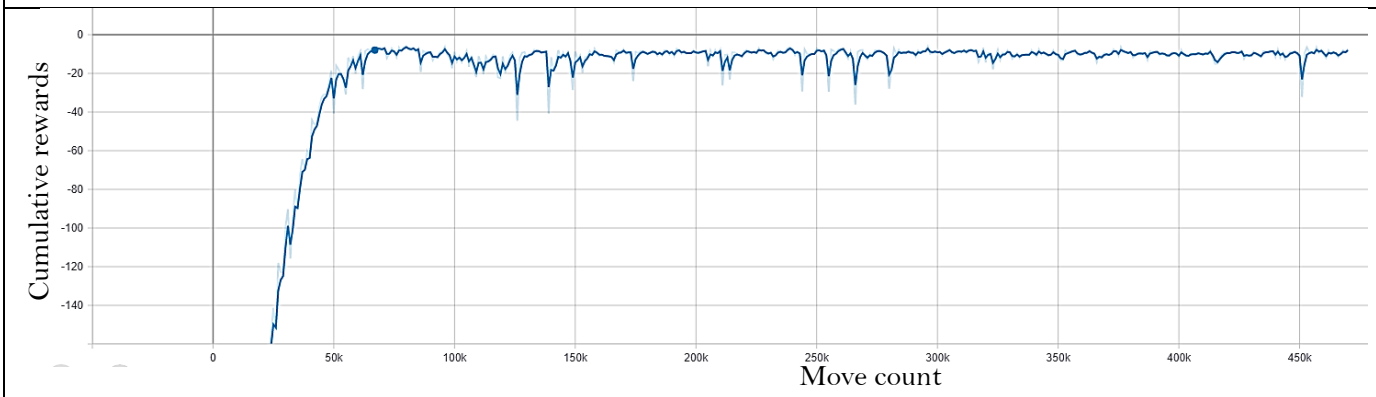

Figure 5.17 Cumulative rewards of agent training with SAC RL algorithm against MinMax with search depth 2

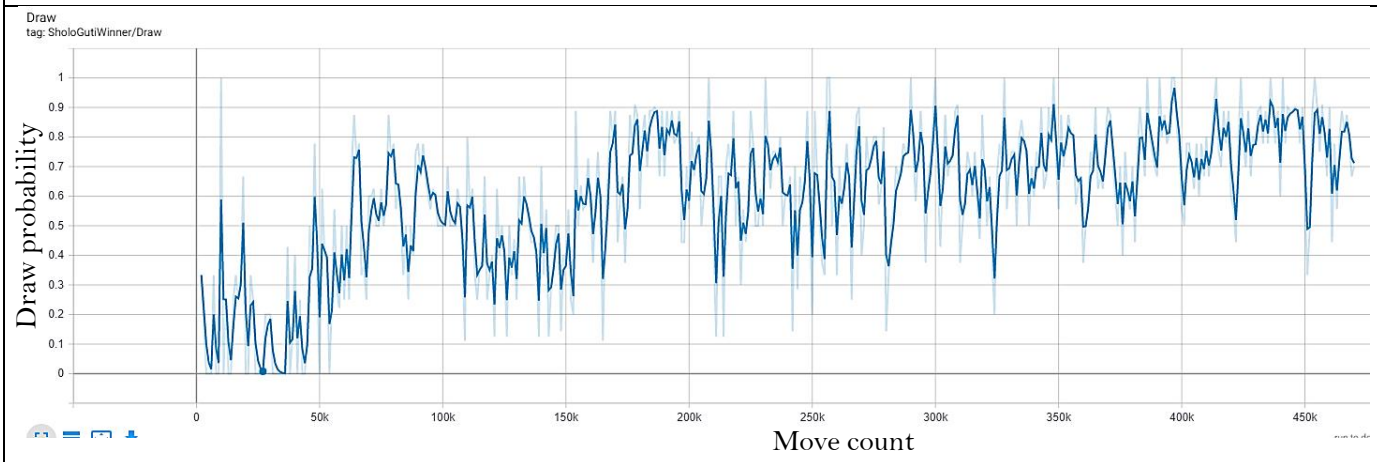

Figure 5.18 Increasing draw probability as SAC agent trains against MinMax with search depth 2

Figure 5.19 Increasing draw probability as PPO agent trains against MinMax with search depth 2

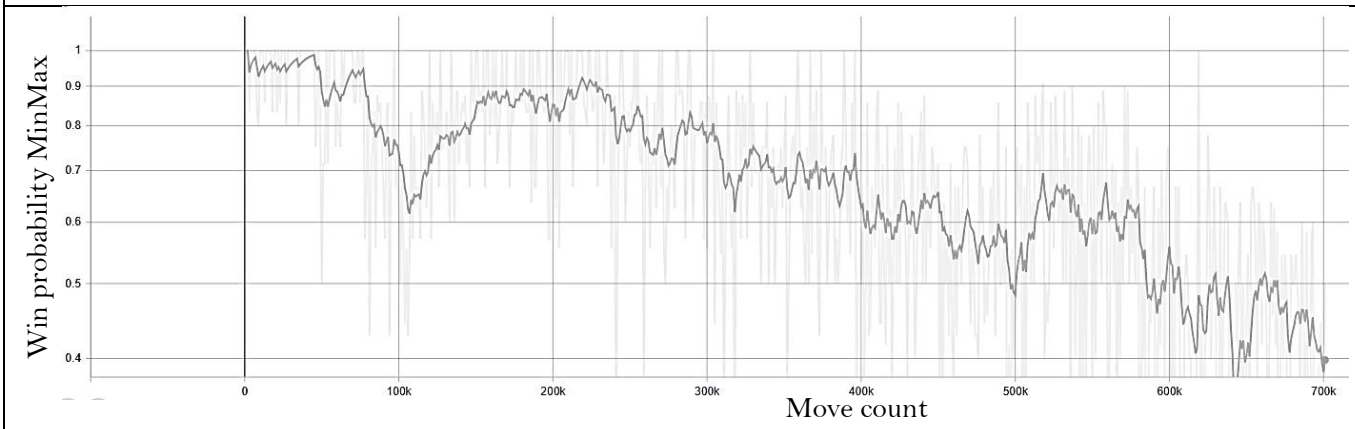

Figure 5.20 Decreasing victory count of agent using MinMax search with depth 2 against agent training with PPO

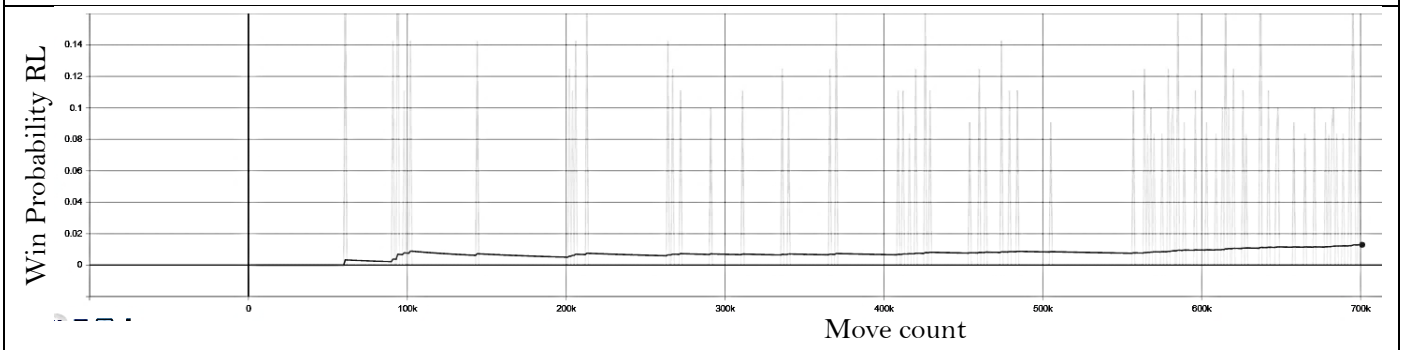

Figure 5.21 Negligible increase in victory count of PPO agent against agent using MinMax with search depth of 2

- **Combined observation and analysis from TS2AC2Exp2 and TS2AC2Exp1**

From the previous 2 experiments under TS1 and the trained agents they produced; we make the following observations:

1. **AC-2 trains faster than AC-1:** Figure 5.15 shows us that AC-2 NN architecture converges to better cumulative rewards faster as it does not waste time expanding illegal moves.
2. **Slow progression of training against MinMax search depth 2:** Figure 5.22, 5.23 and 5.24 show results of training run that show very slow progression while training of over 5 million steps. The draw probability is almost constant while the victory count of the agent training with PPO against the agent with MinMax module is increasing slightly. However, this is at the cost of developing a very defensive strategy where the agent refuses to leave its half of the board.
3. **Trained agents developing highly defensive strategies:** Here we observed that the agents had become increasingly more defensive as they train. The agent even winning against the opposing MinMax search depth 1 using agents by getting a minor lead in score and then stalling the match until the maximum move count was reached. Against MinMax search depth 2, the trained agents chose to draw the game or at least try to lose as few guti as possible during the game. The agents never aggressively tried to win and waited for the opposing player to attack.
4. **Trained agents have lacklustre performance against MinMax search depth 2:** Figure 5.15 shows lacklustre performance of the trained agents against MinMax depth 2 even after multiple training runs.

Further attempts to improve upon this performance did not yield results and the high training times resulted in us attempting to find a better and faster way to train the RL agents.

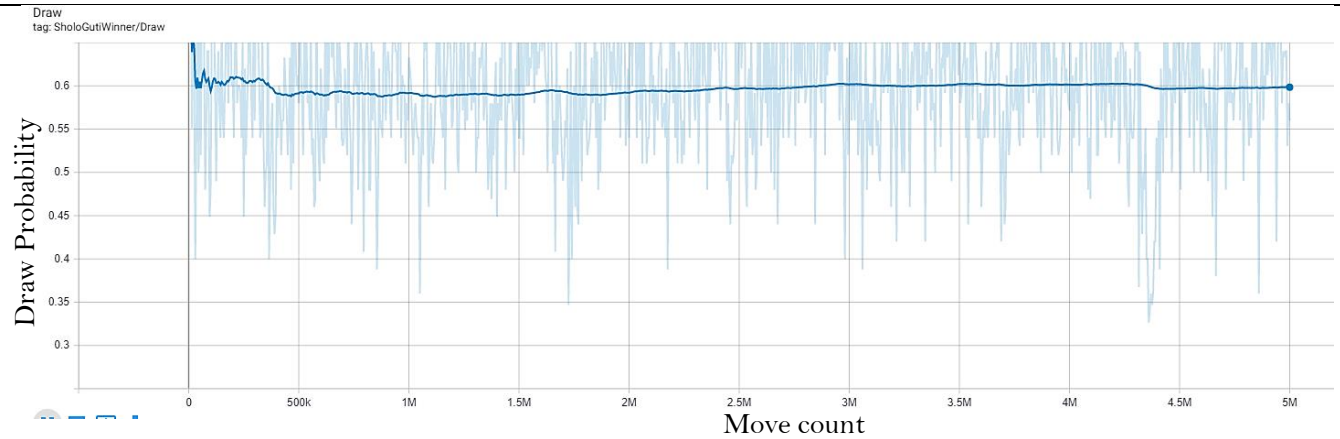

Figure 5.22 Near constant Draw rate after training PPO agent a further 5 million episodes

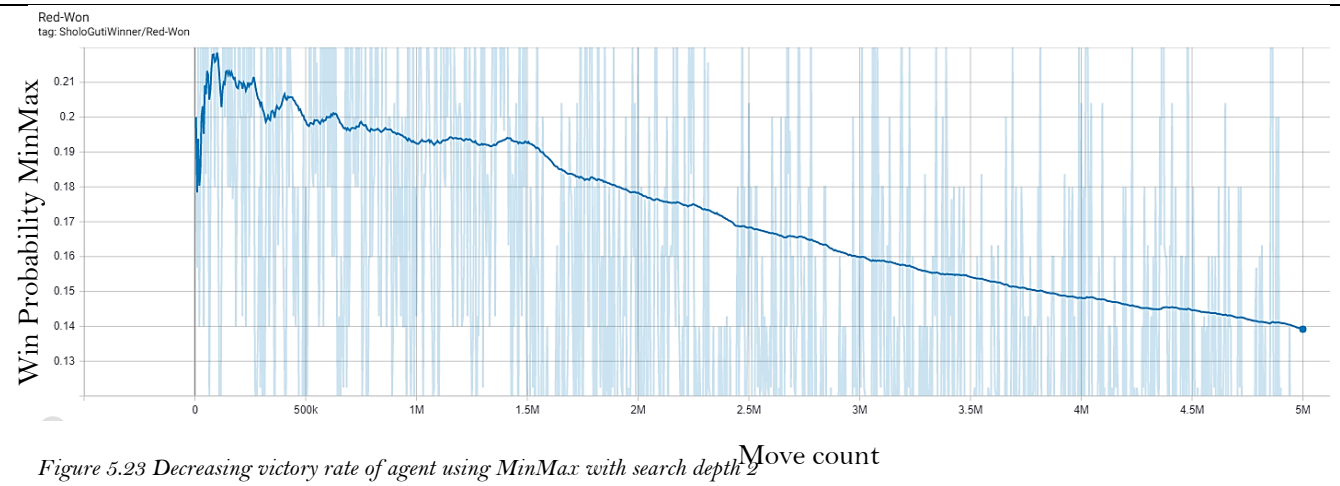

Figure 5.23 Decreasing victory rate of agent using MinMax with search depth 2

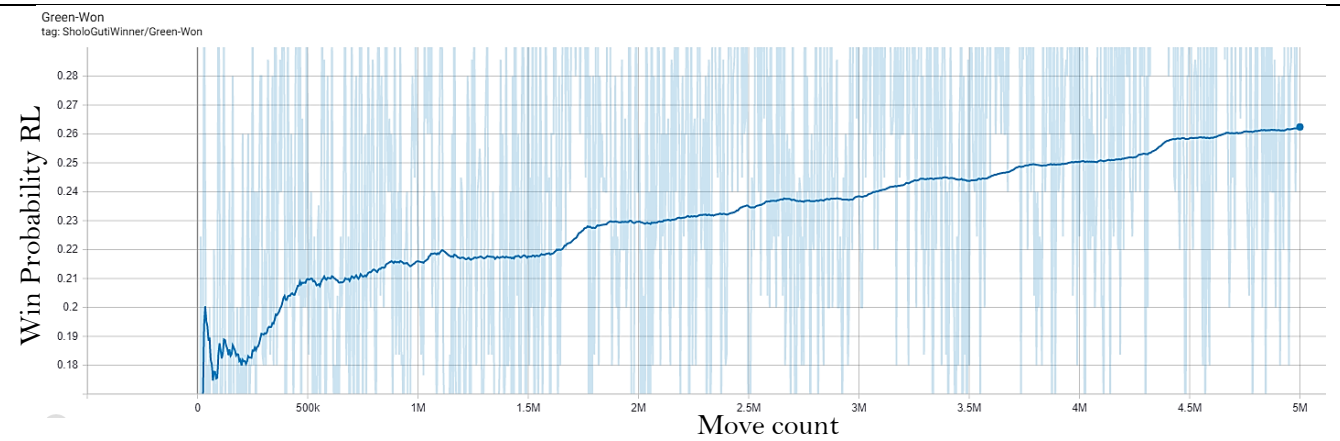

Figure 5.24 Slowly increasing victory count of agent training with PPO

■ **Results training setup 3**

To remedy the problem of high draw probability and overly defensive behaviour seen in the last training setup results, we decided to try a new reward system that we termed **intermediate reward states**. This managed to break the cycle that the agent had got stuck into and it managed to help train agents that defeated MinMax depth 2 decisively. Increasing training worker count played a huge role in the viability and it even resulted in higher victory rate.

- **Observations and analysis from TS3AC2Exp1**

For this experiment the agents were trained using self-play with PPO RL algorithm with 4 parallel environments with each have two deployed agents playing and training against each other. Only 4 agents generate training data at any time as their opponents have to have a fixed action policy (often using older snapshots of the NN) for training stability. All the worker agents train the same neural network.

1. **Positive cumulative rewards:** The red curve in figure 5.25 shows the cumulative rewards for a training run with 5 million steps. The cumulative rewards achieved are finally above zero as the agents are winning decisively instead of stalling the match to victory with a small lead in score or drawing continuously.
2. **High speed of training:** Due to the parallelization of training workers, a high number of 5 million training steps was also achieved much faster.
3. **Decreasing draw rate:** Figure 5.27 shows the decreasing draw rate as the agent trains.
4. **Fluctuating victory rate of training agents:** Figure 5.26 shows the fluctuating victory rate of the training agents. This result is caused by the fact that the agents constantly play against older versions of policies generated by one of the 4 training worker agents. When the fixed policy being played against is suddenly switched with another, the victory rate often drops. This prevents overfitting to a particular strategy and trains a more generalized agent.
5. **Winning decisively against MinMax search depth 2:** Fig 5.28 and 5.29 show the high victory rate achieved by the agent trained using intermediate rewards and parallel workers against agents using MinMax search with search depth 2.

Effects of setting intermediate goal states

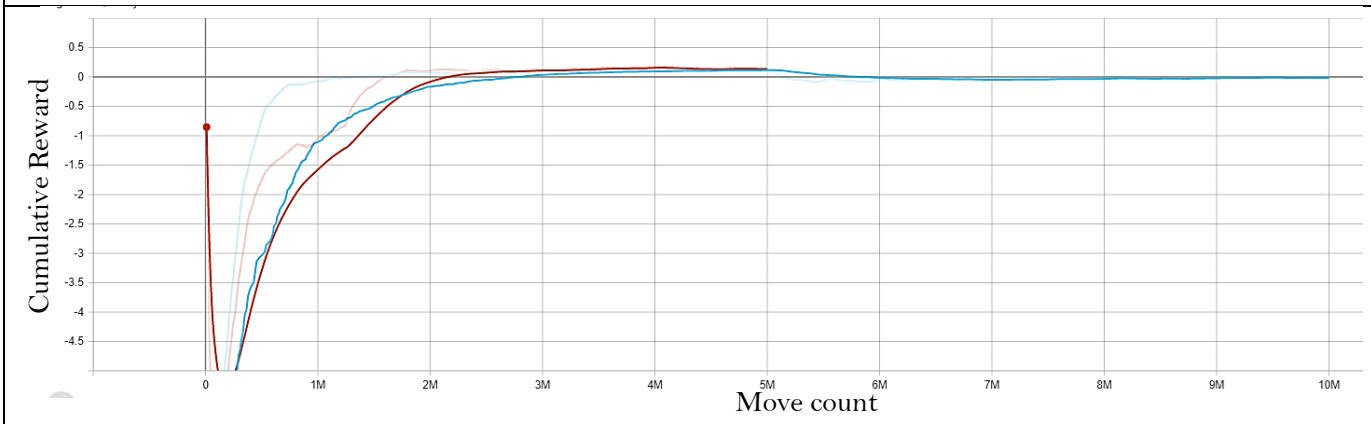

Figure 5.25 Cumulative reward for agents training with intermediate goal states and self-play with 4 training workers

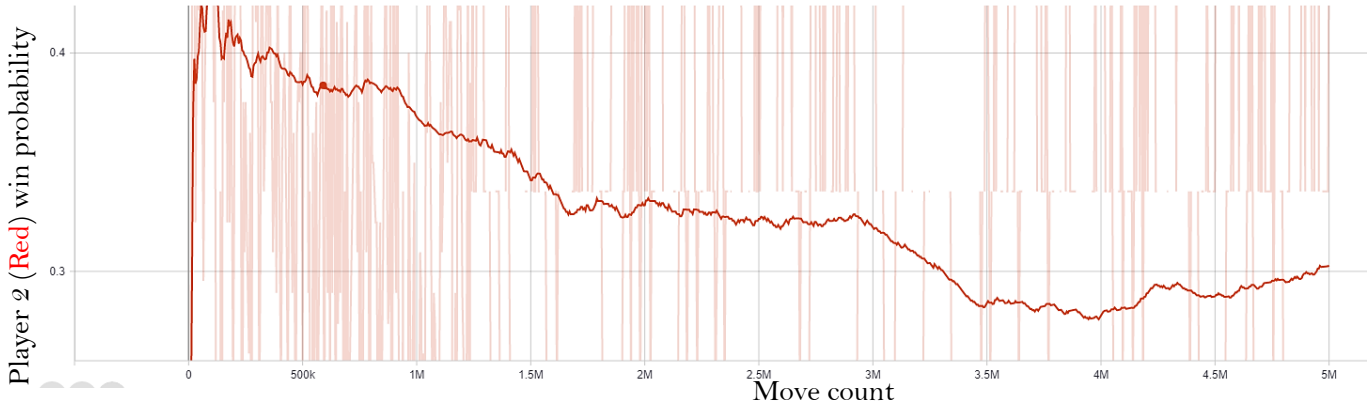

Figure 5.26 Win rate for agents training with intermediate goal states and self-play with 4 training workers

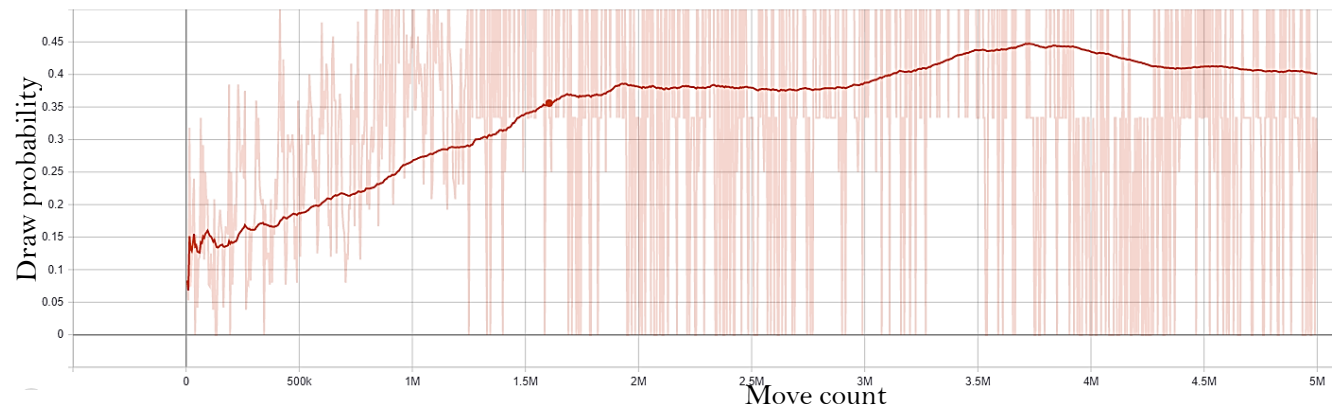

Figure 5.27 Draw rate for agents training with intermediate goal states and self-play with 4 training workers

RL trained agent Vs MinMax agent performance

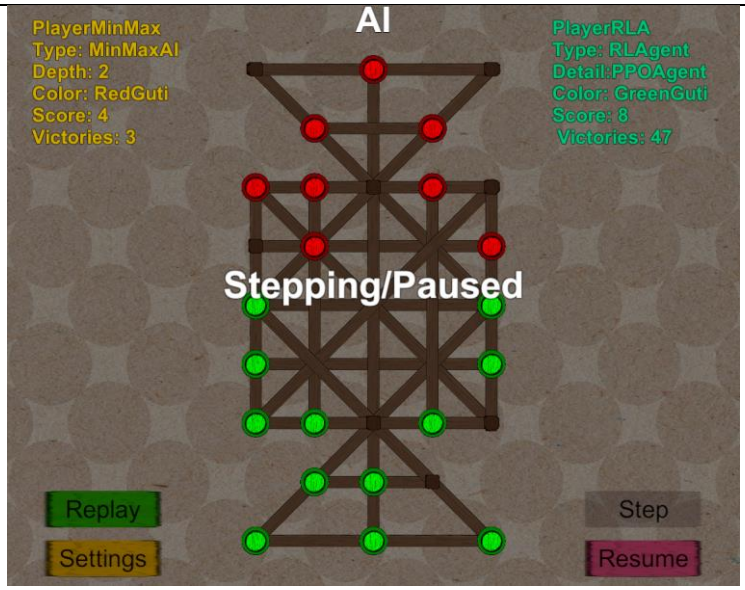

Figure 5.28 Win rate for agents trained with intermediate goal states and self-play with 4 training workers vs MinMax with search depth of 2

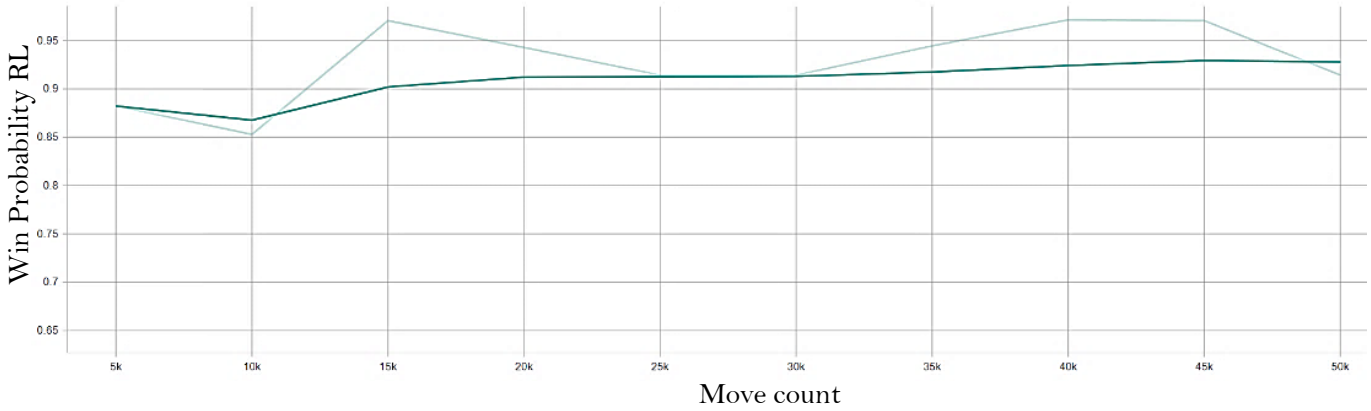

Figure 5.29 Win rate for agents trained with intermediate goal states and self-play with 4 training workers vs MinMax with search depth 2

- **Observations and analysis from TS3AC2Exp2**

For experiment 2 of TS3 nothing was changed except the number of parallel environments being used for training. A total of 8 parallel environment were used.

1. **Lower draw probability with increased parallel learning environments:** Figure 5.30 shows that increasing training workers reduced draw probability noticeably.
2. **Increased victory count for both players during self-play training:** Figure 5.31 and 5.32 shows how the victory counts for agents playing on both teams increased, meaning that they learned more useful information.
3. **Similar Cumulative rewards:** Figure 5.30 shows similar cumulative curves of the two experiments using intermediate rewards states.
4. **Higher win rate against MinMax search depth 2:** Figure 5.33 and 5.34 shows the even higher victory rate achieved by the agent trained with intermediate reward states with self-play and 8 parallel learning environments

Effects of increasing training workers

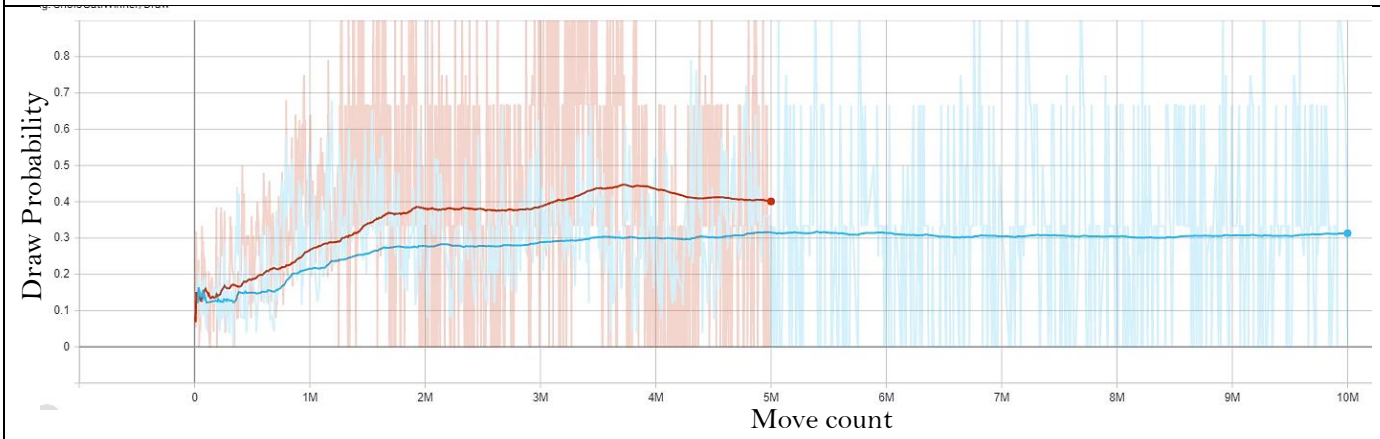

*Figure 5.30 Draw rate for agents training with intermediate goal states and self-play with **8** training workers*

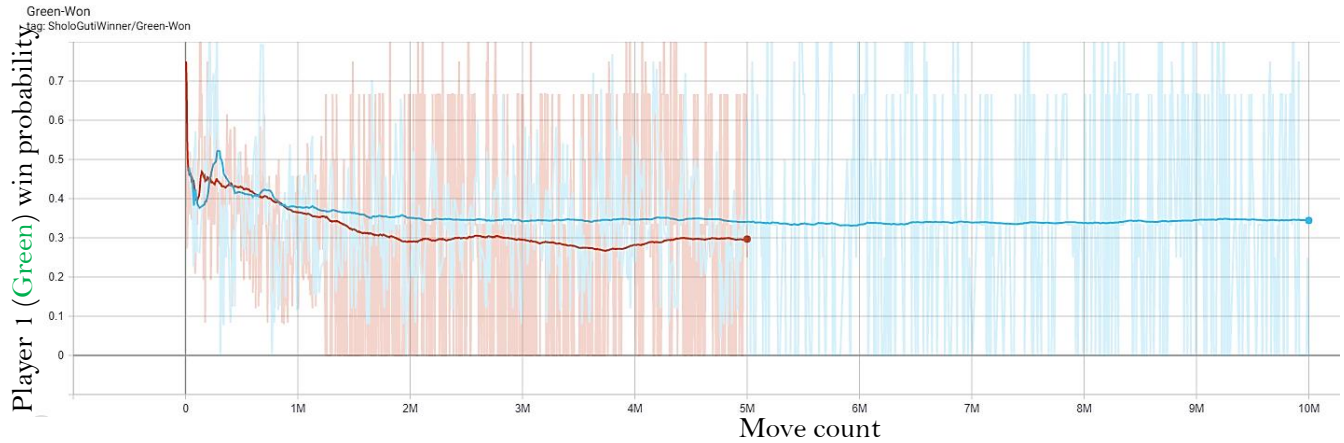

*Figure 5.31 Win rate for agents playing with green guti, training with intermediate goal states and self-play with **8** training workers*

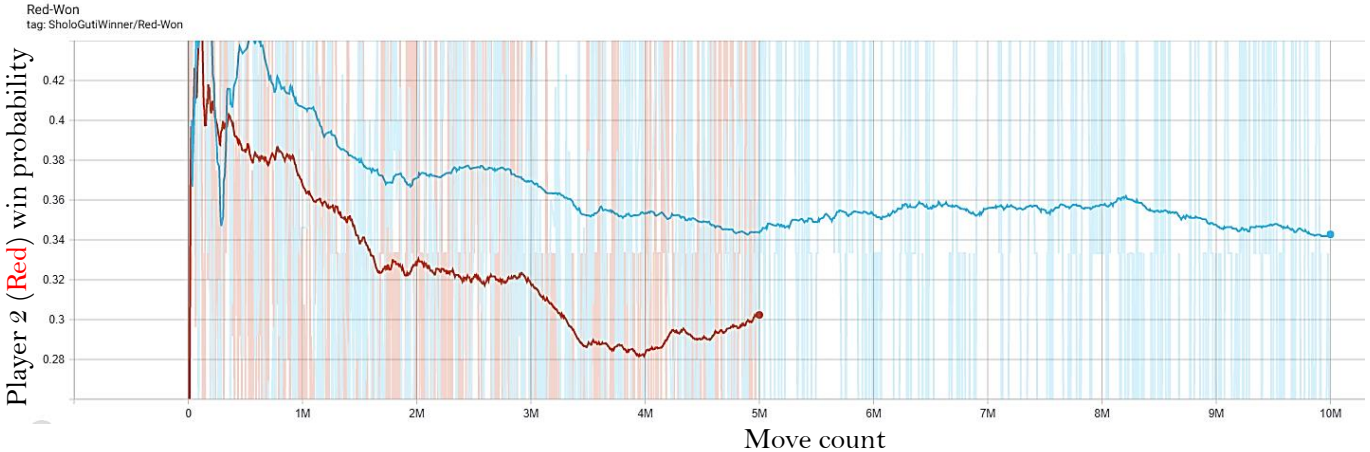

*Figure 5.32 Win rate for agents playing with red guti, training with intermediate goal states and self-play with **8** training workers*

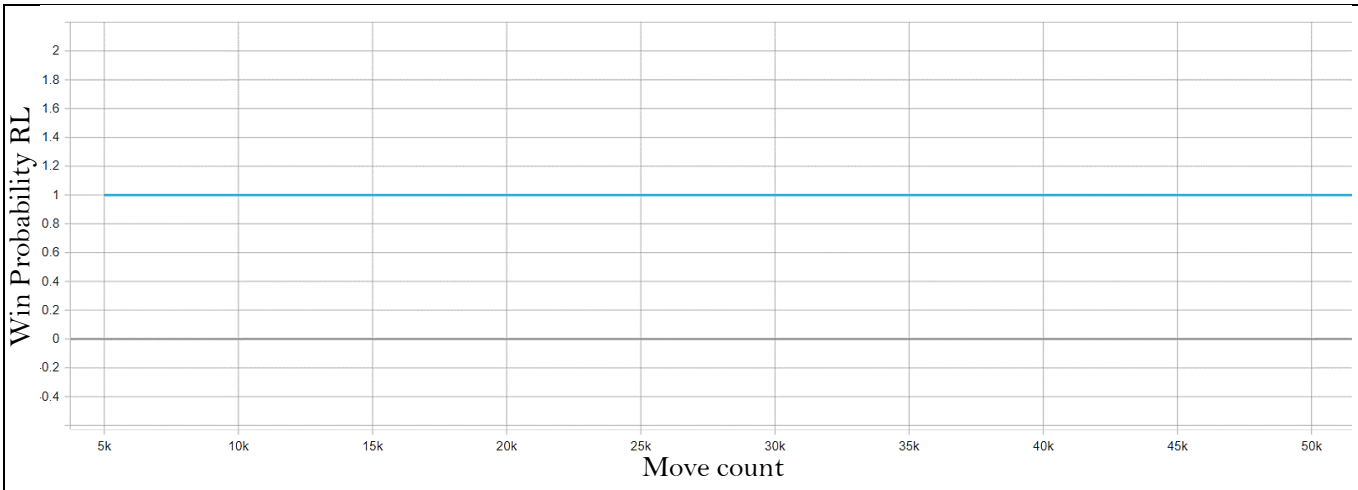

*Figure 5.33 Win rate for agents training with intermediate goal states and self-play with **8** training workers vs MinMax with search depth of 2*

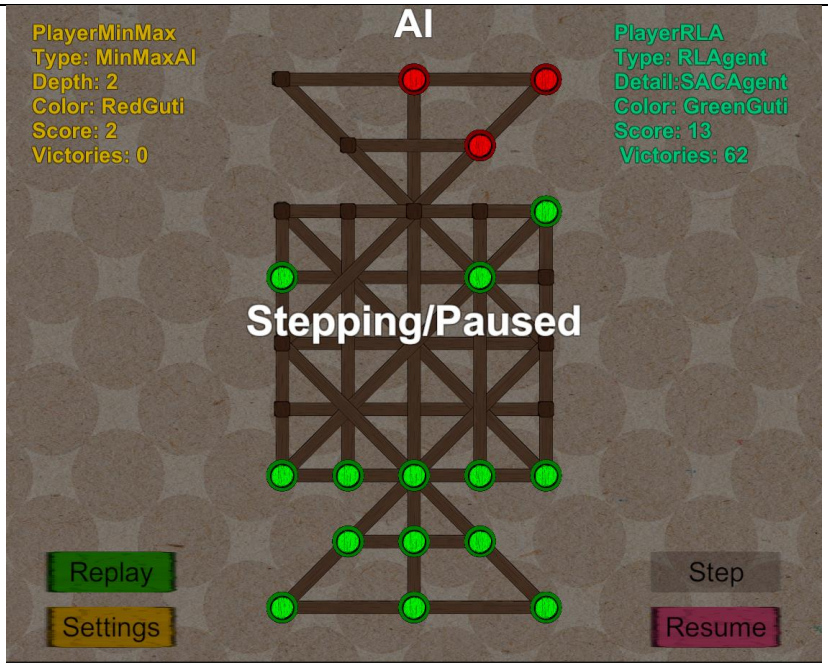

*Figure 5.34 Win rate for agents trained with intermediate goal states and self-play with **8** training workers vs MinMax with search depth of 2*

- **Combined observation and analysis from TS3AC1Exp1 and TS3AC1Exp2**

From the experiments conducted with this training setup TS3, we can conclude that:

1. **We can also conclude that intermediate goal states reward system is a viable solution for breaking agents out of local optima.**
2. **We can also conclude that RL algorithms show notably better performance with the increase of parallel training workers.**
3. **Self-play is a viable training method when multiple parallel learning environments can be used.**

■ **Results training setup 4**

We also tried to reward the agent to find new states and break it out of a defensive drawing strategy using a reward generation system known as Curiosity. However, training with Curiosity lead to general instability. With Curiosity we did however manage to find a bug in the code or really a gap in the rules for Shologuti

- **Observations and analysis from TS4AC2Exp1**

In this training experiment we trained an agent using PPO with self-play and reward structure R3 which includes Curiosity generated rewards.

1. **Very good at finding rare states:** Fig 5.37, 5.38 and 5.39 shows checkmate states, that were discovered with the help of curiosity rewards. The checkmate state is so rare, that no official rules found for the game Shologuti had a say on what should be done in a situation like this. Hence, when the checkmate state was encountered the simulator simply did not know what to do as the Rule system did not account for such a state (this bug was later patched by simply ending the match and declaring the player with more gutis the winner). This shows how under researched Shologuti truly is and it also shows how good curiosity is at finding rare states and bugs.
2. **Highly unstable:** Figure 5.35 shows an instance of failure while training with Curiosity where the curve flattens as the environment has stalled and is sending the same results continuously. The curve in figure 5.36 also shows general instability during training with Curiosity as the agent crashed the environment a few times and corrupted the logs files stored for tensorboard.

Hence, while curiosity may not be good to train board game playing agents, it can certainly be quite useful for testing games and software.

Effects of curiosity-based rewards

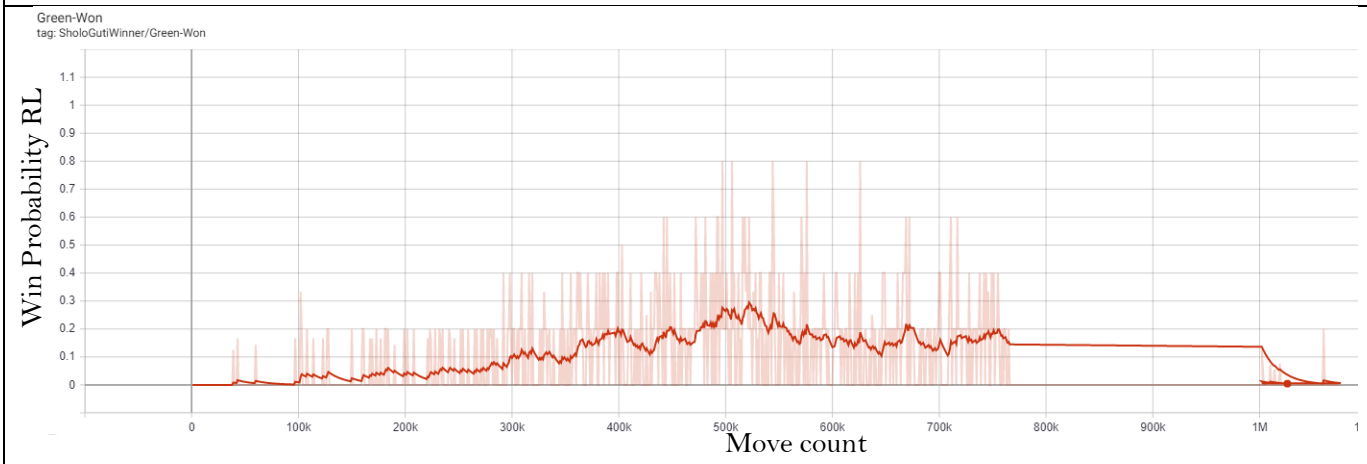

Figure 5.35 Agents training with PPO and Curiosity

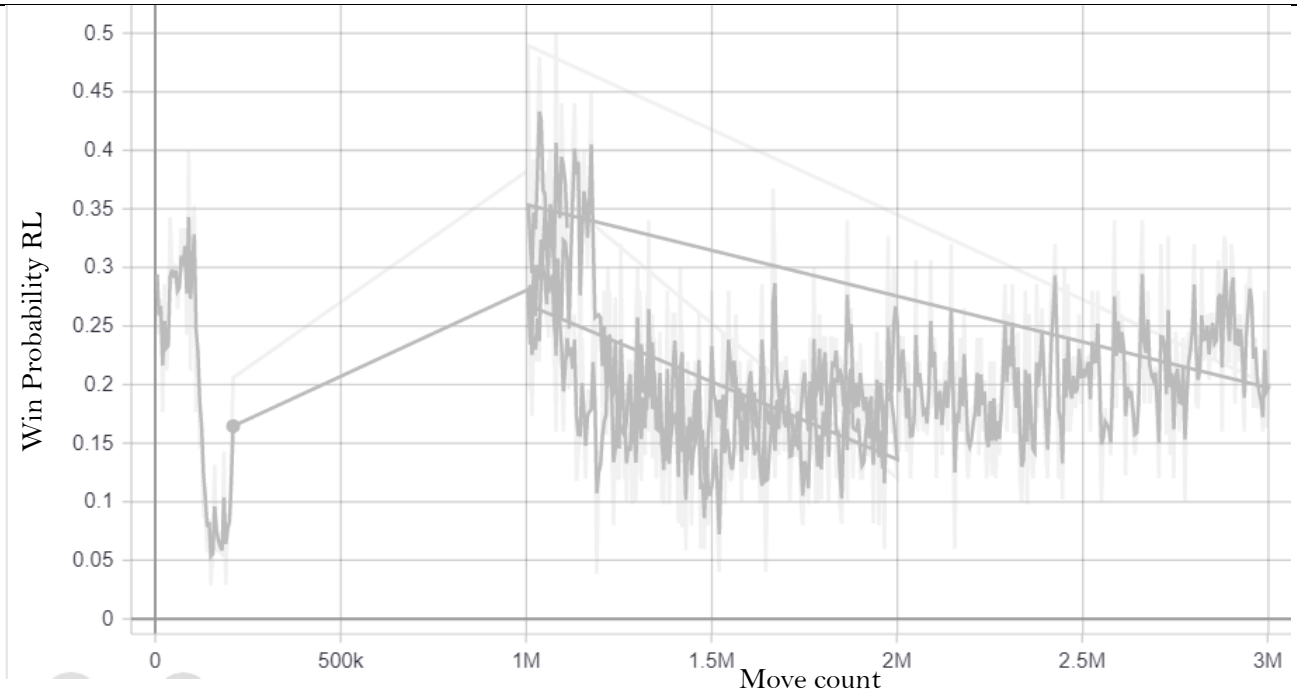

Figure 5.36 Agents training with PPO and Curiosity corrupting tensorboard log files

Checkmate States

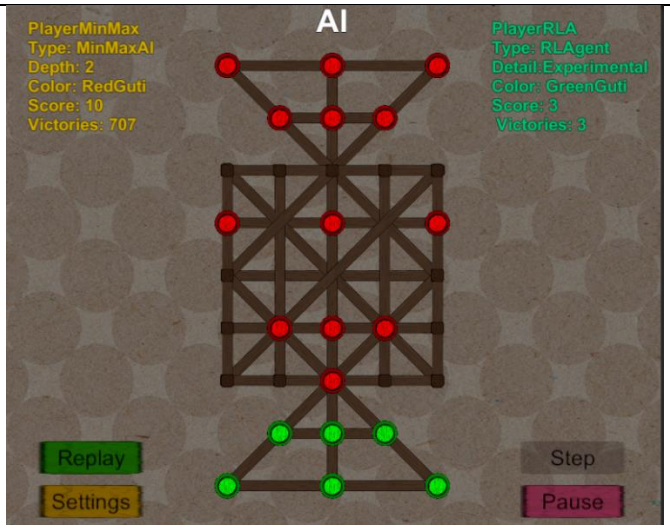

Figure 5.37 Checkmate 1 Green has no moves

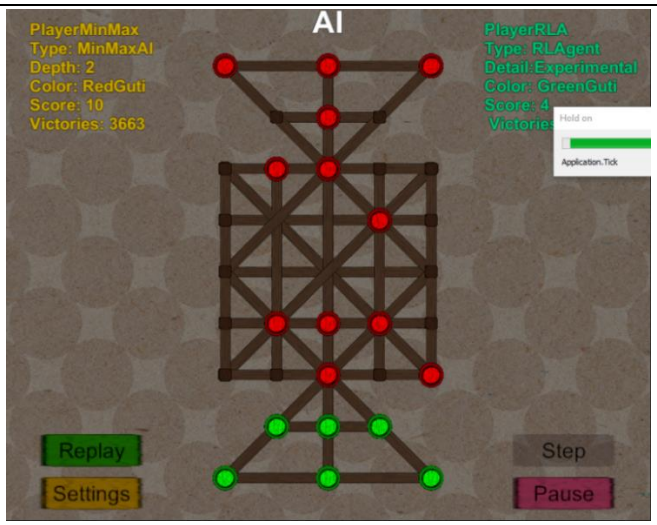

Figure 5.38 Checkmate 2 Green has no moves

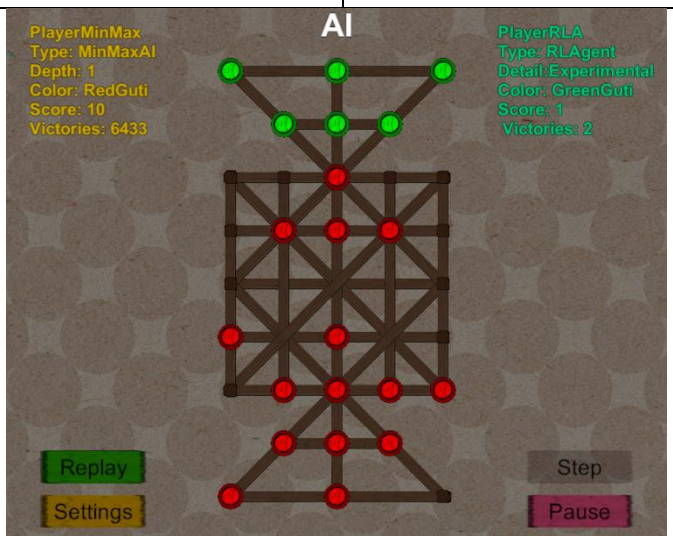

Figure 5.39 Checkmate 3 Green has no moves

5.4 Summary of Results and Analysis

■ Conclusions from Training setup 1

From training setup 1 we concluded that:

1. SAC was more sample efficient that PPO while training.
2. Agents trained with self-play with only 1 active learning environment tend to overestimate their potential rewards and perform notably worse against MinMax using agents when compared to RL agents trained against MinMax directly.
3. AC-1 NN architecture allowed too many illegal moves to be produced which was slowing down training process and producing misleading cumulative reward curves.

■ Conclusions from Training setup 2

From training setup 2 we concluded that:

1. AC-2 NN architecture trained faster and better than AC-1 as it did not waste time and computation power expanding illegal moves.
2. We also concluded that training RL agents directly against MinMax with search depth 2 searching agents would most likely always produce extremely defensive agents that have with high draw probability.

■ Conclusions from Training setup 3

From training setup 3 we concluded that:

1. Self-play is a viable training method for when more parallel learning environments can be used.
2. RL algorithms train faster and perform better when training with more parallel learning environments.
3. Intermediate goal state reward system is a viable system to break RL agents out of local optima.

■ Conclusions from Training setup 4

From training setup 4 we learned that:

1. Curiosity reward generation system was highly unstable in the Shologuti environment, but it is very good at finding rare states.
2. It found a gap in the rules for Shologuti game itself by discovering a checkmate state that the official rules do not account for. This reveals how under research Shologuti truly is.

■ **Overall best training strategy**

Overall, from the 7 training experiments conducted, we concluded that in training balanced and skilled RL agents, the following proved to be the best strategy:

1. Intermediate goal states reward system
2. More parallel learning environments
3. Self-play training method with more than 2 parallel learning environments
4. Improved NN architecture that helps in cutting training times.

6 Conclusion

6.1 Summary

In this research we have looked at the state-of-the-art algorithms that are currently popular in the field of Reinforcement Learning. We have outlined how Open-AI Gym provided a large collection of environments that are available for testing and benchmarking reinforcement learning algorithms using a very intuitive and easy to understand python interface. We have studied other environment and libraries of this type that had come before OpenAI Gym and also more recently released libraries that all seek to keep their interface similar to that of widely adopted OpenAI Gym.

We looked at the state of research on Board games that are popular in Bangladesh like Carrom, Ludo and Shologuti. We concluded that except for Ludo, the other board games were still overlooked, especially Shologuti.

In order to enable our own research and future research on AI in the game of Shologuti, we created a robust reinforcement learning environment for Shologuti using Unity3D and ML-Agents that has been built and tested for the Web and Windows platforms. The environment can be exported to other platforms with some testing and tuning. We implemented MinMax Algorithm to serve as the symbolic Artificial Intelligence in the game and to be an opponent for Human players and reinforcement learning agents. We implemented multiple reinforcement learning agents using the ML-Agents toolkit. We have provided trained agents trained using state of the art SAC and PPO RL algorithms as part of the environment. The learning environment is customizable using the GUI Settings panel.

The Shologuti environment also has custom graphics and animations and is a fully playable game that runs on the Web and windows, (can be exported to mobile platforms as well) with user-friendly interfaces and feedback systems. The game allows humans to play against pre trained agents trained with RL.

We also used the Python API provided by the ML-Agents toolkit to connect our game environment to a custom python environment. We have also created a wrapper around python API provided by ML-Agents that connects to a state evaluation-based agent in the environment that functions like TD learning (*Tesauro, Gerald (March 1995)*, n.d.).

We have trained, benchmarked and analysed SAC and PPO algorithms using implementation provided by Unity ML-Agents and have also tested and found some training policies that work well for training board game agents.

We also found an effective reward system for breaking RL agents out of cycles that we termed Intermediate Goal States.

6.2 Future Work

We wish to create a detailed documentation of the environment we created so that anyone in the future can extend it capabilities easily if they wish to. We also wish to add different board layouts and other games like Shologuti that are popular in our neighbouring countries of India, Sri Lanka, etc. We also wish to eventually add Ludo and Carrom environments to our library to provide a comprehensive collection of all popular board game in Bangladesh to enable and springboard for further Artificial Intelligence Research on these games and contribute to our culture.

We also wish to implement and benchmark more RL algorithms in the Shologuti environment like Alpha-Zero and also design and build our own custom RL algorithm making use of Graph Neural Networks (GNNs) to compress the observation domain.

We wish to expand on our intermediate goal state rewards system and improve upon the system and further investigate what can be done. One future possibility is to use a trained generative model to generate goal states.

We also wish to try imitation learning approaches like Generative Adversarial Imitation Learning (GAIL) to incorporate human knowledge into the RL agents. This was never considered during the initial project as getting human data for an under researched game would have been tedious. Now that we have a fully playable game with custom graphics and animations, humans can generate labelled data by simply playing our game once we finish implementing and testing the feature for storing the move histories of games.

References

[1] Calculating State Space complexity of board games https://www.pipmodern.com/post/complexity-state-space-gametree#:~:text=So%20to%20recap%2C%20tic%2Dtac,43%2C%20and%20Go%20is%20172.&text=Each%20path%20consists%20of%20multiple,larger%20than%20state%2Dspace%20complexity.

[2] Wikipedia Rules Shologuti https://en.wikipedia.org/wiki/Sixteen_Soldiers

[3] Wikipedia Known State Space Complexity of game https://en.wikipedia.org/wiki/Game_complexity

*Tesauro, Gerald (March 1995). "Temporal Difference Learning and TD-Gammon". Communications of the ACM. **38** (3). doi:10.1145/203330.203343. Retrieved Nov 1, 2013.*

Alhajry, M., Alvi, F., & Ahmed, M. (2012). TD(λ) and Q-learning based Ludo players. *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, 83–90. https://doi.org/10.1109/CIG.2012.6374142

Alvi, F., & Ahmed, M. (2011). Complexity analysis and playing strategies for Ludo and its variant race games. *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, 141–148. https://doi.org/10.1109/CIG.2011.6031999

Arulkumaran, K., Cully, A., & Togelius, J. (2019). AlphaStar: An evolutionary computation perspective. *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 314–315. https://doi.org/10.1145/3319619.3321894

Baby, N., & Goswami, B. (2019). *Implementing Artificial Intelligence Agent Within Connect 4 Using Unity3d and Machine Learning Concepts*. *7*(6), 9.

Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2013). The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, *47*, 253–279. https://doi.org/10.1613/jair.3912

Berner, C., Brockman, G., Chan, B., Cheung, V., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., Józefowicz, R., Gray, S., Olsson, C., Pachocki, J., Petrov, M., Salimans, T., Schlatter, J., Schneider, J., Sidor, S., Sutskever, I., … Zhang, S. (n.d.). *Dota 2 with Large Scale Deep Reinforcement Learning*. 66.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). OpenAI Gym. *ArXiv:1606.01540 [Cs]*. http://arxiv.org/abs/1606.01540

Chhabra, V., & Tomar, K. (2015). Artificial Intelligence: Game Techniques Ludo—A Case Study. *Advances in Computer Science and Information Technology*, *2*(6), 5.

Derakhshan, D. (n.d.). *APPLIED REINFORCEMENT LEARNING WITH USE OF GENETIC ALGORITHMS*. 10.

Duan, Y., Chen, X., Houthooft, R., Schulman, J., & Abbeel, P. (2016). Benchmarking Deep Reinforcement Learning for Continuous Control. *ArXiv:1604.06778 [Cs]*. http://arxiv.org/abs/1604.06778

Feng-Hsiung Hsu. (1999). IBM's Deep Blue Chess grandmaster chips. *IEEE Micro*, *19*(2), 70–81. https://doi.org/10.1109/40.755469

Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., Kumar, V., Zhu, H., Gupta, A., Abbeel, P., & Levine, S. (2019). Soft Actor-Critic Algorithms and Applications. *ArXiv:1812.05905 [Cs, Stat]*. http://arxiv.org/abs/1812.05905

Hölldobler, S., Möhle, S., & Tigunova, A. (n.d.). *Lessons Learned from AlphaGo*. 11.

Juliani, A., Berges, V.-P., Teng, E., Cohen, A., Harper, J., Elion, C., Goy, C., Gao, Y., Henry, H., Mattar, M., & Lange, D. (2020). Unity: A General Platform for Intelligent Agents. *ArXiv:1809.02627 [Cs, Stat]*. http://arxiv.org/abs/1809.02627

Lanctot, M., Lockhart, E., Lespiau, J.-B., Zambaldi, V., Upadhyay, S., Pérolat, J., Srinivasan, S., Timbers, F., Tuyls, K., Omidshafiei, S., Hennes, D., Morrill, D., Muller, P., Ewalds, T., Faulkner, R., Kramár, J., De Vylder, B., Saeta, B., Bradbury, J., … Ryan-Davis, J. (2020). OpenSpiel: A Framework for Reinforcement Learning in Games. *ArXiv:1908.09453 [Cs]*. http://arxiv.org/abs/1908.09453

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (n.d.). *Playing Atari with Deep Reinforcement Learning*. 9.

Nanaware, P., Huisan, S., Bhalerao, A., & Rathod, T. (2020). *Multiplayer Carrom Board Game with an Ai Opponent*. *07*(05), 4.

Nawshin, S., & Saifuzzaman, M. (2018). *The Role of Shologuti in Artificial Intelligence Research: A Rural Game of Bangladesh*. 10.

Pettingzoo_gym_for_multi_agent_reinforcement_learning.pdf. (n.d.).

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., Lillicrap, T., & Silver, D. (2020). Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. *Nature*, *588*(7839), 604–609. https://doi.org/10.1038/s41586-020-03051-4

Schulman, J., Levine, S., Moritz, P., Jordan, M. I., & Abbeel, P. (2017). Trust Region Policy Optimization. *ArXiv:1502.05477 [Cs]*. http://arxiv.org/abs/1502.05477

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. *ArXiv:1707.06347 [Cs]*. http://arxiv.org/abs/1707.06347

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., & Hassabis, D. (2017). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *ArXiv:1712.01815 [Cs]*. http://arxiv.org/abs/1712.01815

Tesauro, Gerald (March 1995).pdf. (n.d.).

Zha, D., Lai, K.-H., Cao, Y., Huang, S., Wei, R., Guo, J., & Hu, X. (2020). RLCard: A Toolkit for Reinforcement Learning in Card Games. *ArXiv:1910.04376 [Cs]*. http://arxiv.org/abs/1910.04376