Street Fighter Game with Reinforcement Learning

Submitted By: Tania Kapoor

Registration no.: 2320757

A thesis submitted for the degree of Master of Science in Artificial Intelligence

Supervisor: Dr Michael Fairbank
School of Computer Science and Electronic Engineering
University of Essex

October 2024

Abstract
This project successfully integrated a Street Fighter game with reinforcement learning (RL) by developing game-playing bots. A custom OpenAI Gym environment was designed to make the game playable via Python and enable the implementation of reinforcement learning. Proximal Policy Optimisation (PPO) and Deep Q-Networks (DQN) were used to train the agent. The trained agents were able to learn and optimise their strategies according to the situation. The agents demonstrated their ability to outperform a random player. Also, a PPO-trained model was used as a benchmark to evaluate the performance of other RL models. The agents were tested in various setups like PPO vs. DQN and PPO vs. PPO, which demonstrates PPO's adaptability to dynamic changes and DQN's proficiency in exploiting deterministic reward structures. Additionally, the project included evaluation with humans, which helped to gather real-time feedback. During the process, critical fixes were applied to the game code, and a pull request was submitted and merged into the original GitHub repository. The developed Gym wrapper could also be used for further research, and other developers can train their own models using it. The work highlights the potential of RL in gaming and its broader applicability to robotics and autonomous systems.

Acknowledgements
This dissertation would not have been possible without the constant support, guidance, and encouragement of my supervisor, Dr Michael Fairbank. I have learnt a lot under your mentorship and would always be grateful for your insightful and valuable feedback. I truly feel honoured to have been taught by you.

I would like to thank my family, especially my mother. Your unconditional love, understanding, and unwavering belief in me have been my constant source of strength. Thank you for always being there for me and teaching me to never give up.

I am grateful to all the people who participated in the survey. This project would never have been completed without you all. Thank you so much.

Finally, a special thanks to my friend Vishwas, through your help and support, I was able to transform my roadblocks into achievements.

Table of Contents

1. Introduction

(The following section is same as that in proposal [16])

In the wake of artificial intelligence, perpetual advancement can be seen in the gaming sector these days. Reinforcement learning (RL) has played a vital role in it. With RL, not only has the development process changed, but the gaming experience has also been taken to the next level by offering challenging gameplay. RL allows the creation of non-player characters (NPCs) that learn and adapt to player strategies. These NPCs provide a consistently stimulating experience as they can adjust their tactics based on the player's behaviour, making each game session unique. Google DeepMind's AlphaGo [1] is an exemplary example of this.

AlphaGo, a computer system that is trained using reinforcement learning algorithms and deep neural networks to master the game of Go, defeated legendary Go players and was declared the first-ever AI system to win against Go professionals [1]. Since Go is renowned for its complexity in gameplay, with around 2.1 x 10^170 board positions (vastly more than the number of atoms in the observable universe), an AI system to beat a pro is a breakthrough for many reasons. The most important one is that just through exploration and exploitation (to get maximum reward), they can plan and look ahead, giving them the ability to solve the most intricate of the problems by themselves in real-time.

Figure 1. AI (Copilot Designer) generated image of AI vs Go professional

Figure 2. AI (Copilot Designer) generated image of Street Fighter game

In this project, the objective is to build a similar bot trained using neural networks and RL algorithms, but to master the game of street fighting. Street fighter games are known for their intricate and well-balanced fighting mechanics. Players battle against other players, using a combination of punches, kicks, and other unique moves with intention to hurt their opponent's health. The games emphasize skill, timing, and strategy, requiring players to master various combos and techniques. Reinforcement Learning (RL) has been effectively used in Street Fighter games to develop advanced AI agents. These agents learn complex strategies, adapt to player actions, and provide a challenging gaming experience. Through self-play, they explore a variety of strategies and counterstrategies, resulting in strong performance. Curriculum learning helps them build competence gradually, starting with simpler opponents and moving to more complex ones.

Decision-making is optimized with policy gradient methods like Proximal Policy Optimization (PPO). Evaluations using metrics like Elo ratings show the AI's ability to handle different scenarios and opponents. All in all, RL has significantly enhanced the creation of dynamic and engaging AI opponents in Street Fighter games [16].

1.1. Scope

(The following section is similar to that of proposal [16] with a few additions)

The scope of this project is to integrate a street fighter game with reinforcement learning. An OpenAI Gym [2] wrapper was created to provide an environment for reinforcement learning. This gym wrapper created could also be used by other developers to train their agents for the game. Additionally, the RL agent built can learn and optimise its strategy so that it can outperform a random player. Also, the project showcases the application of reinforcement learning algorithms in the building of intelligent game agents in Python. A small survey was also conducted to get feedback for the trained bot in real-time gameplay with humans. This project does not involve the building of the Street Fighter game and would focus more on the RL section of the project. An already-built close approximation of the Street Fighter game [3] would be used, which was made by Drew Berry, a student at the University of Essex. Some fixes to the game were applied though.

1.2. Research Questions

1. Would it be possible to create a bot that could play the Street Fighter game?
2. Would this bot be better than a player who takes random actions?
3. Would it be possible to beat this new bot by training another bot player?

1.3. Definitions, Acronyms, and Abbreviations

(The following section is the same as that in the proposal [16])

Terms/ Acronyms, and Abbreviations	Description
RL	Reinforcement Learning
AI	Artificial Intelligence
NPC	Non-Player Character
PPO	Proximal Policy Optimization
HRL	Hierarchical Reinforcement Learning
DP-PPO	Decay-based Phase-Change Memristive Character-Type Proximal Policy Optimisation
MARL	Multi-Agent Reinforcement Learning
PSRO	Policy Space Response Oracles
FSP	Fictitious Self-Play
IPPO	Independent Proximal Policy Optimization
MDP	Markov Decision Process
DQN	Deep Q-Networks
A2C	Advantage Actor-Critic
P1	Player 1
P2	Player 2

2. Literature Review

This section outlines the findings of some of the similar works done and the technical foundation required for the development of the project [16].

2.1. Related Work

(All the related works are the same as that in the proposal [16] except the last one [8].)

In order to portray exciting gameplay, the NPCs must be believable and play in a human-like fashion. To achieve this, an innovative approach has been presented in [5]. The two fundamental challenges addressed in [5] are the exploration of high-dimensional state-action spaces and producing behaviour diversity that is adaptative to the opponent's tactics. The framework used is a model-based hierarchical reinforcement learning (HRL) technique, which breaks complex tasks into smaller sub-problems using temporally extended actions.

This hierarchical structure not only simplifies the learning process by reducing the state space but also enhances the believability of the NPC. The effectiveness of HRLB^2 was evaluated using a third-person Turing test in the context of Street Fighter IV. The bot's human-likeness was assessed by comparing its behaviour to that of human players and the built-in AI agents. With more than 60% human-likeliness, the results indicated that the HRLB^2 bot exhibited more human-like behaviour, demonstrating the potential of hierarchical reinforcement learning in creating believable game AI. The work was also cited in [4], which highlighted various other approaches using RL and AI, like RL with imitation learning, utilised in order to build human-like NPCs.

Another study [6] demonstrated the effectiveness of combining phase-change memory with deep reinforcement learning in order to create AI agents that can rapidly outperform human players in complex games like Street Fighter. The novel Decay-based Phase-Change Memristive Character-Type Proximal Policy Optimisation (DP-PPO) algorithm and hybrid training approach provide a robust framework for future advancements in AI-driven competitive gaming and beyond. The findings suggest that phase-change memristive reinforcement learning can be a powerful tool for developing AI agents capable of complex physical and strategic manoeuvres. Additionally, the use of Gym Retro, an extension of OpenAI Gym for games, was crucial in this research, providing a rich environment for training and evaluating the agent. Gym Retro's ability to emulate classic games allowed the researchers to leverage existing game dynamics and interactions, thereby facilitating a more realistic and comprehensive training process.

Additionally, this paper was also cited in [7], in which a platform for benchmarking the multi-agent reinforcement learning (MARL) algorithms is created. The platform is designed such that experimental evaluations of existing RL and MARL algorithms in both singleplayer and two-player modes using Street Fighter are provided. In single-player mode, a curriculum learning-based scheme is proposed, successfully training a general RL agent capable of consistently defeating CPU opponents across different characters. In two-player mode, Elo ratings and exploitability tests are utilized as evaluation criteria. Implementations of league training and Policy Space Response Oracles (PSRO) produce stronger agents compared to Fictitious Self-Play (FSP) and Independent Proximal Policy Optimization (IPPO) in terms of Elo ratings. However, the study finds that single-agent RL and human players can exploit all agents trained using current algorithms.

In another work [8], PPO+LSTM was utilised for playing League of Legends. PPO, a policy gradient-based algorithm, was employed to optimise the policy, with rewards calculated based on points of the player's and enemy's champions. Also, the LSTM model was used to observe the game state and actions like attacking an enemy champion or minions or fleeing. The bot was trained by playing against a built-in League of Legends bot in a custom 1v1 game mode. The LSTM+PPO bot was found to outperform the LSTM-only bot in achieving first blood against the enemy champion. With further improvements to the action space and some additional training hours, the LSTM+PPO bot could achieve near-human-level performance in the customised game environment.

2.2. Background

(The subsections from 2.2.1 to 2.2.5 are the same as that in proposal [16])

2.2.1. Reinforcement Learning

Reinforcement learning [9] or RL is a machine learning technique used to solve problems based on rewards and punishments. It is an iterative process. Unlike supervised and unsupervised learning, RL depends on exploration rather than training dataset. Most learning tasks involve:

- **Environment**: The external system or surroundings with which the agent interacts. It could be a game, a robot, or any other dynamic environment.
- **Agent**: The learner that interacts with the environment. It observes states, takes actions, and receives rewards.
- **State (s)**: The current situation or configuration of the environment. The agent perceives the state to make decisions.
- **Episode**: The process of agent accumulating rewards from the scores defined.

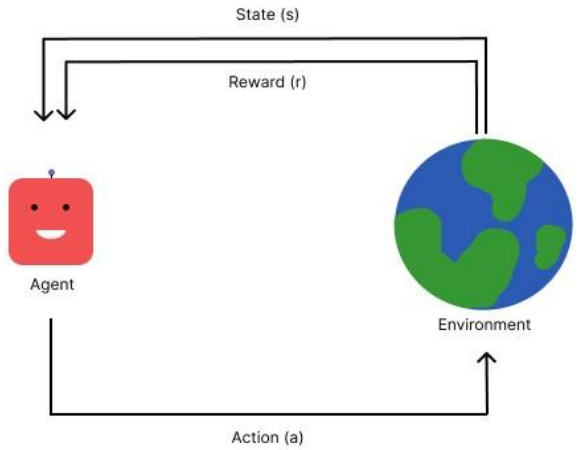


Figure 3. Reinforcement Learning process

- **Action (a)**: The choices the agent can make. Actions influence the environment and lead to transitions to new states.
- **Policy (π)**: The strategy that maps states to actions. It guides the agent's decisionmaking process.
- **Reward (r)**: The feedback signal received by the agent after taking an action. It indicates the desirability of the action.
- **Value Function (V)**: Estimates the expected cumulative reward from a given state. It helps the agent evaluate different states.
- **Q-Function (Q)**: Estimates the expected cumulative reward for taking a specific action in a given state. It guides action selection.
- **Exploration vs. Exploitation**: Balancing between trying new actions (exploration) and choosing known good actions (exploitation).
- **Learning Algorithm**: The method used to update the agent's policy based on experience. Examples include Q-learning, SARSA, and policy gradients.
- **Cumulative Discounted reward**: It is the total reward an agent expects to receive over the future, with each reward being reduced by a factor that reflects its distance in the future. This reduction is done using a discount factor, denoted by γ (gamma), where $0 \leq \gamma \leq 1$. Mathematical notation: $G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$.

During the learning task, at every discrete timestep, the agent and environment interact. For a_t (action at any time t) the agent makes, the environment responds with a state s_t and a reward r_t. After a timestep, the environment presents the agent with a new state s_{t+1} and a reward r_{t+1}. For optimal learning, the agent's job is to maximize the cumulative discounted reward.

2.2.2 Markov Decision Process (MDP)

An RL task can be better explained using the Markov decision process mathematically. MDPs provide a formal foundation for modelling decision-making problems where outcomes are influenced by both deterministic and probabilistic factors [9]. It is defined by a tuple (S,A,P,R,γ) where S is a set of states, A is a set of actions, $P(s'|s,a)$ is the transition probability function representing the probability of moving from state s to state s' for a given action a, $R(s,a,s')$ is the reward function giving the immediate reward received after transitioning from s to s' due to action a, and $\gamma \in [0,1]$ is the discount factor for future rewards. The objective is to find a policy $\pi: S \rightarrow A$ that maximizes the expected cumulative discounted reward.

2.2.3 Policy Gradient Methods

Policy gradient methods [11] are powerful tools in reinforcement learning, especially suitable for complex and high-dimensional environments like video games. These methods encourage exploration and are adaptable to both discrete and continuous action spaces, making them ideal for sophisticated game AI development. In the policy gradient approach, the optimization of the policy is done directly. Instead of deriving the policy from the estimated value functions, policy gradient methods introduce parameters for policy which are then optimized to maximize the expected reward. Given α being the learning rate, and $J(\theta)$ as the expected cumulative reward, the parameter θ is updated in the direction of the gradient of $J(\theta)$. This is called the gradient ascent and the equation is given by: $\theta \leftarrow \theta+\alpha\nabla_{\theta}J(\theta)$.

Some popular algorithms following this approach are:

1. **REINFORCE**:

A simple policy gradient algorithm where the gradient is estimated using Monte Carlo methods. The update rule for θ: $\theta_{t+1} = \theta_t + \alpha G_t \nabla \ln \pi(A_t|S_t, \theta_t)$

2. **Actor-Critic Methods**:

These methods combine policy gradient (actor) with value function approximation (critic) to reduce variance in the gradient estimates. A2C (Advantage Actor-Critic) and PPO (Proximal Policy Optimization) are well-known actor-critic methods.

2.2.4 OpenAI Gym

OpenAI Gym [2] is a toolkit used for developing and comparing reinforcement learning (RL) algorithms. Utilities for managing and interacting with environments, such as functions for resetting environments, stepping through time, rendering environments for visualization, and monitoring performance metrics, are provided by it. These components enable experimentation with and benchmarking of various RL algorithms in standardized environments, facilitating progress and comparison in the field of reinforcement learning. A wide variety of environments for testing and benchmarking the performance of RL agents are provided by OpenAI Gym. These environments encompass a spectrum of tasks, from basic control problems to more intricate simulations involving robotics and video games. Atari, Box2D, and robotics are among the variety of environments provided by it. Additionally, it can be easily integrated with popular RL libraries and frameworks, such as TensorFlow, PyTorch, and Stable Baselines, allowing for a seamless development experience.

2.2.5. Nash Equilibrium

The Nash equilibrium [10] is a concept from game theory that describes a situation where, in a game involving two or more players, each player has chosen a strategy and no player can benefit by changing their own strategy. Essentially, it's a state where every player's strategy is optimal given the strategies of all other players, leading to a stable outcome where no player is willing to change their strategy as it may result in punishment instead of reward.

In the context of self-play in a video game like a street fighting game involving two agents, each agent is continually adapting and evolving strategies based on past outcomes. Self-play is a method often used in reinforcement learning where the same algorithm plays against itself, learning from each encounter. This method can rapidly accelerate the learning process and lead to the discovery of complex strategies.

Potential Issues with Nash Equilibria in Self-Play:

1. **Stagnation at Suboptimal Equilibria:** If both agents reach a Nash equilibrium that is not globally optimal (i.e., there could be better strategies that neither agent explores because they are currently at a local optimum), they might stop improving. Each agent finds that changing their own strategy does not yield better

results, so no further learning occurs. This is especially problematic in complex games where many possible equilibria exist, and not all are equally desirable.

2. **Cycling Through Equilibria:** In some games, especially those with multiple Nash equilibria, agents might end up cycling through a set of strategies without converging to a stable strategy. This can make the learning process inefficient and unpredictable.

Addressing Nash Equilibria in Self-Play:

1. **Exploration Techniques:** To prevent agents from getting stuck in suboptimal Nash equilibria, adding more exploration to the agents' learning algorithms can be beneficial. Techniques like epsilon-greedy (where there's a probability ϵ\epsilonϵ that the agent will try a random action), entropy bonuses (adding a term to encourage more diverse actions), or other advanced exploration methods can encourage the agents to explore potential strategies beyond the current equilibrium.
2. **Varying Opponents:** Periodically changing the opponents or strategies that an agent faces can help prevent the convergence to a non-ideal Nash equilibrium. This can be implemented by introducing agents trained with different strategies or by periodically resetting the agent's strategy to explore different parts of the strategy space.
3. **Multi-Objective Optimization:** Sometimes, adjusting the reward structure so that it's not just about winning or losing but also about how the win or loss is achieved can shift the Nash equilibria. For example, adding rewards for aggressive tactics or for using a variety of moves might prevent the game from converging to a dull equilibrium where one safe strategy dominates.

Consider a simplified scenario in a street fighting game where each player can choose between two moves: Attack or Defend. To illustrate Nash Equilibrium in this context, a payoff matrix is created where each entry shows the outcomes for Player 1 (row player) and Player 2 (column player) when they choose their respective strategies. Assumptions for the Payoff Matrix:

Attack vs. Attack: Both players attack, resulting in a high-risk scenario. Each receives moderate damage, leading to a neutral outcome for both.

Attack vs. Defend: The attacker gets through the defense, causing significant damage to the defender. The attacker gains an advantage.

Defend vs. Attack: The defender successfully blocks the attack, countering effectively. The defender gains an advantage.

Defend vs. Defend: Both players defend, resulting in a stalemate with minimal interaction.

	Agent 2: Attack	**Agent 2: Defend**
Agent 1: Attack	(0, 0)	(1, -1)
Agent 1: Defend	(-1, 1)	(0, 0)

Table 1. Payoff matrix to illustrate Nash Equilibrium

In last two cases where both defend or both attack, neither of these scenarios gives either player a reason to change their strategy if the other player keeps their choice. Therefore, these can be considered Nash equilibria because no player can improve their outcome by changing their strategy on their own.

2.2.6. Curriculum Learning

Curriculum learning is a training technique that is used in RL where the training environment is too difficult. The idea behind curriculum learning is to start the agent in a simpler environment and then gradually increase the complexity once the agent seems to improve.

Curriculum learning is useful in RL mainly for:

Easier learning: By starting in a simpler environment, the agent can learn the basic skills and strategies required to succeed more easily. This leads to faster convergence and better overall performance compared to immediately placing the agent in a very complex environment.
Exploration and exploitation balance: Curriculum learning helps the agent strike a better balance between trying new things and exploitation (using what it has learned). In the early, simpler phases, the agent can focus more on exploration to discover effective strategies. As the environment becomes more challenging, the agent can rely more on exploitation.
Avoidance of local optima: Jumping straight into a complex environment can sometimes cause the agent to get stuck in local minima, where it learns a suboptimal policy. This could be fixed by gradually increasing the difficulty; curriculum learning can help the agent discover a more globally optimal policy.
Transfer learning: The skills and knowledge the agent acquires in the earlier, simpler phases can often be transferred and applied to the more complex phases, which further accelerates the learning process.

2.2.7. PPO - policy networks

In PPO, the policy network [11] is typically implemented as a feedforward neural network. The input to the policy network is the current state of the environment, which can be represented as a feature vector or an image.
The hidden layers of the policy network use various activation functions, such as ReLU (Rectified Linear Unit) or tanh, to introduce non-linearity and enable the network to learn complex representations of the input state. The number and size of the hidden layers are hyperparameters that can be tuned to find the optimal architecture for a given problem.
The output layer of the policy network is a softmax layer, which produces a probability distribution over the available actions. The softmax function ensures that the output probabilities are non-negative and sum up to 1, representing a valid probability distribution.
Formally, the policy network in PPO can be represented as:
$\pi(a|s) = \text{softmax}(f(s; \theta))$
Where:

- $\pi(a|s)$ is the probability of taking action a in state s.
- $f(s; \theta)$ is the output of the neural network with parameters θ, representing the unnormalized log-probabilities of the actions.
- $\text{softmax}(\cdot)$ is the softmax function, which converts the unnormalized log-probabilities into a valid probability distribution.

During training, the policy network's parameters θ are updated using the PPO algorithm, which aims to optimise the policy by maximising the expected cumulative reward while ensuring that the updated policy does not deviate too much from the previous policy.

2.2.8. Elo Ratings

Elo ratings [12] can be a valuable metric for evaluating agents in reinforcement learning, particularly in competitive or adversarial environments like your Street Fighter setup. The system was originally developed by Arpad Elo for chess, but it has since been adapted for many other competitive activities, including video games, sports, and even online multiplayer games. Steps for how the Elo rating system works:

1. Initial Rating: Each player starts with an initial rating, typically around 1500. Stronger player is given a higher rating.
2. Rating Changes: After each match or game, a player's rating is adjusted based on the outcome and the expected result:

- If a lower-rated player wins against a higher-rated player, they gain more points.
- If a higher-rated player wins against a lower-rated player, they gain fewer points.
- Draws or close matches result in smaller rating changes.

Math for expected and actual outcome:

1. **Expected Outcome**: Before a match, the system calculates the expected score for each player based on their ratings: $EA=\frac{1}{1+10^{\frac{RB-RA}{400}}}$, where:
   - EA: Expected score of Player A.
   - RA: Rating of Player A.
   - RB: Rating of Player B.
2. **Actual Outcome**: Assigned scores based on the match result:
   - Win = 1
   - Loss = 0
   - Draw = 0.5
3. **Rating Update**: Adjust ratings using the formula: $RA'=RA+K\times(SA-EA)$, where:
   - RA′: Updated rating for Player A.
   - K: A constant determining how much the rating changes (e.g., 20 in chess).
   - SA: Actual score of Player A.
   - EA: Expected score of Player A.
3. Calculation: The specific mathematical formula considers:
   - The current ratings of both players
   - The actual result of the match (win, loss, or draw)
   - An expected probability of winning based on the rating difference
4. Predictive Power: Over time, the Elo rating becomes a good predictor of a player's relative skill level. A higher Elo rating suggests a more skilled player.

Advantages: It is known for its fairness as it takes opponent strength into account. Also, it reflects recent performance.

Limitations: Initial ratings may not reflect true skill. Additionally, it only measures relative skill and it doesn't account for other factors like fatigue. Moreover, players on a losing streak may drop disproportionately in rating.

3. Methodology

3.1. Design and Workflow

Design: There are 4 main components of this project:

1. The first one is the game engine built in Java which simulates the gameplay of Street Fighter. The game is a close approximation to the original one and has various gameplay modes. In this project we would use the 'story' mode and 'arcade' mode. In the story mode, player 1, which is the left player and is originally idle, will be made to

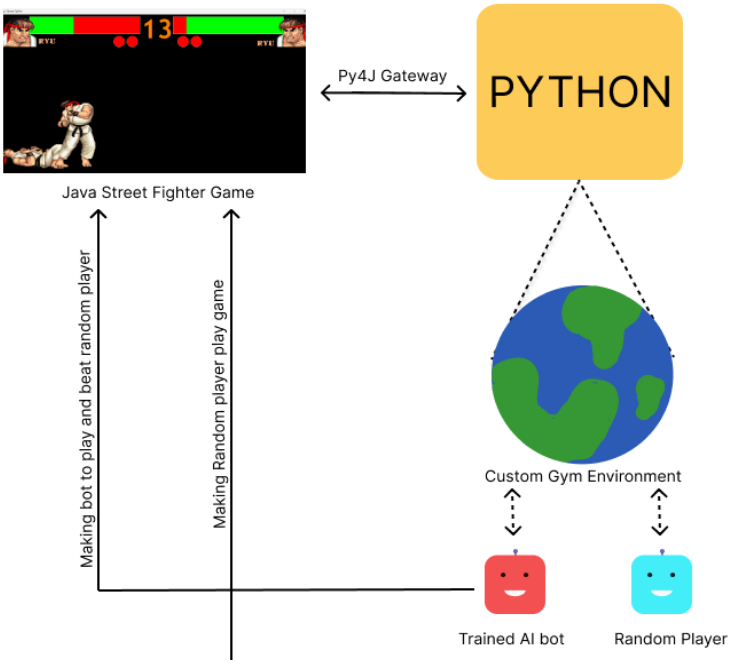


Figure 4. Design and workflow

play via python by trained RL bot or random player taking random actions. The player 2, is a built-in bot in the java game who is designed to take random actions. In arcade mode, both the players are idle, i.e., no built-in bot players. So, in arcade mode, player 1 would be made to be played by the previously trained agent against player 2, which could be another trained model or a human.

2. The second component is the Py4J gateway which acts a bridge between Python and the Java game. This enables the implementation of reinforcement learning on the game by making the game accessible via RL libraries.
3. Third component is the Python environment which acts as the control hub for implementing RL in the game. Creation of gym environment, training of the agent and testing, all these tasks are done in Python.
4. Gym Environment, the fourth component acts as an interface between Python and the game. It converts the game data into observations (like the player's health, velocity, position etc.) that the AI can use. The bot or random player would act on this environment and would be returned a reward (positive or negative) and the next state in the environment which would reflect the changes made by that action.

Workflow: A random player is first used to test the environment and benchmarking. Then a Reinforcement Learning (RL) bot is trained to play the game, optimizing its performance through reward-based learning.

3.2. Making game playable via Python

The Street Fighter game is in Java but all the reinforcement learning algorithms to be used are in Python. Therefore, a connection between the two is required. Py4j library is being used to create the bridge between Python and Java. A gateway server is created by it that needs to be initialized from one end and fetched from the other. In this project, the entry point for this connection is initialized through Java and the code is run in Python to get this entry point to make the connection possible. Once the gateway is built, function calls, along with getter and setter methods made in Java are mainly used to access the game in Python. A couple challenges faced with this method are:

1. Not all datatypes are supported by the library. A workaround to this is to use functions to return the values instead of fetching the java variable directly. Once the value is returned, cast it to the desired datatype in Python.
2. Sometimes the gateway breaks after a few hours and the code needs to be executed again to rebuild the bridge.

```java
import java.io.IOException;

import py4j.GatewayServer;

public class GameEntryPoint {
    private Game game;

    public GameEntryPoint() throws IOException {
        game = new Game();
    }

    public Game getGame() {
        return game;
    }

    public static void main(String[] args) {
        try {
            GameEntryPoint entryPoint = new GameEntryPoint();
            GatewayServer server = new GatewayServer(entryPoint);
            server.start();

            System.out.println("Gateway Server Started");

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Figure 5. Code to build gateway on Java side

```python
from py4j.java_gateway import JavaGateway

# Connect to the Java GatewayServer
gateway = JavaGateway()
# Get the Game instance through the entry point
game = gateway.entry_point.getGame()
print('gateway created')
```

Figure 6. Code to connect to gateway on Python side

3.3. Building Custom Gym Environment

Just like how humans observe their surroundings and make decisions accordingly, the goal is to make the RL agent do the same. To achieve this, building an environment is essential. Unlike humans who use their senses to observe their environment, an RL agent would depend on the game data such as the player's health, position etc. To provide this information to the bot, a custom OpenAI gym is to be created to define the state, reward, and action for the game. Creating a gym environment for a Street Fighter game requires defining several key

components to enable effective interaction with reinforcement learning algorithms. A breakdown of the state, reward, action space, observation space, and other necessary parameters:

1. **State**: The state, which represents the current status of the game environment, typically includes relevant information for AI decision-making. For a street fighting game, the state includes health, position, distance between players, player moves etc.

2. **Reward**: The reward function, which defines how the agent is reinforced based on its actions, can be designed to encourage winning and penalise losing in a Street Fighter game. Rewards could be a health loss, damage dealt, victory or defeat, or a score.

3. **Action Space**: The action space, which defines the set of all possible actions the agent can take, could include, for movement: move left, move right, jump, crouch; for attacking: light attack, medium attack, or heavy attack.

4. **Observation Space**: Imagine playing Street Fighter, to make smart moves, one needs to know where both fighters are, how fast they're moving, how much health they have left, and what actions they're currently performing (like attacking or dodging). Just like how a human player uses their senses and observations to react during a fight, the bot could use the detailed information about the environment such as his position, velocity and opponent's velocity and how far he is etc., to perform effectively and improve over time.

In this setup, the AI would receive all this important information as observation space which includes:

- **Positions and Velocities:** With this, it knows exactly where each fighter is on the screen and how quickly they're moving. For example, Fighter A might be at the centre moving left, while Fighter B is advancing from the right.
- **Health Status:** The AI can know how much health each fighter has left. If Fighter A is low on health, the AI might decide to defend or attack more carefully.
- **Distance Between Fighters:** Knowing how far apart the fighters are which would help the AI decide whether to attack, block, or move closer.
- **Current Actions:** It can tell if a fighter is currently attacking, defending, or performing a special move. For instance, if Fighter B is throwing a heavy punch, the AI might choose to block or dodge.

By providing such information, the AI gets a complete picture of what's happening in the game at any moment. This helps it make smarter decisions, like knowing when to attack, defend, or retreat. Another way to provide this information to the agent is by using snapshots of the game which would be pixel data for each state. This approach would be providing 'eyes' to the model as pixels contain all visual details, capturing everything on the screen, including subtle movements, which might not have been provided in previous approach. With this, AI doesn't have to rely on predefined data structures, making it adaptable to different games or scenarios without changing the input format. However, processing images requires more advanced neural networks like CNN and the training time is slower too. Due to time constraint of the project, this approach could not be explored.

3.3.1. Gym Wrapper

The Gym wrapper would help in making the environment in a standardised form that is acceptable by RL libraries. The wrapper would require the implementation of the following methods:

• **Init**: responsible for initialising observation space, action space, state, and other parameters.
• **Step**: responsible for applying action to the current state, treating actions, and returning the appropriate next state and reward.
• **Render**: The rendering of the environment to the screen is taken care of by this.
• **Reset**: This method would reset the environment after the episode is completed and return the initial state.

3.3.2. Preprocessing Environment

Like any other learning method, the preprocessing step plays a crucial role in RL. Preprocessing the environment would help improve learning efficiency, simplify learning, and facilitate generalisation. The main objective of this step is to simplify the environment, as the simpler it is, the more efficiently the model will learn and perform. Some preprocessing steps that would be taken in this project would be to normalise the observation space as it ensures that the input data (observations) fed into the model is scaled to a consistent range. The purpose of doing this is to enhance the efficiency, stability, and accuracy of the training process. Additionally, reward optimisation is also vital for this, as it would help the model learn what is expected of it and address the issue of sparse reward.

3.3.3 Reward Function Design

The reward function is designed to balance the agent's behaviour in the competitive environment. Its methodology is based on encouraging aggressive strategies, effective attacks, and close-range engagements while penalizing passiveness as well as taking damage. The goal is to balance out the short-term and long-term objectives of engaging the player in effective gameplay but also winning the game ultimately.

3.4. Building and Training RL bots

3.4.1 For player 1 to beat a random player

After building the environment, the next step is to train the model. The way the model is trained and for how long plays a crucial role in its performance. In this project player 1 is made to play against player 2 who would take random actions throughout training to represent an unpredictable opponent. Also, during the process, the environment continuously generates states, actions, and rewards for Player 1 based on its interactions. Player 1 learns to optimize its performance by maximizing rewards (using algorithms like PPO, DQN, or A2C). This approach is adopted because of its simplicity as there is no external dependence, like no need for human interaction or live systems.

3.4.1.1 Algorithm Selection

Several RL algorithms have been made available via libraries like keras-rl2 and stable_baselines3. In this project, most of the models were trained using the latter one. The algorithms explored were:

a) Proximal Policy Optimization (PPO):

PPO is best suited for continuous or discrete action spaces with complex, high-dimensional environments (e.g., fighting games, robotics).

2) **Reason for Selection**:

a) PPO handles stochastic policies, which can explore the environment more effectively than deterministic approaches.
b) It uses a clipped objective function to ensure stable and reliable learning, reducing drastic policy updates.
c) Effective for environments requiring real-time decision-making due to its robustness to noisy reward signals.

3) **Why PPO for the Fighting Game**: The complex interactions between Player 1 and Player 2 require a policy capable of learning diverse strategies and adapting to dynamic game states which could be handled well using PPO.

2. Deep Q-Learning (DQN):

DQN is seen to work well for environments with discrete action spaces and lower-dimensional state representations (e.g., grid-based games or simplified setups).

- **Reason for Selection**:
    1. Uses Q-value estimation to directly learn the value of state-action pairs, making it efficient in simpler environments.
    2. DQN performs well in environments where the state space can be easily represented using neural networks.
    3. Extensions like Double DQN and Dueling Networks can improve stability and learning in more complex tasks.
- **Why DQN for the Fighting Game**: If the fighting game environment has a smaller action space and straightforward state representations, DQN might be a simpler and computationally cheaper alternative.

3. Advantage Actor-Critic (A2C):

A2C is suited for moderately complex environments with discrete or continuous action spaces and when computational efficiency is a priority.

- **Reason for Selection**:
    1. **Actor-Critic Architecture**: Combines the benefits of policy-based methods (actor) and value-based methods (critic) for stable learning.
    2. **Synchronous Updates**: All workers update the model synchronously, which is computationally efficient and ensures stable learning in comparatively smaller setups.
    3. **Advantage Estimation**: Focuses on reducing variance in policy gradients by using advantage values. The advantage is a measure of how much better or worse a particular action performed compared to the expected performance at that state. It provides a way to evaluate whether taking an action was beneficial relative to the baseline expectation.
- **Why A2C for the Fighting Game**: The environment's complexity could benefit from A2C's synchronous updates and advantage estimation, which helps stabilize learning for dynamic interactions between players. Moreover, it is computationally simpler than PPO, making it a good balance between performance and resource consumption.

Aspect	PPO	A2C	DQN
Action Space	Continuous/Discrete	Continuous/Discrete	Discrete only
Environment Complexity	Handles high complexity well	Handles moderate complexity	Handles low complexity
Exploration	Stochastic policies explore well	Balanced exploration	Limited exploration
Stability	High due to clipped updates	Moderate	Can be unstable without extensions
Learning Efficiency	Slower, computationally costly	Moderate, synchronous updates	Faster, requires simpler environments
Advantage Handling	Uses a clipped surrogate advantage	Direct advantage estimation	No advantage mechanism

Table 2 PPO vs. A2C vs. DQN

3.4.2. For Player 2 to beat trained Player 1

Once Player 1 is able to successfully beat a random player, an attempt is to be made to beat this player. Since the actions taken by the AI agent would be more strategic than random, some advanced steps would be implemented.

3.4.2.1. Curriculum Learning Integration

In this, the environment's difficulty is gradually increased in phases, as starting with simpler tasks helps the agent build foundational skills before tackling more complex challenges. Otherwise, the agent could find the environment too hard to learn and might settle for a lower reward by getting stuck in local minima. Also, performance metrics (e.g., average rewards) are monitored to determine when to transition to the next phase. Once the model learns to do well in the current phase, it is moved to the next, more difficult state, as gradual difficulty prevents the agent from becoming overwhelmed and promotes stable learning. Additionally, episodes where transitions occur are to be logged to track curriculum progression. By doing these performance-based transitions, it is ensured that the agent only advances when it has mastered the current difficulty, which would help avoid wasted training time on overly challenging scenarios.

3.4.2.2. Memory Integration

Feature extraction using an LSTM-based approach provides the model with a form of "memory" which enables it to account for temporal dependencies present in the environment. In games such as Street Fighter, decision-making is not purely based on the current state but is influenced by sequences of prior states, such as recognizing an opponent’s attack patterns or anticipating their next move. The LSTM-based feature extractor processes sequential observations to encode this temporal information effectively. By using its internal memory, the LSTM retains critical details from past states and integrates them with the current observation. This memory allows the model to better understand patterns over time, like timing counters or exploiting repeated moves by the opponent. After extracting the temporal features, the fully connected layer reduces dimensionality, making the input manageable for the policy network while retaining essential information. By feeding the LSTM-processed features into PPO, the algorithm effectively learns from both the immediate observations and their historical context which would enable more strategic and informed decisions in the environment as temporal patterns could be crucial in it. This approach improves the model's ability to take better actions, leading to more strategic and effective gameplay.

3.5. Evaluation Metrics and Testing

3.5.1. Evaluation Metrics

The evaluation metrics help in understanding how well the model is doing during the training and testing phase. These metrics ensure the model learns effectively and generalizes well. Some key metrics that would be implemented are:

1. Average Episode Reward

- **What it measures**: The mean total reward obtained by the agent over multiple episodes.
- **Importance**: Higher rewards indicate better performance and alignment with the desired objectives. It Compares average rewards over training and testing phases to ensure consistent performance.
- **How to achieve**: The rewards during all the matches during testing time would be stored.

2. Win Rate

- **What it measures**: The percentage of episodes where the agent defeats the opponent.

- **Importance**: Reflects the agent's ability to achieve the primary goal of winning. Also, it evaluates the win rate against different types of opponents such as random or other pre-trained models.
- **How to achieve**: The final outcome during all the matches during testing time would be tracked in the game loop.

3. Time to Win

- **What it measures**: The average time (or steps) it takes for the agent to win a game.
- **Importance**: Faster wins suggest efficient strategies. This metric is monitored to assess if the agent's strategies become more efficient over time and to compare with the opponent to see which one wins the game fast.
- **How to achieve**: The no. of steps taken during all the matches by the model during testing time would be tracked in the game loop.

4. Action Efficiency

- **What it measures**: The ratio of meaningful actions, in this case attack actions, to total actions taken by the agent.
- **Importance**: Indicates whether the agent is making purposeful decisions rather than random or redundant actions. Also, this metric can provide insights into the agent's playstyle, helping to refine its balance between aggression, movement and defence.
- **How to achieve**: The no. of successful attack actions and total no. of actions taken during all the matches at testing time would be stored in the game loop and then the ratio would be calculated.

  Formula to be used: Action Efficiency = $\frac{Number\ of\ Successful\ Attack\ Actions}{Total\ number\ of\ actions}$

5. Elo Ratings

- **What it measures**: Elo ratings measure relative skill levels between players in zero-sum games. The system assigns a numerical rating to each player, which updates based on the outcomes of matches. A higher rating indicates a stronger player.
- **Importance:** It encourages evaluation against a variety of agents and provides a single, interpretable metric to compare different models.
- **How to achieve:** The outcomes for both players are calculated based on their current ratings using the Elo formula. After a match, the actual scores (win, loss, or draw) are compared to the expected outcomes, and the ratings are adjusted by adding the product of the K-factor and the score difference (score−expected outcome). This ensures that a zero-sum change is maintained, meaning one player's rating gain is balanced by the other's loss, reflecting their relative performance in the match.

3.5.2. Testing

After the training of models is done, evaluating the metrics in various testing scenarios ensures that the RL agent (Player 1) is robust and performs well across a range of opponents. The following are a few testing scenarios to be implemented:

1. Player 1 Against Idle Player:

Setup: The opponent (Player 2) does not perform any actions and remains idle.

Purpose: Test Player 1's basic ability to take advantage of an unchallenged environment and assess how effectively Player 1 uses attack opportunities and optimises rewards without interference. The agent should ideally learn to move towards player 2 and beat him to defeat.

2. Player 1 Against Random Player:

Setup: Player 2 performs random actions without strategy or coherence.

Purpose: Evaluate Player 1's ability to handle unpredictable opponents and test the agent's ability to distinguish between useful and irrelevant actions by Player 2 and learn the counter attacks.

3. Player 1 Against Trained Player 2:

Setup: Player 2 is a trained RL agent with its own optimized policy.

Purpose: To evaluate Player 1's performance against skilled, strategic opponents and test the generalizability of Player 1's policy when competing against another optimized agent.

4. Player 1 Against Humans:

Setup: Human players of varying skill levels play against Player 1.

Purpose: To test Player 1 in real-world scenarios with complex, adaptive, and strategic opponents. Also to validate the agent's ability to generalize learned strategies to human behaviour.

4. Implementation

During this process, the methodology is implemented in various stages. With almost every stage, the complexity of the environment or gameplay is increased. So, in the initial stages, the aim was to achieve the desired results (successful reinforcement) in the simplest of the environments. Once achieved, then more challenges are introduced.

4.1 Prototyping

Before moving to the actual problem, reinforcement learning was implemented in simpler problems, which were Cartpole and Lunar Lander. In both cases, the experiments were done with DQN and PPO algorithms. This was done to get hands-on experience working with RL libraries, how to correctly train RL agents and to get a better understanding of all the hyperparameters involved, like learning_rate, gamma etc., and learn how these parameters impact the learning process. By doing this, it became much easier to analyse and debug the implementation of RL in the actual game.

4.1.1. Cartpole

4.1.1.1. Problem:

(The following problem is same as that in proposal [16])

The task is to balance a pole that is attached by a revolute joint to a cart, which moves along a frictionless track. The system is controlled by applying forces to the cart in either the left or right direction. The objective is to prevent the pole from falling over while keeping the cart within the track limits. The pole starts nearly vertical, and the goal is to maintain it upright for as long as possible.

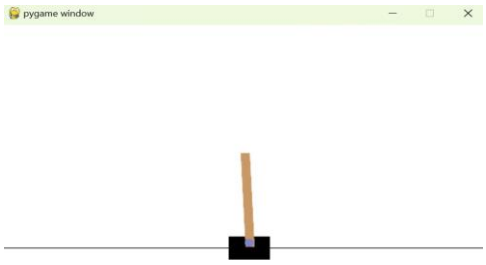


Figure 7. Cartpole game

4.1.1.2. Environment:

The Environment used is the OpenAI Gym environment CartPole-v1.

Action space: {0,1}, where 0 denotes pushing cart to the left and 1 denotes pushing to right.

State Space: {0,1,2,3}

Num	*Observation*	*Min value*	*Max value*
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf

2	Pole Angle	-0.418 rad (-24°)	+0.418 rad (24°)
3	Pole Angular Velocity	-Inf	Inf

Table 3. Observation space for cart-pole

Reward: The agent receives a reward for each timestep that the pole remains upright. The reward is +1 for every timestep where the pole does not fall over and the cart stays within the track boundaries.
Episode Termination: An episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the centre of the track, or a specified time limit (500 timesteps in this case) is reached.

4.1.1.3. Aim and Process

Aim of RL agent: To achieve the maximum reward of 500.

4.1.1.3.1. Process 1- (Using DQN and keras-rl2):

(The following subsection is same as that in proposal [16])

1. A gym environment wrapper is made on top of Gym environment for cartpole.

```
class GymWrapper(gym.Wrapper):
    def __init__(self, env):
        gym.Wrapper.__init__(self, env)
        self.env = env

    def reset(self, **kwargs):
        obs, _ = self.env.reset(**kwargs)
        return obs

    def step(self, action):
        # print(self.env.step(action))
        obs, reward, done, info, _ = self.env.step(action)
        return obs, reward, done, {}

    def render(self, mode='human'):
        if 'mode' in self.env.render.__code__.co_varnames:
            return self.env.render(mode=mode)
        else:
            return self.env.render()


env = GymWrapper(gym.make('CartPole-v1'))
```

Figure 8. Code for Gym wrapper for Cartpole

2. A sequential deep q-network model is made. The state is fed into a neural network. The network processes the state through several layers, and outputs a Q-value for each possible action (push left or right).

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 4)                 0

 dense (Dense)               (None, 512)               2560

 dense_1 (Dense)             (None, 256)               131328

 dense_2 (Dense)             (None, 64)                16448

 dense_3 (Dense)             (None, 2)                 130

=================================================================
Total params: 150466 (587.76 KB)
Trainable params: 150466 (587.76 KB)
Non-trainable params: 0 (0.00 Byte)
```

Figure 9. Sequential model's architecture

3. A DQN agent is then built using Keras-rl DQNAgent in which the Q-network and actions are fed. Along with this, a sequential memory is initialized for replay buffer and policy method (BoltzmannQPolicy) is also given. The policy decides whether to choose the best action (exploitation) based on the highest Q-value or a random action (exploration).

```
policy = BoltzmannQPolicy()
memory = SequentialMemory(limit=50000, window_length=1)
dqn = DQNAgent(model=model, memory=memory, policy=policy,
               nb_actions=actions, nb_steps_warmup=10, target_model_update=1e-2)
```

Figure 10. Code for agent creation

4. During the training of the DQN agent, a selected action is executed in the environment by the agent. The environment returns observations such as the new state, reward for the action, and whether the episode has ended (done). The episode observations (current state, action, reward, next state, done) are stored in the replay memory, a database of past experiences. For the next action, the agent randomly samples a batch of experiences from the replay memory and decides the action based on these past experiences. After the reward is attained for the next action, the loss is calculated, and weights are updated via backpropagation by the Adam optimizer based on the mean absolute error. This entire process was repeated for 50000 steps.

```
dqn.fit(env, nb_steps=50000, visualize=False, verbose=2, callbacks=[reward_callback],nb_max_episode_steps=500)
```

Figure 11. Code for learning

Evaluation and Result:

The model is evaluated against a random player for 100 episodes after training for 50,000 steps.

Results:

1. Random player:

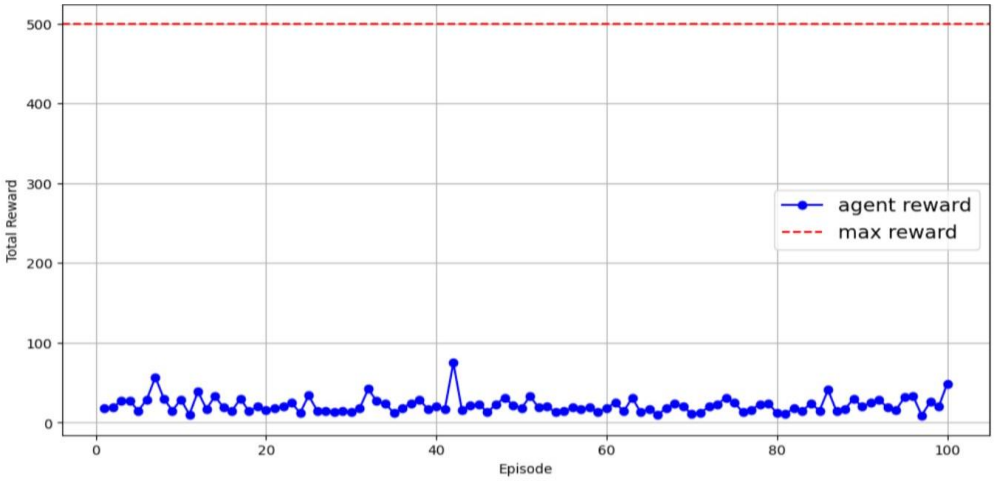


Figure 12. Testing rewards for random player for Cartpole

The random player could never get a reward of more than 100 and was far below the red line which represented the maximum reward.

2. Agent during training time of 50,000 steps:

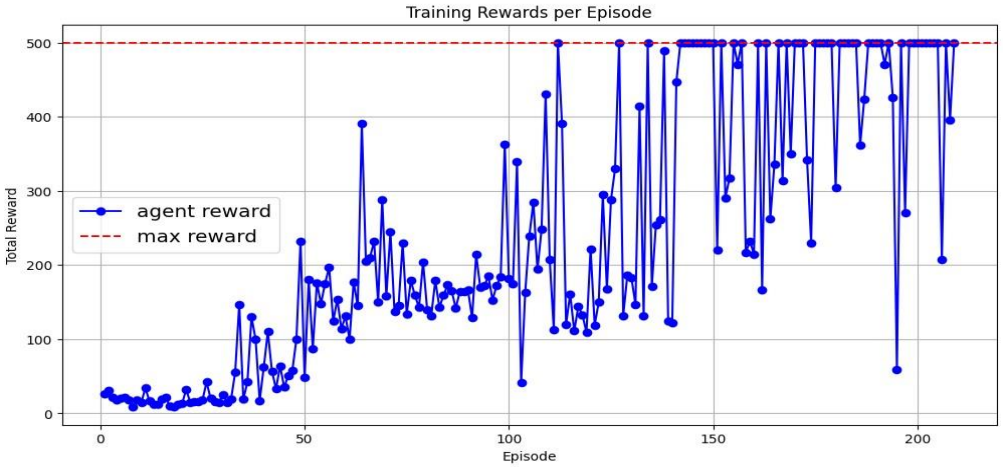


Figure 13. Training progress for Cartpole

By each episode the total reward seems to be increasing with slight variations and was able to achieve maximum reward of 500 (shown in red line) towards the end, which shows that the model did learn.

3. Agent during test of 100 episodes:

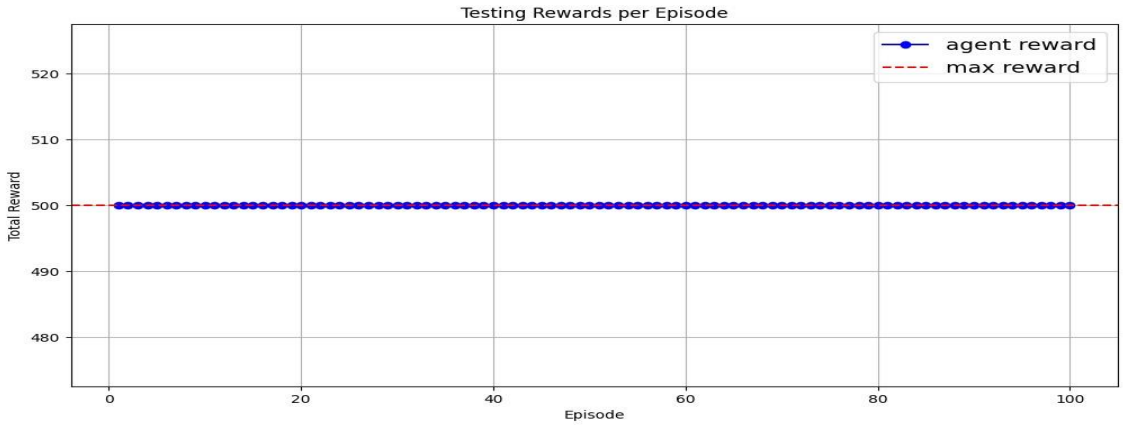


Figure 14. Testing rewards for Cartpole

Agent successfully achieves the maximum reward of 500 in each of the 100 episodes.

4.1.1.3.2. Process 2- (Using PPO and stable_baseline3):

1. Creating an Open-AI gym environment for Cartpole using: env = gym.make('CartPole-v1')
2. Initializing the model using:
model = PPO('MlpPolicy', env, verbose=1)
3. Creating a callback function to store rewards as this would help in monitoring the progress during the training and for visualization purposes.

```
class RewardLoggingCallback(BaseCallback):
    def __init__(self, verbose=0):
        super(RewardLoggingCallback, self).__init__(verbose)
        self.episode_rewards = []
        self.current_episode_reward = 0

    def _on_step(self) -> bool:
        self.current_episode_reward += self.locals['rewards'][0]
        if self.locals['dones'][0]:
            self.episode_rewards.append(self.current_episode_reward)
            self.current_episode_reward = 0

        return True

    def get_rewards(self):
        return self.episode_rewards
```

Figure 15. Code for reward callback

4. Training the model for 100000 steps using:
model.learn(total_timesteps=timesteps,
reset_num_timesteps=False,callback=reward_callback)

Evaluation and Result:

Agent during the training time:

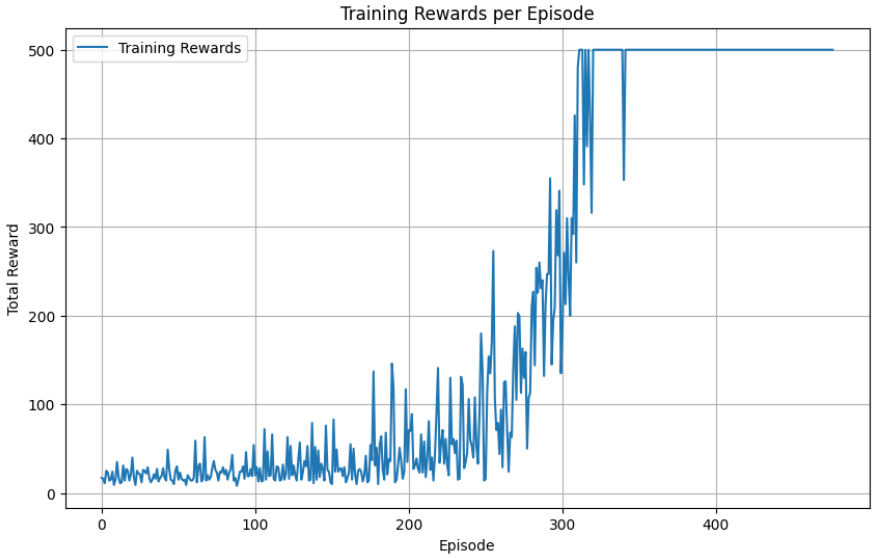


Figure 16. Training progress for Cartpole (PPO)

Just like before, by each episode the total reward seems to be increasing with slight variations and was able to achieve maximum reward of 500 towards the end, which shows that the model did learn.

Agent during the testing period of 100 episodes:

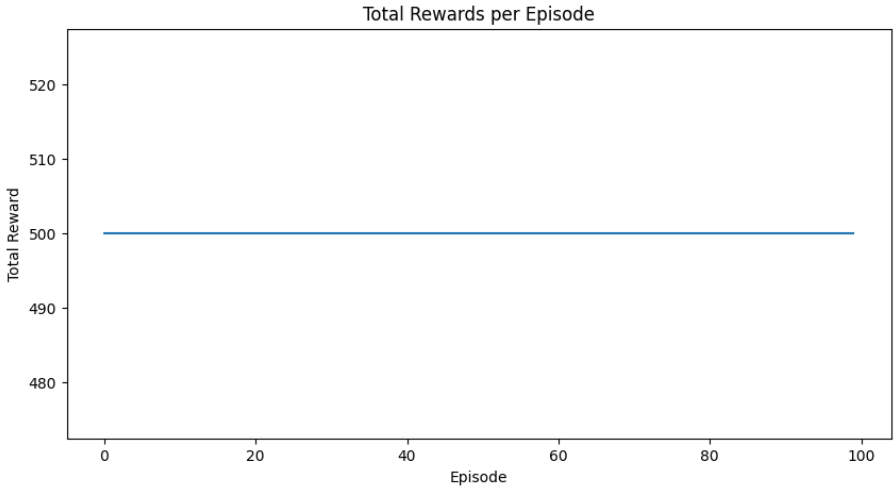


Figure 17. Testing rewards for Cartpole (PPO)

The agent is successfully able to learn and achieve the maximum reward of 500 in every match.

4.1.2. Lunar Lander

4.1.2.1. Problem:

Lunar Lander is a popular RL problem which simulates the landing of a spacecraft. The task is to control a lunar lander spacecraft and guide it to land safely on a designated landing pad on the surface. The goal is to maximize the total reward by landing softly without crashing, staying close to the landing pad and using fuel efficiently.

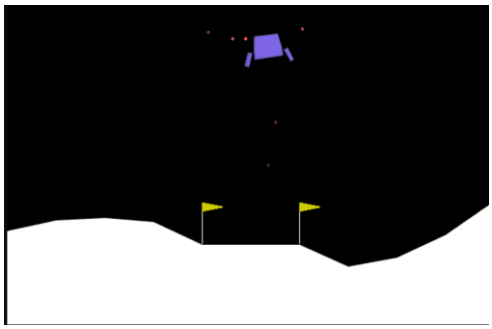


Figure 18. Lunar Lander game

4.1.2.2. Environment:

Environment used is the OpenAI Gym environment LunarLander-v2.

Action space: There are four discrete actions available: do nothing, fire left orientation engine, fire main engine, fire right orientation engine.

State Space: The environment provides an 8-dimensional vector of continuous values describing the lander's current state using: The lander's horizontal position, lander's vertical position, horizontal velocity, vertical velocity, the angle of the lander, angular velocity of the lander, a binary flag indicating if the left leg is touching the ground and a binary flag indicating if the right leg is touching the ground.

Num	*Observation*	*Min Value*	*Max Value*
1	Horizontal Position (x)	-1.5	1.5
2	Vertical Position (y)	-1.5	1.5
3	Horizontal Velocity	-5.0	5.0
4	Vertical Velocity	-5.0	5.0
5	Angle (rotation of the lander)	-3.1415927	+3.1415927
6	Angular Velocity	-5.0	5.0
7	Left Leg Contact	0	1
8	Right Leg Contact	0	1

Table 4. Observation space for Lunar Lander

Reward:

A positive reward for is given for:

- Landing on the pad.
- Having its legs touch the ground safely.
- Reducing the velocity for smooth landing.

A negative reward if given for:

- Crashing.
- Drifting too far from the landing pad.
- Using excessive fuel

Also, a successful landing gives a significant bonus of 200 points.

Episode Termination: An episode ends if the lander crashes, successfully lands on the pad or if the time is up.

4.1.2.3. Aim and Process

Aim of RL agent: To achieve the maximum reward and solve the problem.

4.1.2.3.1. Process 1- (Using DQN and stable_baselines3):

1. Creating the gym environment using: gym.make("LunarLander-v2")

```
model = DQN(
    "MlpPolicy",
    env,
    verbose=1,
    gamma=0.98,
    learning_rate=5e-4,
    batch_size=32,
    buffer_size=15000,
    exploration_final_eps=0.0989,
    target_update_interval=250,
```

Figure 19. DQN model initialization

2. Initializing the model and setting hyperparameters as shown in figure.
3. Define a reward callback function as shown earlier to track training progress.
4. Train the agent for 200000 steps using:
model.learn(total_timesteps=200000,callback=reward_callback)

Evaluation and Result:
The model is evaluated against a random player for 100 episodes after training for 200,000 steps.

Results:
1. Random player:

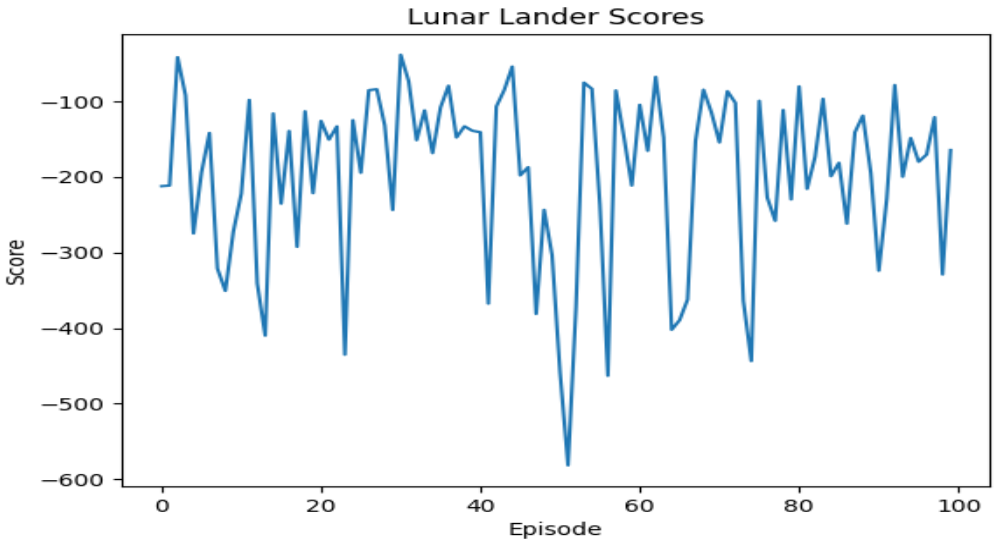


Figure 20. Random player rewards for Lunar Lander

The Random player was never able to achieve a positive reward.
2. Agent during training time of 200,000 steps:
Initially, the model was getting high negative rewards but gradually was able to receive positive rewards for almost all the episodes which shows that the model had been learning.

Figure 21. Training rewards for Lunar Lander (DQN)

3. Agent during test of 100 episodes:

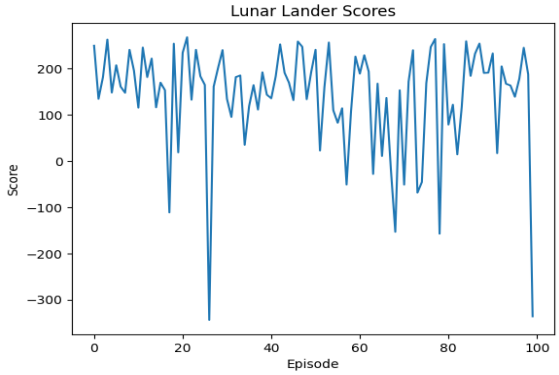


Figure 22. Testing rewards Lunar Lander (DQN)

The agent most of the time is able to achieve a high score with a mean of 140.569 and a standard deviation of 116.669 and is able to beat random player by a significant margin.

4.1.2.3.2. Process 2- (Using PPO and stable_baselines3):

1. Creating an Open-AI gym environment using:
env = gym.make('LunarLander-v2')
2. Initializing the model using:
agent = PPO('MlpPolicy', env, ent_coef=0.09,gamma=0.99, verbose=1,learning_rate=0.001)
3. Creating a callback function to store rewards as this would help in monitoring the progress during the training and for visualization purposes.
4. Training the model for 200000 steps using:
agent.learn(total_timesteps=200000,callback=reward_callback)

Evaluation and Result:

Agent during the training time: In the starting, the model was getting high negative rewards but gradually was able to receive positive rewards for almost all the episodes which shows that the model had been learning.

Figure 23. Training rewards for Lunar Lander (PPO)

Agent during the testing period of 100 episodes: Rewards are positive most of the times which is good compared to random player.

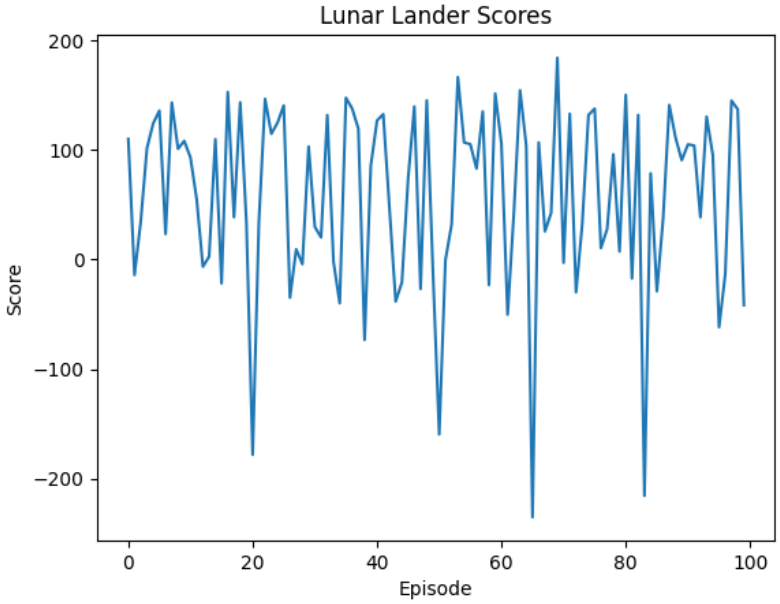


Figure 24. Testing rewards for Lunar Lander (PPO)

The agent is successfully able to learn and achieve a high positive reward most of the time beating the random player. Mean: 57.893, Std. Deviation: 83.632

4.1.3. Observation

For Cartpole, both the techniques were fairly good, but DQN started to converge much earlier than PPO. For Lunar Lander, the results show that DQN outperforms PPO in terms of mean reward (140.569 vs. 57.893), indicating it learns a better policy for safely landing the spacecraft. However, DQN has a higher standard deviation (116.669 vs. 83.632), reflecting less consistent performance across episodes. In contrast, PPO achieved lower rewards on average which demonstrates more stable and predictable behaviour. This suggests it is less prone to extreme successes or failures. Overall, PPO seems to be a more stable algorithm, and DQN, on the other hand, is able to achieve a higher reward but it isn't as stable.

4.2 Setup and Environment Customization

4.2.1. Setup

1. To make the game accessible via Python several components were added to the Java game engine. Since building the environment would need to have access to action space and observation space, getter and setter methods were created on the Java end to make it available by call in Python. For example, to get player 1's actions, getPlayer1Controls() method was created which would return an array of player 1's actions

More methods like this were created to access health, position etc. for both the players.

Similarly for observation space getObs() method is created in the Player class which returns an array with all the necessary information that could be required for environment creation like velocity in x-axis, x-position, health, distance between players, state of players- attacking or idle for both the players, as shown in figure.

```
public float[] getObs() {
    float[] observation = new float[11];  // Adjusted to hold info for both players

    // Player 1 (this) information
    observation[0] = (float) this.position.x;          // Player 1 position x
    observation[1] = (float) this.velocity.x;          // Player 1 velocity x
    observation[2] = this.health;                      // Player 1 health

    // Player 2 (the opponent) information
    observation[3] = (float) this.other.position.x;    // Player 2 position x
    observation[4] = (float) this.other.velocity.x;    // Player 2 velocity x
    observation[5] = this.other.health;                // Player 2 health

    // Distance between players
    observation[6] = (float) this.position.dist(this.other.position); // Distance between players

    // Player 1 states
    observation[7] = (this.states.attacking ? 1.0f : 0.0f); // Player 1 attacking state

    // Player 2 states
    observation[8] = (this.other.states.attacking ? 1.0f : 0.0f); // Player 2 attacking state

    // Player 1 Standing Attacks
    observation[9] = getStandingAttackState(this.states.standingLight, this.states.standingMedium,
            this.states.standingHeavy, this.states.crouched); // Encoded Player 1 standing state

    // Player 2 Standing Attacks
    observation[10] = getStandingAttackState(this.other.states.standingLight,
            this.other.states.standingMedium,
            this.other.states.standingHeavy, this.other.states.crouched); // Encoded Player 2 standing state

    return observation;
}
```

Figure 25. Java code for GetObs function

2. To make the Players move in the game through Python, player1Move() and player2Move() methods in Java were created and on Python side, _handle_action(action) method was created. The latter method maps discrete RL actions to specific Java game controls which then invoke player1Move() or Player2Move() method that handles all the key events such as, key press , key release etc. which make the players act in the game.

3. To keep the game state synchronized with the RL loop for ensuring accurate feedback without skipping any state frames and enabling controlled interaction between the agent and the environment, a method requestAdvance() is created on Java side. This method is called inside the step function. This method sets a flag variable 'shouldAdvance' = True. The variable is used in the actual game loop and the game loop continues only if the value is True. Once the action is made in the game, the variable is set back to False. This way the environment and the game can communicate and be on the same page without loosing any data.

```
def step(self, action):
    self._handle_action(action)
    # Request Java to advance the game state
    self.game.requestAdvance()
    # After requesting, proceeding to get the new state
    obs = self._get_next_frame()
    reward = self._compute_reward()
    done = self._check_done()
    truncated = False
    self.cumulative_reward += reward
    return obs, reward, done, truncated, {}
```

Figure 26. Step function in environment class

4.2.2 Environment

A custom OpenAI Gym environment is created for the game with all main methods implemented with some additional methods. The components of the environment are:

1. Initialization (__init__): This is where the environment is set up and initialization is done for:

Action Space: Defines the possible actions Player 1 can take, such as moving left, right, or attacking, which could be light, medium, or heavy (5 discrete actions). The action space is much smaller than the actual no. of actions available to keep things simpler.

Observation Space: Specifies the range of values for various game parameters like player positions, health, and states in box datatype which is supported by Gym.

For example, Player 1's position can range from 0 to 1400, and their health ranges from 0 to 100 (refer table 5).

Num	*Observation*	*Min Value*	*Max Value*
1	Player 1 position x	0	1400
2	Player 1 velocity x	-600	600
3	Player 1 health	0	100
4	Player 2 position x	0	1400
5	Player 2 velocity x	-600	600
6	Player 2 health	0	100
7	Distance between players	0	1800
8	Player 1 state (idle, attacking)	0	1
9	Player 2 state (idle, attacking)	0	1
10	Player 1 attack states (light, medium, heavy, crouched)	0	4

11	Player 2 attack states (light, medium, heavy, crouched)	0	4

Table 5. Observation space for Street Fighter

Game Connection: Connects to the game using JavaGateway. If no game instance is passed, it tries to fetch one.
Health Tracking: Initializes variables to track the health of Player 1 and Player 2 for calculating rewards.

2. Simulating Key Presses (simulate_key_press): Sends a command to the game via the Java interface to simulate a key press (like pressing "Enter" or selecting a character).

3. Starting the Game Automatically (start_game_auto): Automates the game startup process by simulating key presses, starts the game menu and then selects Player 1 and Player 2 characters. This ensures the game begins in an automated way without manual intervention.

4. Step Function (step): The step function takes care of the majority of the tasks. A walkthrough of its steps:

1. Player 1 takes an action (e.g., move or attack).
2. The game advances to the next state.
3. Observations of the new state (e.g., positions, health) are collected.
4. Rewards are calculated based on health changes or other criteria.
5. Checks if the episode (game) is done (e.g., one player's health is 0).
6. Returns:
   - **Observations**: Details about the new game state.
   - **Reward**: A score for Player 1's action in this step.
   - **Done**: Whether the game is over.
   - **Truncated**: Placeholder for incomplete episodes (not used here).
   - **Info**: Extra details (left empty in this case).

5. Handling Actions (_handle_action): Maps Player 1's actions (e.g., move left, attack) to game controls. Executes the corresponding game command using Java methods.

6. Getting the Next Frame (_get_next_frame): Fetches the game state (like health, position, velocity) as an array of values.

7. Computing Rewards (_compute_reward): Rewards Player based on game outcomes:
Positive reward: When Player 2 loses health. (+1 initially)
Negative reward: When Player 1 loses health. (-1 initially)
Finally, updates health tracking variables for future reward calculations.

8. Checking if the Game is Done (_check_done): Returns true if the game ends. The game ends if:

1. Player’s health drops to 0.
2. The timer runs out.
3. A player wins based on rounds or scores.

9. Resetting the Environment (reset): Resets the game to its initial state, resets health, scores, and the game environment and returns the initial observation for a new episode.

10. Rendering (render): Placeholder for visualizing the game environment. (Not implemented though.)

4.3 Player 1 RL Agent Creation

In this section, all the stages that were implemented while creating the Player 1 would be mentioned. By every stage the level of difficulty for the agent would be increased.

4.3.1. Stage 1: Beating Idle Player

In this stage, the task is to beat Player 2 who would be standing still and doing nothing. The game is played in 'arcade' mode.

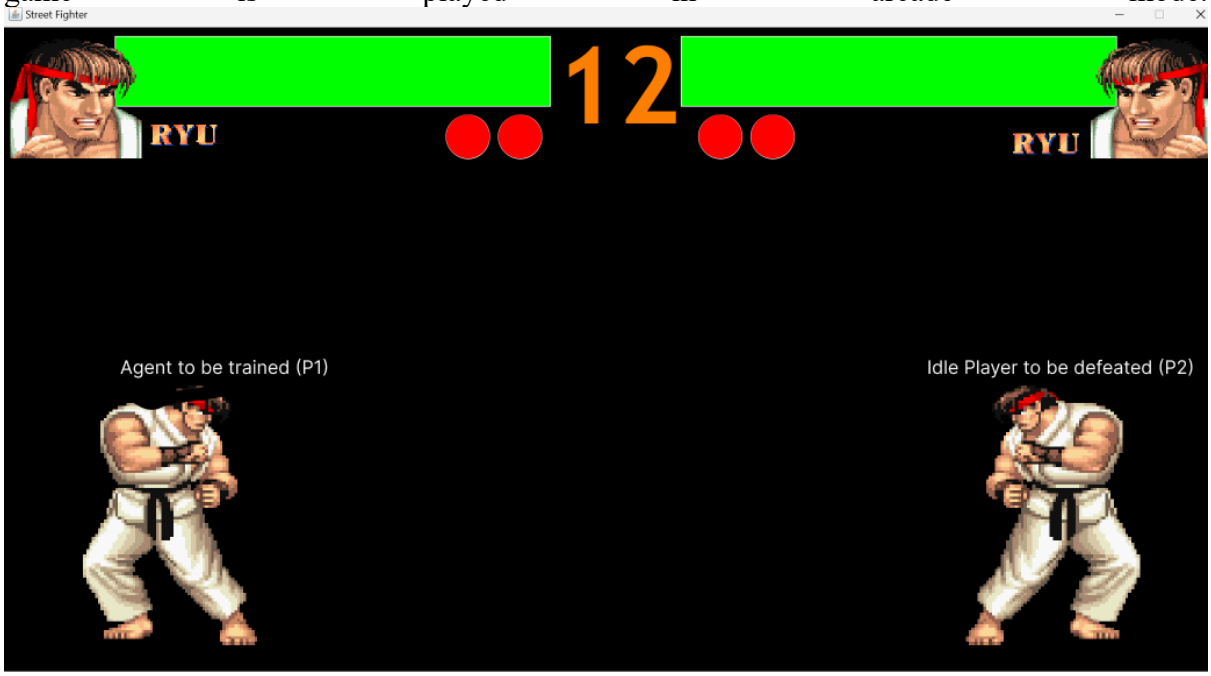


Figure 27. Stage 1 gameplay

4.3.1.1 Steps

1. Defining Custom Callback: RewardLoggingCallback

- This callback is used to log the total rewards the agent receives during training.
- **Key Components**:
  - **episode_rewards**: Stores the total rewards for each episode.
  - **_on_step**: Gets called after each environment step:
    - Adds the reward for the current step to the episode's total reward.
    - When the episode ends, the total reward is stored, and the counter is reset.
  - **get_rewards**: A helper function to access the logged rewards.

2. Registering the Environment

- The custom **StreetFighterEnv** is registered with Gymnasium using the register function.
- This step tells Gymnasium how to initialize and interact with the custom environment.

3. Setting Up the Game

- **JavaGateway** connects to the Java-based game and retrieves the game instance.
- The environment (**StreetFighterEnv**) is initialized with the game instance.

4. Environment Validation

- **check_env**: Ensures the custom environment complies with Gymnasium's standards (e.g., action and observation spaces are correctly defined).

5. Model Setup: PPO is initialized with the following parameters:

- MlpPolicy: The policy architecture is a multi-layer perceptron (MLP).
- learning_rate=0.0003: Controls how quickly the model updates weights.
- gamma=0.99: Discount factor for future rewards.

- ent_coef=0.01: Encourages exploration by penalizing deterministic policies.
- verbose=2: Provides detailed logs during training.

6. Training: The model is trained for 10000 steps using: model.learn(total_timesteps=num_timesteps, callback=reward_callback)

4.3.1.2 Results and Observations

During training process (fig. 28), in the beginning of it the agent was able to get a reward of 11 on average with a lot of fluctuation which could be indicating the agent's initial exploration and learning phase. However, towards the end it seems to be attaining average reward of 13, with minimum being 12 which shows a steady improvement in the agent's performance over the episodes.

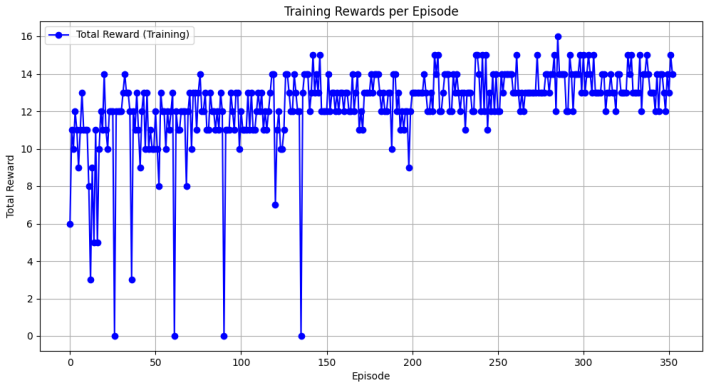


Figure 28. Training rewards for stage 1

The agent was made to play 10 games during the testing process. The testing plot demonstrates a consistent performance of the trained agent. as indicated in fig. 29. The agent gets a high reward of 17 in every match, which is much higher than 0 (min. reward attained during training). In the test matches, it could be clearly seen that the agent had learnt to move towards Player 2 and then knock him out completely till all his health becomes zero.

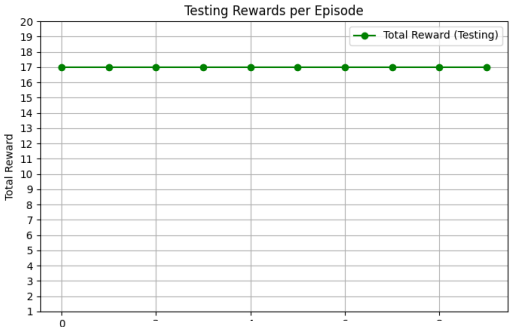


Figure 29. Testing rewards for stage 1

Result: Agent successfully beats idle player.

4.3.2. Stage 2: To Beat Built-In Random player

In this stage, the idle player would be replaced by the built-in player (who takes random actions) that is made available as player 2 in the 'story' mode of the Java game. The task is to beat him more no. of times than a random Player 1 (Player 1 taking random actions) would.

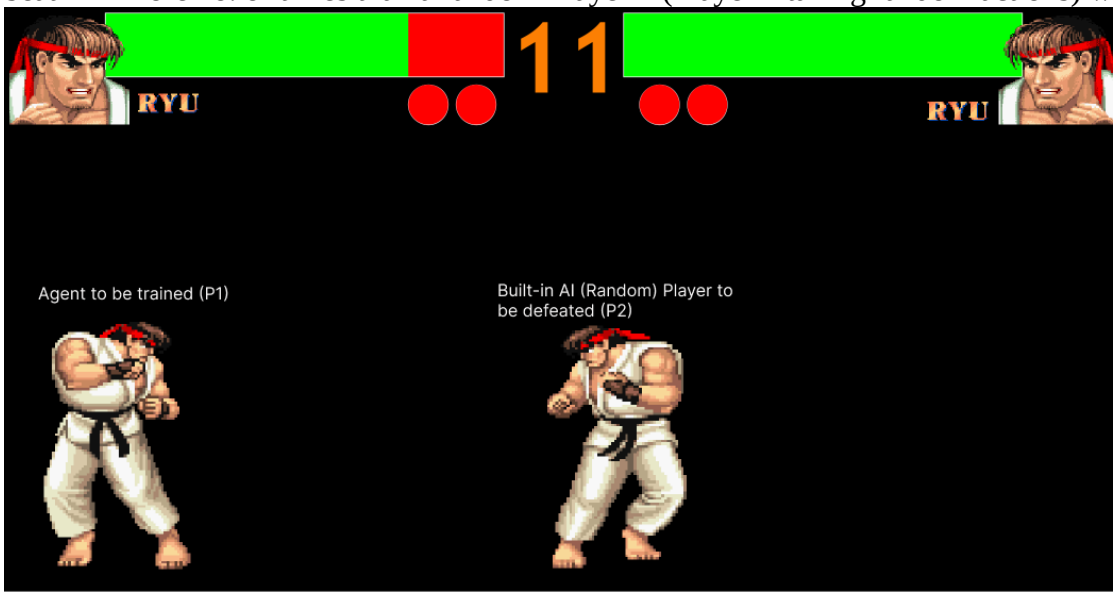


Figure 30. Stage 2 gameplay

4.3.2.1. Steps:

The same steps with the same setup as the previous stage were used, except this time the game is in story mode instead of arcade, and the training timesteps = 100,000 instead of 10,000. This is because the complexity of the problem has increased, and the model would need more time to learn and converge.

4.3.2.2. Results and Observations:

The training plot (fig. 31) shows the total reward per episode during the training process. The upward trend in the rewards indicates that the agent is learning to improve its performance over time, as evidenced by the higher rewards achieved in later episodes. However, the fluctuations throughout training suggest variability in the agent's performance, possibly due to the exploration-exploitation trade-off or environmental complexities. Despite the noise, the overall increasing trend signifies successful training progress and the agent's ability to adapt to the environment.

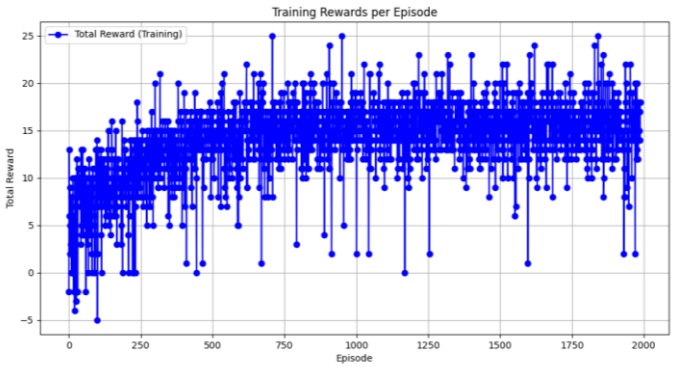


Figure 31. Training rewards stage 2

The model was tested for 100 matches, and it won 98% of the time as shown in fig. 32. The green bar indicates the total wins of P1 out of 100 and the red denotes the lost matches. A random player was made to play same no. of times against the built-in random player and managed to win 64% of the times as shown in fig. 33.

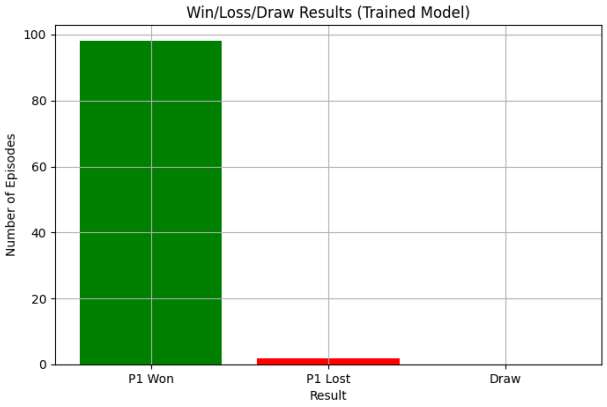


Figure 33. Player 1 test results for win, loss or draw

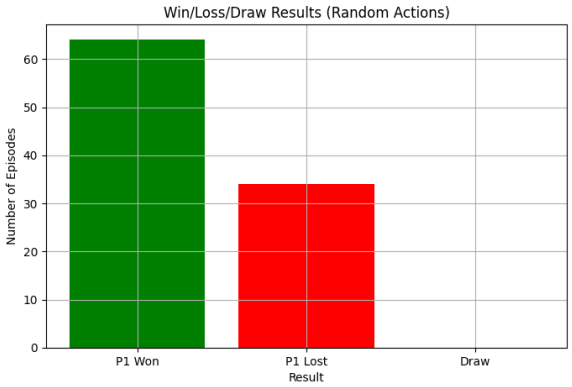


Figure 34. Random player test results for win, loss or draw

The rewards comparison plot in fig. 34 however shows that even though random player won 65% of the times, its reward was still too low. The consistent gap between the two lines highlights the superior performance of the trained model compared to random actions, validating the training process and the model's ability to solve the task effectively.

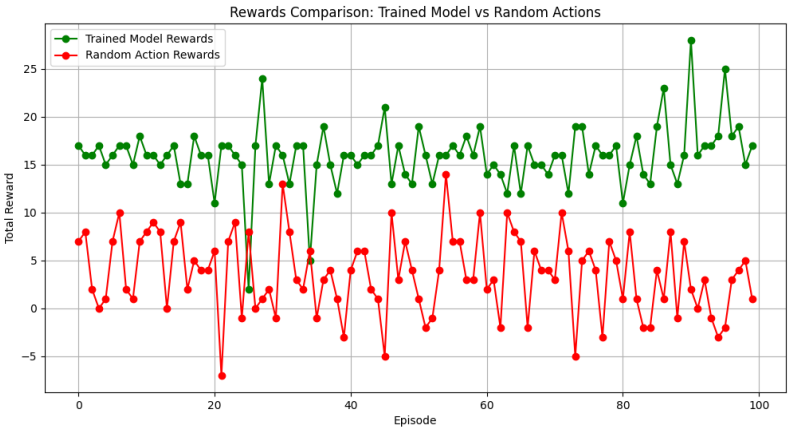


Figure 32. Rewards comparison between random player and agent

Result: The agent is successfully trained to beat a random player. However, the bot just punches (light attack) most of the times and do not use other attacks. A reason for this could be that reward is always +1 whenever the other player loses health, instead of actually capturing how much is the reduction in health.

Note: Random vs Random wasn't exactly 50-50 in terms of win rate as one might expect as shown in fig. 33.

4.3.3. Stage 3: To Beat Built-In Random player with more actions

This stage is very much similar to the previous stage. However, there would be crouching introduced as well. Also, to handle the problem of agent using just one kind of attack the reward would be reshaped as well.

4.3.3.1. Steps

1. Modification in action space:

self.action_space = spaces.Discrete(6)

Actions: 0=Left, 1=Right, 2=Light Attack, 3=Medium Attack, 4=Heavy Attack, 5=Crouch

Now there are total of 6 actions.

2. Modified reward function:

Calculating Health Differences: It calculates the change in health for both players since the last frame:

- p2_health_diff: How much Player 2's health decreased.
- p1_health_diff: How much Player 1's health decreased.
- Positive values indicate health loss for the respective player.

```
def _compute_reward(self, max_health=100):
    # Get current health values for both players
    p1_health = self.game.getPlayer1().getHealth()
    p2_health = self.game.getPlayer2().getHealth()

    # Calculate the health difference since the last frame
    p2_health_diff = self.prev_p2_health - p2_health  # Positive if player 2 lost health
    p1_health_diff = self.prev_p1_health - p1_health  # Positive if player 1 lost health

    # Normalize the health differences by the maximum health
    p2_health_diff_normalized = p2_health_diff / max_health
    p1_health_diff_normalized = p1_health_diff / max_health

    # Reward calculation with normalized values
    reward = 0
    reward += p2_health_diff_normalized  # Reward for reducing player 2's health
    reward -= p1_health_diff_normalized  # Penalty for player 1's health reduction

    # Update previous health values for next calculation
    self.prev_p1_health = p1_health
    self.prev_p2_health = p2_health
```

Figure 35. _compute_reward function in environment class

Normalizing Health Differences: The health differences are normalized by dividing by max_health (default: 100). This ensures the reward values remain consistent across different scales of health and lie between 0 and 1. By doing this, the intention is to promote more efficient and effective learning.

Rest of the training steps are the same as previous stage.

4.3.3.2. Results and Observations:

This training plot (fig. 36) shows a steady increase in total rewards over episodes, indicating that the agent is learning and improving its performance. The fluctuations represent exploration and variability during training, which gradually stabilizes. By normalizing the convergence appears to be better than before.

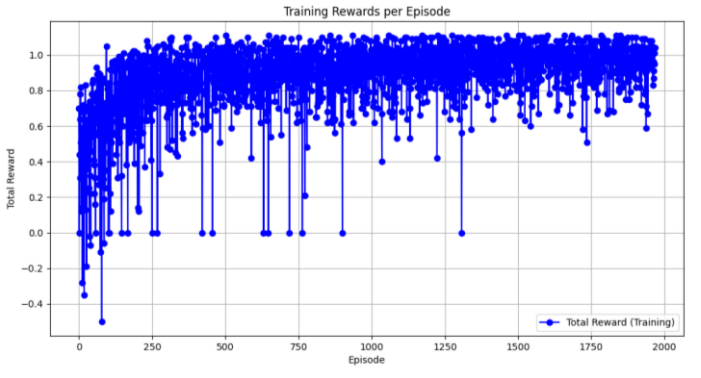


Figure 36. Stage 3 training rewards

During the testing time of 100 matches, the newly trained agent was able to win every single time (as shown in fig. 38) that too with a greater margin than before as the average reward of the current model (depicted by the line in the 1st box in fig. 37) is around 1 which is the maximum reward value. This states that P1 rarely got hit in any of the matches. Also, a comparison was drawn for the three approaches: the current model, the old model, and random actions. The current model shows consistently high rewards with minimal variation, indicating strong and stable performance. The old model (trained in previous stage) achieves moderate rewards with some variability,

suggesting decent but less reliable performance. Random actions have the lowest rewards and a wider spread, reflecting poor and inconsistent behaviour.

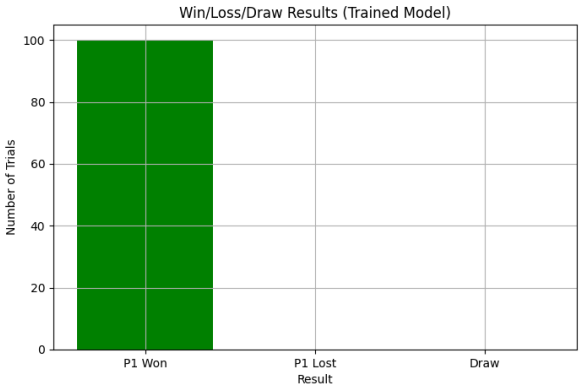


Figure 38. Player 1 test results for win, loss or draw (stage 3)

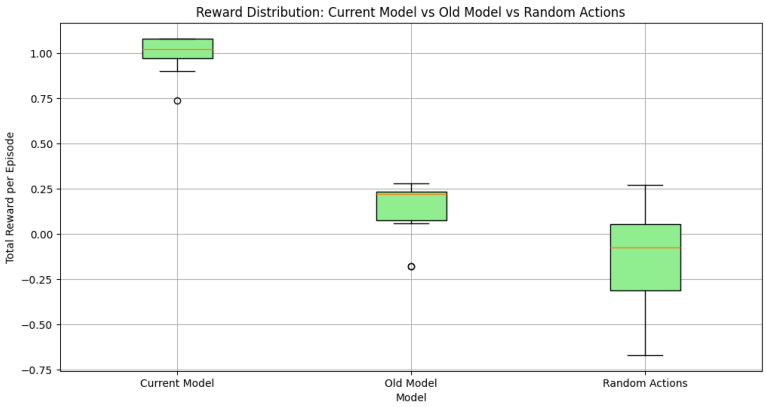


Figure 37. Reward comparison Current vs. Old vs. random

Result: The agent is <u>successfully trained</u> to beat a random player. The agent is now able to use all the actions. If a player is far away from agent, then the agent would maintain a safe distance and keep doing light attacks (punching). As the opponent starts to come nearer, the bot starts medium and heavy attacks based on the proximity. The agent was however mostly seen to be taking an aggressive approach and not crouching or dodging from opponent and prefers to attack instead.

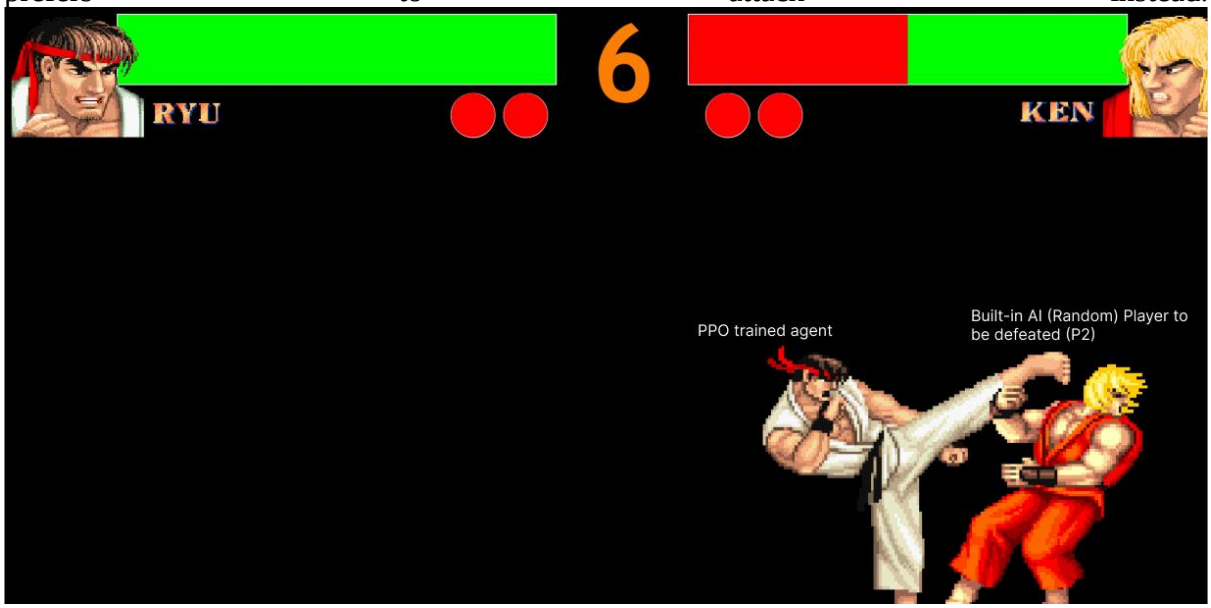


Figure 39. Picture showing agent beating Built-in Player 2

4.4 Player 2 RL Agent Creation

In the previous section, a PPO agent was successfully trained as Player 1. In this part, all the attempts made to create another agent as Player 2 that would be able to beat the previously trained Player 1 would be discussed.

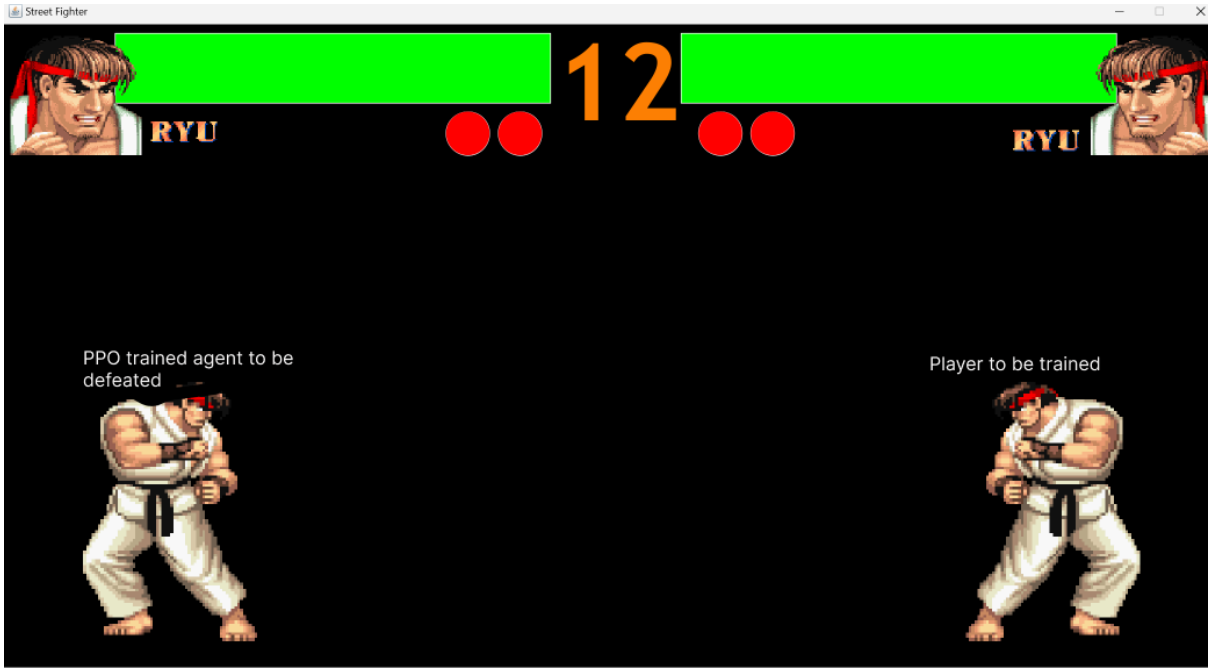


Figure 40. gameplay between PPO agent vs. new model to be trained

4.4.1. Stage 1: Training Player 2 using PPO and A2C

4.4.1.1. Steps

1. Just like how functions were created to handle actions of Player 1, the functions must handle Player 2 as well so that the Player 2 can access the environment. The function _handle_action was modified to take care of that by adding a new argument called player whose value can be 1 or 2. If it's 1 then it calls getPlayer1Controls, maps the current action and calls player1Move with that action. If the value is 2, then uses getPlayer2Controls and player2Move functions of Java game.

2. In the __init__ method of the environment, the Player 1's model is loaded and is to be referred to as ppo_old: self.ppo_old = PPO.load("ppo_street_fighter.zip ").

3. This model is called in the step function of the environment to predict Player 1's actions. At the same time, the function takes in the action for player2 as input and calls _handle_action function.

4. Since beating PPO would be much harder than a random player, it would be better if the environment could be simplified a bit more. To do so, the observations were normalized too so that it would be easier for the new model to converge just like how it was observed when normalizing the regards in previous stage. The new observation space is now defined as shown in table 6. Encoded values, however, would not be normalized and the _get_next_frame method was changed to normalize the remaining values by dividing them by their maximum value to keep them in the defined range.

```
def step(self, action_p2):

    self.current_action = action_p2  # Stores the current action
    self._handle_action(action_p2, player=2)  # Handles Player 2's action
    self.game.requestAdvance()  # Advances the game state
    obs_p1 = self._get_next_frame()  # Get the observation after Player 2's action
    action_p1, _ = self.ppo_old.predict(obs_p1, deterministic=True)  # PPO action
    action_p1 = int(action_p1)  # Convert action to integer
    self._handle_action(action_p1, player=1)  # Handle Player 1's action
    self.game.requestAdvance()  # Advance the game state after Player 1's action
    obs = self._get_next_frame()
    reward = self._compute_reward()

    # Check if the episode is done
    done, result = self._check_done()
    truncated = False

    self.cumulative_reward += reward
    info = {"result": result} if result is not None else {}

    return obs, reward, done, truncated, info
```

Figure 41. Step function altered to handle actions for pretrained agent

```
self.observation_space = spaces.Box(
    low=np.array([
        0,  # Relative Player 1 position x (min)
        -1,  # Relative Player 1 velocity x (normalized)
        0,  # Player 1 health (normalized)
        0,  # Relative Player 2 position x (min)
        -1,  # Relative Player 2 velocity x (normalized)
        0,  # Player 2 health (normalized)
        0,  # Normalized Distance between players
        0, 0,  # Player 1 and Player 2 states (e.g., idle, attacking)
        0,  # Player 1 attack states: light, medium, heavy
        0  # Player 2 attack states: light, medium, heavy
    ]),
    high=np.array([
        1.0,  # Relative Player 1 position x (max normalized to 1)
        1.0,  # Relative Player 1 velocity x (max normalized to 1)
        1.0,  # Player 1 health (max normalized to 1)
        1.0,  # Relative Player 2 position x (max normalized to 1)
        1.0,  # Relative Player 2 velocity x (max normalized to 1)
        1.0,  # Player 2 health (max normalized to 1)
        1.0,  # Normalized Distance between players (max normalized to 1)
        1, 1,  # State encoding (0 = idle, 1 = attacking)
        4,  # Player 1 attack states: 0, 1, 2, 3
        4  # Player 2 attack states: 0, 1, 2, 3
    ]),
    dtype=np.float32
)
```

Figure 42. Normalized observation space declaration

Num	*Observation*	*Min Value*	*Max Value*
1	Relative Player 1 position x (normalized)	0	1

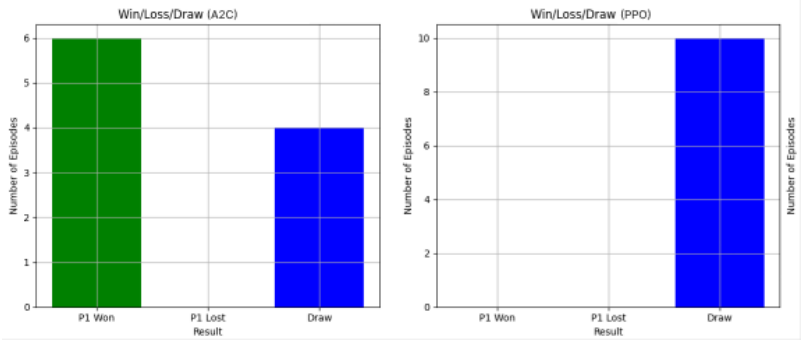
2	Relative Player 1 velocity x (normalized)	-1	1
3	Player 1 health (normalized)	0	1
4	Relative Player 2 position x (normalized)	0	1
5	Relative Player 2 velocity x (normalized)	-1	1
6	Player 2 health (normalized)	0	1
7	Normalized distance between players	0	1
8	Player 1 state (idle, attacking)	0	1
9	Player 2 state (idle, attacking)	0	1
10	Player 1 attack states (light, medium, heavy)	0	4
11	Player 2 attack states (light, medium, heavy)	0	4

Table 6. Normalized observation space

5. PPO and A2C algorithms were chosen, and respective models were trained.

4.4.1.2. Results and Observations

Figure 44 Win/Loss/Draw between P1 (PPO) vs. P2 (A2C) and P1 (PPO) vs. P2 (PPO)



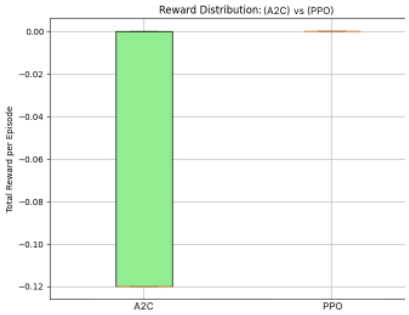


Figure 43 Rewards for A2C and PPO as Player 2

When testing, according to the reward plot (fig 43), A2C achieves negative rewards consistently, indicating poor performance, while PPO maintains near-zero rewards. None of them ever received a positive reward consistently. Looking at the win plots (fig. 44), PPO managed to make it a draw, suggesting it is neither aggressive nor capable of dominating its opponent.

Result: Both the models failed to beat Player 1 even once. Also, it was noticed that in both models, especially PPO, instead of going in to beat the player, they were just standing in their spot and settling for local minima (reward of 0). Moreover, Player 1 itself was avoiding the opponent and not attacking when fighting against PPO. The problem seems to be the same as Nash equilibrium as both players decide to stand in their initial positions and not attack one another till the time is up which leads to a draw.

Conclusion:

1. Some reward reshaping must be done to work around Nash equilibrium.
2. Also, beating Player 1 seems to be too hard a problem, and some simplifications must be done in the beginning stage of the training.
3. Promote higher rewards to get out of local minima.

4.4.2. Stage 2: Training Player 2 using DQN

4.4.2.1. Steps

To address the issues of the previous stage, the following steps were taken:

1. Reward reshaping:

a) **To get out of local minima:** Giving three times more positive reward when agent lands a successful hit but only 1/3 of a penalty if the hit is taken and giving a very high reward if agent wins the game. This should motivate the bot to attack more and get a higher reward.

b) **To avoid Nash equilibrium:** Giving a penalty if action taken is not an attack and a slightly higher penalty for a missed attack. This should encourage the bot to hit the opponent.

2. Simplifying the problem: To simplify the problem the agent is trained to beat only 50% of PPO and 50% of the times a random player. Once converged to get higher positive reward, the model is further trained against 80% PPO and 20% randomness. This was done by adjusting the p_random variable in the environment's step function. The variable p_random was assigned a value of 0.50 first and then to 0.20 and the actions of Player 1 were decided accordingly using:

```
if np.random.rand() < p_random:
    action_p1 = self.action_space.sample()
else:
    action_p1, _ = self.ppo_old.predict(obs_p1, deterministic=True)
```

This code implements an epsilon-greedy action selection strategy:

1. np.random.rand() generates a random number between 0 and 1
2. If this number is less than p_random, it chooses a completely random action using self.action_space.sample()
3. Otherwise, it uses the trained PPO model to predict the best action based on the current observation

In simpler terms:

- With probability p_random, do something random
- Otherwise, do what the AI model thinks is best

This helps balance exploration (trying new things) and exploitation (using learned knowledge) during training.

4.4.2.2. Results and Observations

Training plot for 50% randomness: A lot of fluctuations can be seen which shows that the model was struggling in terms of stability. However, by the end episodes, the average reward became more positive which states model did improve.

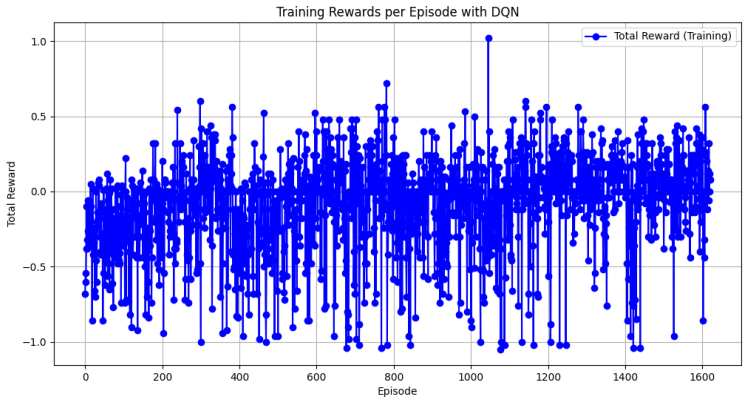


Figure 45. Training process for P2 against 50% randomness

Training plot for 20% randomness: After achieving a reward of 0, model seldom seems to be getting a positive reward. This could indicate that the model kept getting stuck in a local minimum, which means it settled for a suboptimal reward.

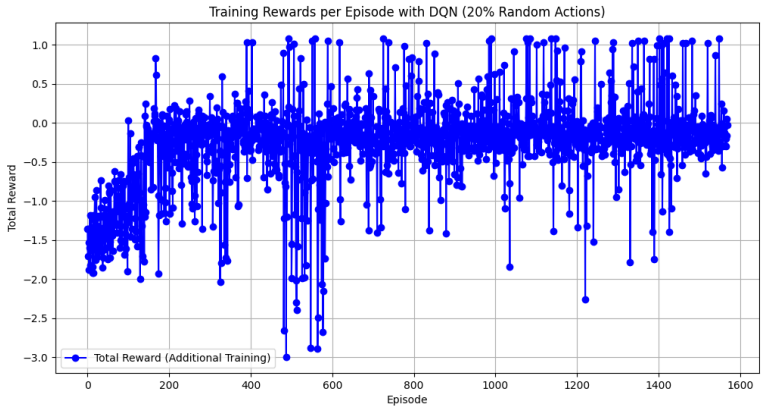


Figure 46. Training process for P2 against 20% randomness

Result: When fighting against 50% PPO, the model was able to receive higher positive rewards and was even able to win sometimes. However, when further trained to beat 80% PPO, then the model is seen to be stuck on 0 most of the time, again and never won.
Conclusion: The problem seems to be too hard to be solved and needs to be investigated if it can be solved at all.

4.4.3. Stage 3: Investigation

To check if beating the PPO agent was possible at all, some test matches were played against it by humans, and it was found that at a certain distance Player 1 can hit Player 2 but at the same distance player 2 was not able to hit Player 1. With some further checks, it could be declared that:

1. The player on the left would always have an upper hand.
2. The player 2 was getting frozen a lot of the times in the game while taking actions.

4.4.3.1. Fixing the game to introduce fair play

1. **Issue #1:** The hardcoded handling of xShift for left (+xShift) and right (-xShift) players created an asymmetry, making it possible for left side player to hit at a certain distance while the other one cannot.

Fix: A distance threshold was introduced. Once the horizontal distance is less than the threshold and the yOverlap is true, only then would an attack be landed. Also, the detectHit() method now dynamically adjusts xShift based on the player's facing direction (left or right).

```
protected void detectHit(int xShift, int yShift, int damage, int stun) {
    int otherY = (int) other.position.y;
    int otherHeight = other.hitboxHeight;
    if (other.states.crouched) {
        otherY += 100;
        otherHeight -= 100;
    }

    int xShiftAdjusted = xShift * facing + (facing == 1 ? -ATTACK_X_OFFSET : ATTACK_X_OFFSET);
    int attackXStart = (int) (position.x + xShiftAdjusted);
    int attackXEnd = attackXStart + attackHitboxSize;

    int attackYStart = (int) (position.y + yShift);
    int attackYEnd = attackYStart + attackHitboxSize;

    int otherXStart = (int) other.position.x;
    if (facing == -1) {
        otherXStart = (int) other.position.x + 30;
    } else {
        otherXStart = (int) other.position.x - 30;
    }

    int otherXEnd = otherXStart + other.hitboxWidth;
    int otherYEnd = otherY + otherHeight;

    double horizontalDistance = Math.abs(this.position.x - other.position.x);
    int distanceThreshold = 300;
    boolean xOverlap = attackXStart < otherXEnd && attackXEnd > otherXStart;
    boolean yOverlap = attackYStart < otherYEnd && attackYEnd > otherY;
    boolean withinDistance = horizontalDistance < distanceThreshold;

    if ((withinDistance && yOverlap) ) {
        states.landedAttack = true;
        handleAttack(damage, stun);
    }
}
```

Figure 47. modified detectHit function in Java

2. **Issue #2:** Player 2 was getting frozen because of multiple states being active at the same time inside the Controls class while handling the key press and release in the nested if-else conditions for Player 2 controls. E.g., for Player 2, at the same time walking_backward and walking_forward were set to true.
Fix: The code was debugged to handle the states as they should be.

Also, a pull request for the code changes had also been made and was merged in the original code. Link for pull request: https://github.com/drewberry612/street-fighter/pull/1

Note: There are still a few more issues in the game, but they don't affect the fair play of the game. For e.g., players get stunned for too long at times, etc.

4.4.3.2. Retrain Player 1

After the game fixes were applied, Player 1 PPO agent was retrained against Built-in random player of the game. The training plot (fig. 49) shows perpetual increase in rewards over the episodes stating successful learning, and reward plot (fig. 48) shows that the average reward is still higher than random and previous model. Also, previously trained model was able to achieve higher rewards (close to 1) but it is dropped by a lot now, which tells that the P1 definitely had an upper hand earlier but now the game seems to be a fair play.

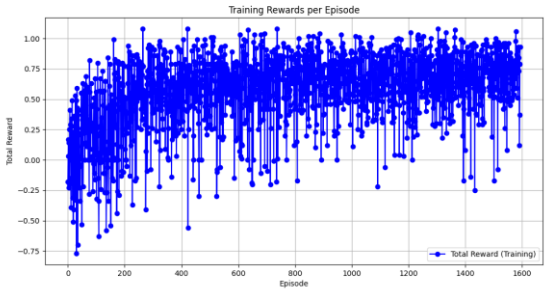


Figure 49. Training rewards for retrained PPO (P1)

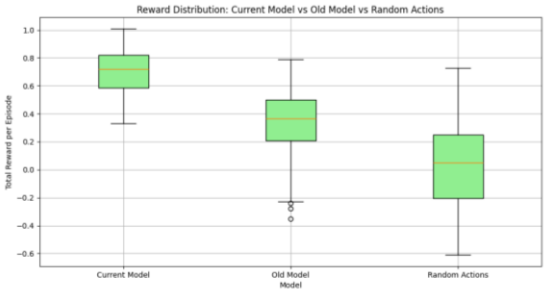


Figure 48. rewards for retrained PPO vs. Old PPO vs. Random as Player 1

The Win loss plot for retrained PPO vs. old PPO vs. Random actions:

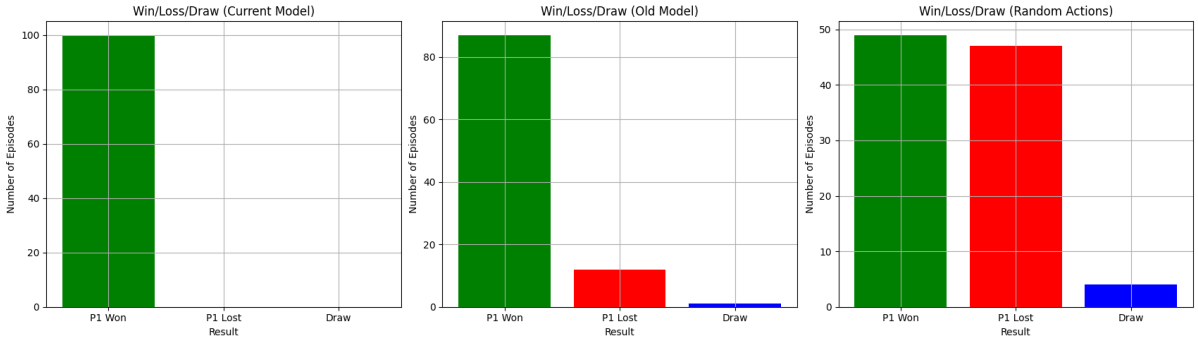


Figure 50. Win/Draw/Loss test for retrained PPO vs. Old PPO vs. Random as Player 1

It is noteworthy that earlier random vs. built-in random (in section 4.3.2.2, figure 33.) wasn't a 50-50 chance and Player 1 could be seen to have an upper hand there. However, it is not the case now.

4.4.4. Stage 4: Beating Retrained PPO

In this section, attempts to beat the trained Player 1 in previous stage, would be made. The aim is to beat PPO with 20% randomness with newly trained Player 2. A few new techniques were implemented.

4.4.4.1. Curriculum Learning Mechanism

This step is intended to dynamically increase the difficulty of the environment based on agent's performance to ensure steady learning progression.

- **Key Parameters**:

- patience: Minimum episodes to evaluate performance before adjusting difficulty.
- threshold: The minimum improvement in average reward required to trigger a phase change.

- **Logic**:
  - Tracks rewards for each episode in self.episode_rewards.
  - Evaluates performance after patience episodes using the moving average of rewards.
  - If the mean reward improves beyond best_mean_reward, the current curriculum phase is incremented (self.env.current_phase += 1), and the randomness in opponent's behaviour is reduced using set_current_mixture_ratio() method in the environment class.
  - Logs episodes where phase changes occur (phase_change_episodes) for visualization.
  - In total, there are 3 phases, 1st with 80% randomness, 2nd with 50% and 3rd with 20% randomness.
  - It is implemented in the callback function.

```
class CurriculumLearningCallback(BaseCallback):
    def __init__(self, env: StreetFighterEnv, patience: int = 500, threshold: float = 10.0, verbose=1):
        super(CurriculumLearningCallback, self).__init__(verbose)
        self.env = env
        self.patience = patience
        self.threshold = threshold
        self.best_mean_reward = -np.inf
        self.no_improvement_count = 0
        self.episode_rewards = []
        self.phase_change_episodes = [] r

    def on_step(self):
        if "episode" in self.locals["infos"][0]:
            reward = self.locals["infos"][0]["episode"]["r"]
            self.episode_rewards.append(reward)
            if len(self.episode_rewards) >= self.patience:
                mean_reward = np.mean(self.episode_rewards[-self.patience:])
                if mean_reward > self.best_mean_reward:
                    self.best_mean_reward = mean_reward
                    self.no_improvement_count = 0
                    if self.env.current_phase < len(self.env.curriculum) - 1:
                        self.env.current_phase += 1
                        self.env.set_current_mixture_ratio(self.env.current_phase)
                        current_episode = len(self.episode_rewards)
                        self.phase_change_episodes.append(current_episode)
                        print(f"Curriculum Phase Increased to {self.env.current_phase + 1} "
                              f"at Episode {current_episode}, ratio: {self.env.get_current_mixture_ratio()}")
                        self.no_improvement_count = 0
                else:
                    self.no_improvement_count += 1
                    print(f"No improvement for {self.no_improvement_count} episodes.")

        return True  # Continue training
```

Figure 51. Code for curriculum Learning implementation

4.4.4.2. Reward Function Enhancements

1. **Health-Based Rewards:**
   - The function calculates rewards based on health changes for both players.
   - Damaging Player 1 is heavily rewarded (multiplied by 3).
   - Taking damage from Player 1 is penalized (divided by 3).
   - Winning the match (Player 1's health reaches 0) provides a large bonus reward of 10.0.
2. **Distance Management:**
   - Penalizes being too far from the opponent.
   - Applies a small negative reward that increases as distance grows beyond 500 units.
   - Provides a small positive reward for closing the distance between players.
3. **Action Quality:**
   - Penalizes inactivity (not attacking).
   - Applies a penalty for attack actions that don't deal damage.
   - Gives a small bonus for executing attack actions.
4. **Low Health Dynamics:**
   - Introduces special reward mechanics when Player 2's health is below 50%.
   - Adds a larger penalty for taking hits when low on health.
   - Provides a more significant reward for dealing damage when low on health.
   - This encourages more strategic behaviour during critical health moments.

This should:

- Encourage aggressive but smart play.
- Discourage passive behaviour which was the problem faced in previous stages.
- Reward strategic positioning and attacking.

The reward function is designed to guide the AI agent towards:.

- Dealing damage to the opponent.
- Avoiding taking damage.
- Maintaining close proximity which was an issue in the earlier stages.
- Being actively engaged in the fight.

- Performing well even when at low health.

4.4.4.3. Memory Integration (PPO+LSTM)

To give the model memory to remember past observations, LSTM is integrated with PPO. A custom LSTM feature extractor function is created to inculcate a memory mechanism to process sequential data from the environment.

```
class CustomLSTMFeaturesExtractor(BaseFeaturesExtractor):
    def __init__(self, observation_space: gym.Space, features_dim: int = 256, buffer_size: int = 10):
        input_dim = observation_space.shape[0] // buffer_size
        super(CustomLSTMFeaturesExtractor, self).__init__(observation_space, features_dim)
        self.buffer_size = buffer_size

        # Define the LSTM feature extractor
        self.lstm = nn.LSTM(input_dim, 256, batch_first=True)
        self.fc = nn.Linear(256, features_dim)

    def forward(self, observations: th.Tensor) -> th.Tensor:
        batch_size = observations.size(0)
        sequence_length = self.buffer_size
        input_dim = observations.size(1) // sequence_length
        observations = observations.view(batch_size, sequence_length, input_dim)
        lstm_out, _ = self.lstm(observations)
        lstm_out = lstm_out[:, -1, :]
        features = self.fc(lstm_out)
        return features
```

Figure 52. Memory layers

The complete end to end flow for integrating PPO and LSTM:

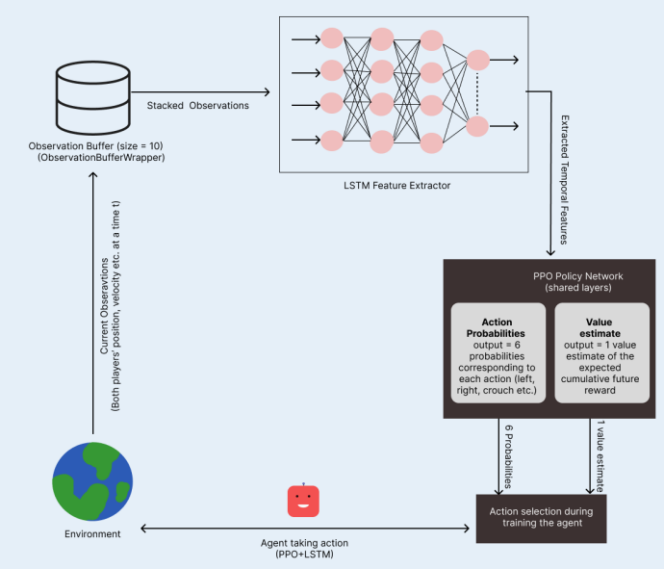


Figure 53. PPO+LSTM architecture

1. Agent would make an action in the environment.
2. This action will be treated as an input to the environment (StreetFighterEnv).
3. The Environment would in return provide with:
    a) **Observation:** Current game state including:
    - **Continuous Features:**
        - Positions (p1_pos, p2_pos)
        - Velocities (p1_vel, p2_vel)
        - Health statuses (p1_health, p2_health)
        - Distance between players (distance)
    - **Discrete Features:**
        - Player states (e.g., idle, attacking)
        - Attack states (e.g., light, medium, heavy)

    This observation is in normalized form and is same as Table 6. So total of 11 observations.
    b) **Reward:** Numerical value indicating the desirability of the action taken.
    c) **Done:** Boolean flag indicating if the episode has ended.
    d) **Info:** Additional information (e.g., game result).
4. The observations from the environment are provided to the observation buffer, which stores a buffer of these (10 in this case), and gives last 10 stacked observations in the form of a concatenated vector as output. This accumulates a sequence of past observations.
5. These past observations are given as an input to the custom LSTM feature extractor which transforms these observations into a feature vector that captures temporal dependencies. There would be 256 features in the output after the extraction.
6. These extracted temporal features are then given as input to the PPO's policy network via policy_kwargs. After this, two primary outputs are generated:
    a) **Action Probabilities:** 6 no. of outputs (one for each action). Each output corresponds to the probability of selecting a specific action, e.g.: [0.1, 0.2, 0.3, 0.1, 0.2, 0.1] for actions 0 through 5. This is done because PPO samples actions based on these probabilities during training process to explore different strategies.
    b) **Value Estimate:**1 output (a single scalar value) representing an estimate of the expected cumulative future reward from the current state. This is used to calculate advantages, which help in deciding whether actions are better or worse than expected.
7. These outputs are used by PPO during the training process for action selection. The sampled actions, along with the rewards received, are used to update the network.

8. The model was trained for 300,000 steps.

4.4.4.4. DQN+LSTM

Another model was trained to beat the same player (PPO with 20% randomness). This one was trained without curriculum learning though. However, the LSTM integration part was the same but without the buffer.

4.4.4.5 Results and observations

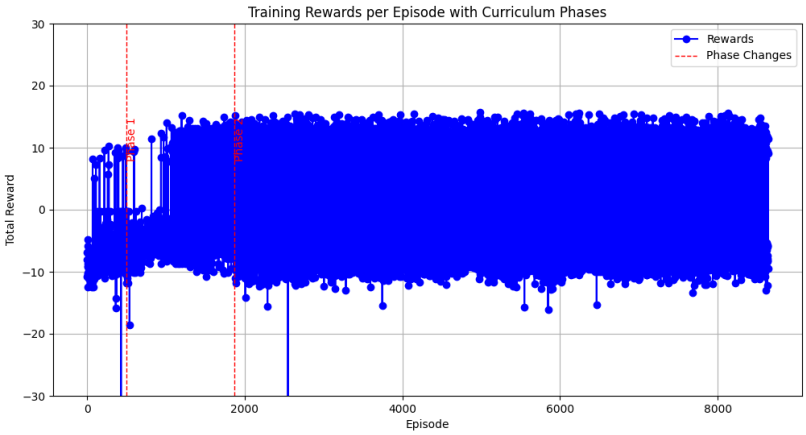


Figure 54. Training plot for P2 with curriculum learning

The training plot for PPO+LSTM model shows the progression of training rewards across episodes with curriculum learning phases. Rewards increase gradually during earlier episodes, reflecting the agent's learning progress. Red dashed lines mark phase transitions, indicating when the environment's difficulty was adjusted based on performance. 1st red line indicates switch from 80% randomness to 50% randomness and the 2nd one represents switch from 50% randomness to 20% randomness. Over time, the rewards don't converge very much towards the end though as the rewards keep fluctuating between -10 and +10. The best model was received and saved after 280,000 steps with best mean reward of 11.6.

During the testing phase, a total of 100 matches were played against PPO with 20% randomness (Player 1) by the newly trained PPO, DQN and a random player.

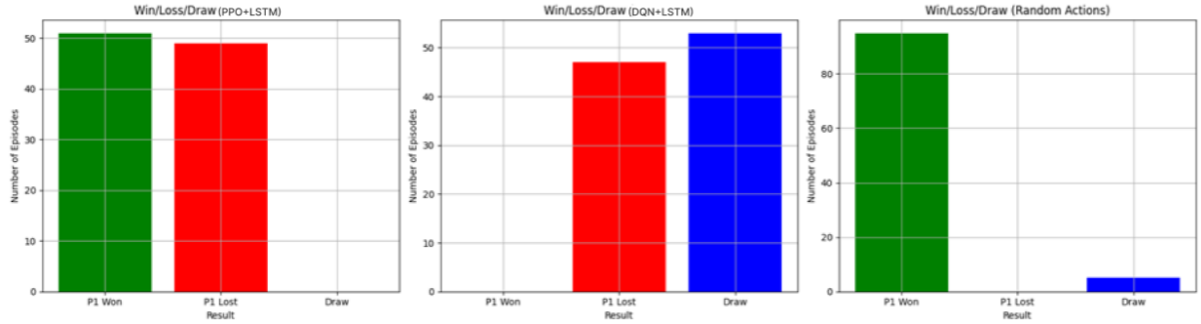


Figure 55. Win/Loss/Draw test for PPO+LSTM vs. DQN+LSTM vs. Random actions as Player 2

Win/Loss/Draw Comparison Plot above shows that:

1. **Current Model (PPO with Curriculum Learning):**
   - Shows a balanced distribution of wins and losses for Player 1 and Player 2, with no draws. It is able to beat almost 50% of the times.
   - Indicates competitive gameplay with frequent adjustments due to curriculum learning.
2. **Old Model (DQN):**
   - Displays a higher proportion of draws, suggesting that the DQN model is less decisive compared to PPO.
   - Indicates moderate performance by never losing but lacks the ability to consistently win.
3. **Random Actions:**
   - Player 1 dominates the games entirely, with 0 wins for Player 2.
   - Reflects the randomness and inefficiency of untrained actions.

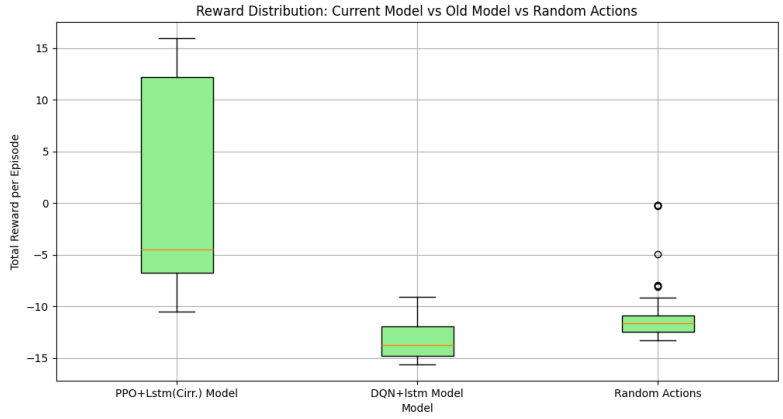


Figure 56. Rewards for PPO+LSTM vs. DQN+LSTM vs. Random actions as P2

Reward Distribution Boxplot above shows:

1. **Current Model (PPO with Curriculum Learning):**
   - Exhibits a higher reward range, with rewards extending above 10.
   - Indicates effective learning and better gameplay strategies with a wide range of successful actions.
2. **Old Model (DQN):**
   - Shows a narrower reward range, with rewards concentrated around the lower end.
   - Highlights suboptimal learning and less adaptability compared to PPO.
3. **Random Actions:**
   - Displays the lowest and most negatively skewed reward distribution.
   - Reflects the ineffectiveness of random actions, with poor outcomes in most episodes.

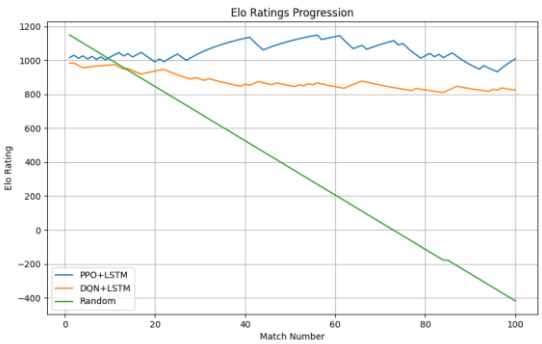
Result: PPO with curriculum learning was able to win against Player 1 almost 50% of the time as shown in fig. 55, indicating a competitive performance but not a definitive ability to consistently beat Player 1. The DQN model seems difficult for Player 1 to beat, as Player 1 never achieves any wins against it. However, this does not imply that DQN consistently outperforms Player 1 but rather, it often results in draws, suggesting a defensive but not dominant strategy as it just stands in a corner and hits the opponent when it approaches. On the other hand, PPO engages in the fight which was the main intention and curriculum learning made it possible. Also, random actions fail completely. Overall, Player 1 seems to be the stronger player.

5. Further Evaluation and Testing

5.1. Player 2 vs Player 2

In this section, comparisons among different models are made, which played as Player 2. The readings were calculated over 100 matches.

5.1.1. Elo Ratings Progression



- **PPO+LSTM**: Elo ratings improve over matches, consistently as it increased from 1000 to near about 1200 in the middle. However, it did drop towards the end which states that it lost some matches against P1. However, still it is seen to be

outperforming DQN+LSTM and Random.

- **DQN+LSTM**: Elo ratings drop slightly, remain mostly steady but below PPO, reflecting stable yet less adaptive performance.
- **Random**: Elo ratings drastically drop, reflecting poor performance.

PPO+LSTM is seen to have much superior performance, while DQN+LSTM is consistent but less effective. The DQN model had lower ratings because of having more draw matches which reduce the rating. Random actions perform poorly.

Figure 57. Elo Ratings for PPO+LSTM vs. DQN+LSTM vs. Random actions as P2

5.1.2. Episode Length Distribution (Violin Plot)

- **PPO+LSTM**: Shows a compact distribution around 30-40 steps per episode, indicating efficient and consistent gameplay as it was able to finish the game sooner.
- **DQN+LSTM**: Episode lengths are fixed and not distributed, reflecting deterministic decision-making.
- **Random**: The distribution is more spread, with episodes lasting both very short and very long, reflecting randomness.

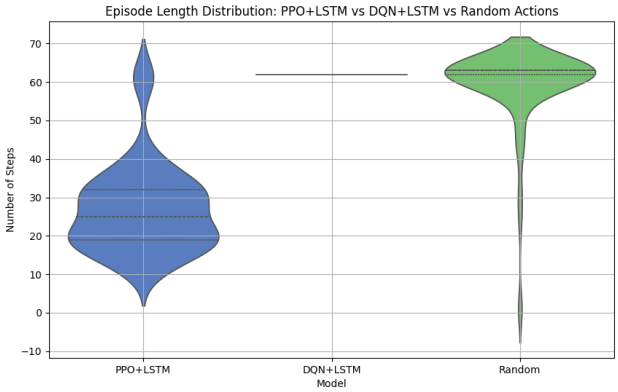


Figure 58. Episode length comparison among P2 models

PPO+LSTM adapts well to the game dynamics, while Random actions are inefficient. DQN is more consistent but lacks variability.

5.1.3. Action Efficiency (Box Plot)

PPO+LSTM: Median efficiency around 0.25, with variability, indicating a balance of attack and other actions.

DQN+LSTM: Near-perfect efficiency, as it almost always performs attack actions.

Random: Low efficiency and highly variable, as many actions are unrelated to attacking.

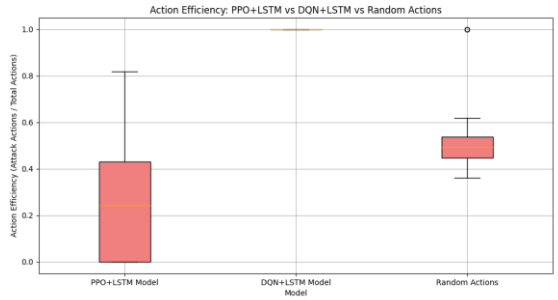


Figure 59. Action efficiency comparison among P2 models

DQN+LSTM focuses only on attacks which shows it never moves from its position at all, while PPO+LSTM explores more diverse strategies. Despite having a lower attack action frequency, it is still able to beat the player half of the time which shows he is more of a strategic player than an aggressive one.

5.1.4. Action Frequency vs Reward (Heatmap)

PPO+LSTM: Shows high rewards with moderate action frequencies, indicating efficient use of actions for optimal outcomes.

DQN+LSTM: Fewer data points, showing consistent but lower rewards than PPO.

Random: Almost all frequencies correlate with low or negative rewards.

PPO+LSTM balances action frequency with rewards, outperforming DQN+LSTM, which is efficient but less optimal. Random actions yield poor results. This also shows that PPO

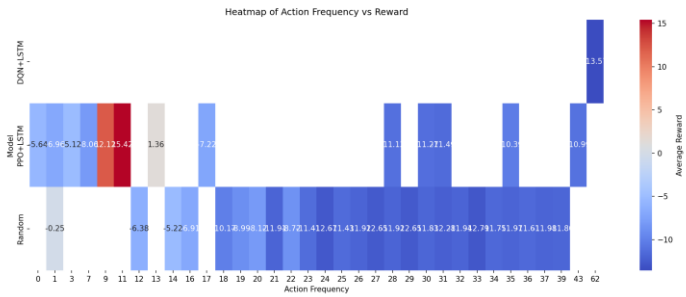


Figure 60. Action frequency vs. reward comparison among P2 models

takes just 8-10 actions to beat the other player. If he skips the window, then he is not able to turn the tables on the opponent.

5.1.5. Observations

1. **PPO+LSTM**:
   - Most adaptive and effective strategy making him strategic player.
   - Outperforms others in action efficiency, reward maximization, and Elo ratings.
2. **DQN+LSTM**:
   - Reliable but rigid strategy.
   - Focuses too heavily on attacks without exploring other actions making him an aggressive player.
3. **Random Actions**:
   - Inefficient and performs poorly overall.

5.2. Player 1 vs Player 2

Elo ratings were used to compare Player 1 with different models. The readings were calculated over 100 matches. Since Player 1 is a stronger player, it was given a higher initial rating of 1200 and rest all models were initialized to a value of 1000.

5.2.1. PPO (P1) vs. PPO (P2)

The Elo rating distributions of Player 1 (P1) and Player 2 (PPO+LSTM) overlap significantly. The peak density of Player 2 is slightly shifted to the left of Player 1, indicating that while Player 2 performs well, it is slightly weaker than Player 1 on average.

Conclusion: The PPO+LSTM model is competitive but struggles to outperform Player 1 consistently, showcasing comparable skill but lacking dominance.

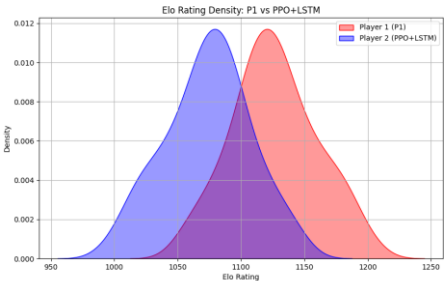


Figure 61. Elo ratings for PPO (P1) vs. PPO (P2)

5.2.2. PPO (P1) vs. DQN (P2)

The Elo distribution of Player 2 (DQN+LSTM) is higher than Player 1's and shows minimal overlap. Player 2's peak density is shifted right, indicating consistent superior performance over Player 1.

Conclusion: The DQN+LSTM model surpasses Player 1 in skill, showing dominance and effectiveness in the task.

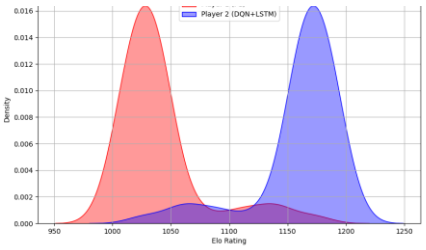


Figure 62 . Elo ratings for PPO (P1) vs. DQN (P2)

5.2.3. PPO (P1) vs. Random (P2)

There is a large separation between the Elo distributions. Player 1 dominates, with its peak density far higher than Player 2's random actions. The random model's Elo is significantly lower, with little overlap.

Conclusion: Random actions are vastly inferior to Player 1's performance, demonstrating no ability to challenge or compete effectively.

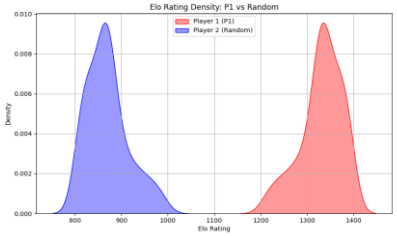


Figure 63. Elo ratings for PPO (P1) vs. Random actions (P2)

5.2.4. Analysis

Why does PPO lose to P1 but win against DQN in Elo ratings?

It has something to do with how Elo ratings are calculated, where a draw match is given a score of .5 and a win a score of 1.So, because of more draw matches in P1 vs DQN and lesser total wins as compared total wins by PPO against P1, PPO+LSTM outperforms DQN+LSTM overall, but DQN+LSTM still has an edge against the higher-rated player P1 as DQN is either

draws the match or wins it but never loses, gaining a higher score when Elo-ratings are compared for P1 vs P2.
The key takeaway is that:

1. The Elo rating dynamics are influenced by both the initial conditions and the relative learning/adaptation capabilities of the different agents.
2. The PPO+LSTM agent appears to be learning more effective strategies that allow it to outperform the DQN+LSTM agent, even though DQN+LSTM has an advantage against the higher-rated P1.

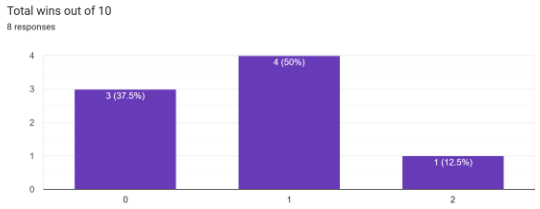
5.3. Player 1 vs Humans

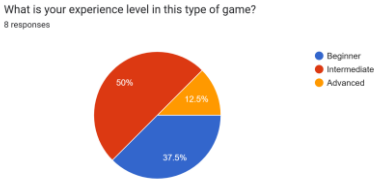
5.3.1. Survey

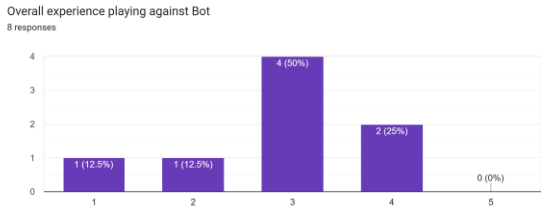
In a test, a group of 8 people was made to play against the Player 1 PPO model with 20% randomness. A total of 10 matches were played by each person, and they were made to answer a few questions in a survey using Google Forms. The results of which are as follows:

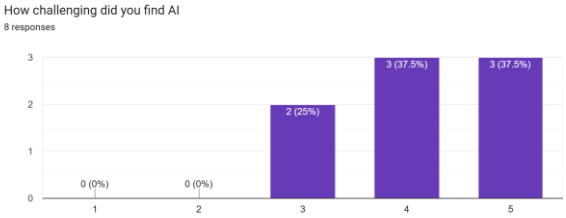
Timestamp	Total wins out of 10	What is your experience level in this type of	Overall experience playing against	How challenging did you find AI	Were AI's moves predictable?	Did bot act in an intelligent manner or logically?	Was the bot good or bad opponent as compared to human?	Do you feel it was a fair play?	If no, why?	Did you have fun playing against bot?	Any comments?	Your name
2024/11/28 9:20:52 PM GMT	0	Intermediate	2	5	Yes	No	Good	No	bot's hitboxes were too much, stunlock was too long with blocking	Yes	add ranged moves e.g. hadouken; less stunlock for block, less hitbox timing so	miguel
2024/11/28 9:24:38 PM GMT	1	Intermediate	4	5	Yes	No	Bad	No	It doesn't give the other player the space to stand in front of it thus the human player fails all the time as if that's a	Yes		Amulya
2024/11/28 9:28:19 PM GMT	1	Beginner	3	4	No	No	Good	Yes		No	Fixing bug that able to keep bot moving would be much	Phuwit Wanitkamonnun
2024/11/28 9:35:29 PM GMT	1	Intermediate	1	3	Yes	No	Bad	No	I figured out it would just stay in one place and spam the same move so I punished it with a faster punch after	No	Make it move left and right?	Azmat
2024/11/28 9:44:18 PM GMT	0	Beginner	3	5	Yes	No	Good	No	spamming punches and	Yes		Ryan
2024/11/29 12:03:46 AM GMT	2	Beginner	3	3	No	Yes	Good	Yes		Yes		
2024/11/29 11:50:40 AM GMT	0	Intermediate	4	4	Yes	Yes	Good	Yes		Yes		Mohan Selvan
2024/12/04 3:32:17 AM GMT	1	Advanced	3	4	Yes	Yes	Good	No	too fast	No		

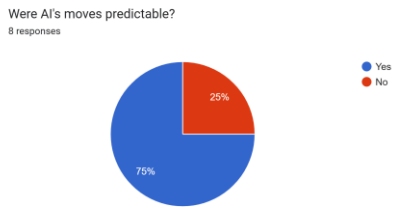
Figure 64. Survey responses

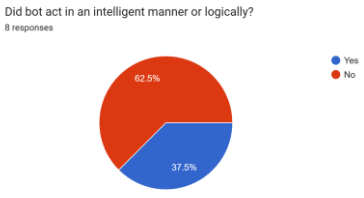


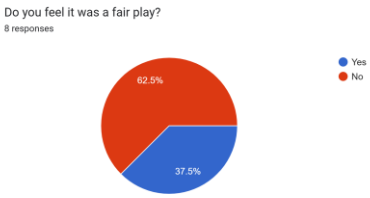


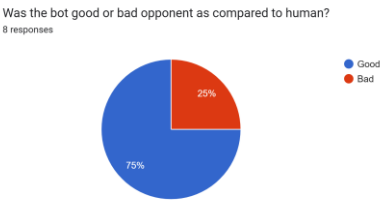












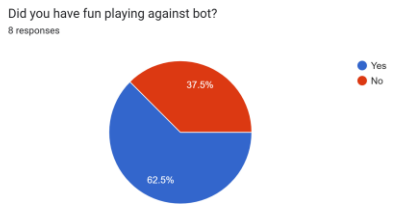


Figure 65. Survey analysis

5.3.2. Observations

From the survey responses, the following insights can be gathered:

Human Feedback About the Bot:

1. **Fair Play Concerns:**
    - Several users felt the bot's behaviour was unfair, citing reasons such as excessive hitboxes, stun lock duration, and spamming repetitive moves. These issues were highlighted by a few, especially when the bot didn't allow room for strategic play.
2. **Challenge Level:**
    - The bot was perceived as moderately to highly challenging (average challenge rating of 4.25 out of 5), indicating its difficulty level was appropriate for most players, though it could feel overwhelming for some.
3. **Intelligence and Predictability:**
    - 75% of the people found it predictable and a few also found it exploitable once patterns were identified.
4. **Enjoyment:**
    - Despite the issues, most players found the experience enjoyable (majority answered "Yes" to having fun).
5. **Skill Level Feedback:**
    - Beginners and intermediate players had the most to say about improvements, indicating the bot may cater more effectively to advanced players due to its difficulty level and repetitive behaviour.

5.3.3. Conclusion

Conclusion:

- **Strengths:** The bot is challenging and engages players, with most finding the experience fun. It has a good balance between being a tough opponent and providing opportunities for strategic play for humans.
- **Weaknesses:** Its predictability, repetitive spamming, and excessive stun lock mechanics lead to frustration, making it less appealing for users.
- **Improvements Needed:** To enhance fairness, the bot should:
    - Avoid spamming repetitive moves.
    - Reduce attacking frequency and stun lock issues to ensure fair play.
    - Introduce varied movement and strategies to prevent exploitability and make it more fun.

Overall, while the bot provides a solid challenge, addressing these weaknesses could significantly improve user satisfaction.

6. Conclusion and Future Work

The project successfully developed reinforcement learning agents to play a Street Fighter game using PPO and DQN algorithms. All agents were better than a random player. DQN excelled in exploiting predictable and repetitive behaviours in deterministic reward settings.

For example, DQN's strategy of standing in one corner and striking when the opponent approached was effective in certain scenarios and was able to avoid getting attacked by overfitting on opponent's actions. However, it rarely was seen to take any other action other than attacks which wasn't a strategic play. PPO's ability to adapt to dynamic gameplay made it more effective in stochastic reward scenarios and against varied strategies. Also, PPO explored a broader set of actions, such as engaging in aggressive or defensive strategy depending on the situation. This adaptability was facilitated by its stochastic policy, which balances exploration and exploitation. This was shown by the benchmarking results too. Overall, PPO seemed a better suited algorithm in areas where a more generalized approach is needed like in the Street Fighter game and DQN where a specific thing needs to be solved, for example clearing a level in Mario. This aligns with existing literature [17], highlighting the respective strengths of value-based and policy-based approaches in different contexts.

The bots demonstrated the ability to outperform random players consistently and exhibited strategic gameplay through iterative training and curriculum learning techniques. Notable achievements include the retrained PPO agent, which demonstrated stability and adaptability across varying levels of difficulty, and the reward-based strategies that guided the agents toward aggressive yet calculated gameplay. However, challenges persisted in training Player 2 to beat a trained Player 1, even after incorporating advanced mechanisms like reward reshaping and curriculum learning, which states that Player 1(PPO) has an optimal strategy. However, these strategies did help in getting out of Nash equilibrium. Also, the integration of memory buffer with policy networks in PPO helped in achieving a higher reward and taking more strategic actions. Human feedback highlighted that while the bots provided an engaging experience but improvements in the diversity of moves and frequency are needed to be adjusted to enhance realism and competitiveness.

This project confirmed many findings from existing reinforcement learning literature [8], particularly regarding the strengths of PPO and curriculum learning and the challenges in multi-agent competitive settings. However, it also provided fresh insights into the nuances of applying these techniques in a highly interactive, human-centred gaming environment. Future work can build upon these findings to address the identified gaps and enhance both AI performance and user experience. Additionally, the environment wrapper built in this project could be used by others to train their own models and they could train a better opponent for PPO-trained Player 1. Also, some issues were found in the Java game engine and were addressed through a pull request on GitHub which got merged.

Learning Outcomes:

1. **Effectiveness of Reinforcement Learning Algorithms**:
    - PPO demonstrated robust performance in dynamic and complex environments, balancing exploration and exploitation effectively, especially when combined with curriculum learning. It was found out to be the most stable.
    - DQN, while effective in simpler scenarios, struggled with adaptability and exhibited suboptimal strategies when faced with diverse or evolving challenges.
2. **Curriculum Learning**:
    - The incremental difficulty adjustments in curriculum learning helped stabilize training and improve agent performance. Gradual exposure to harder scenarios enhanced the agent's ability to gain higher rewards and engage in a competitive play.
3. **Reward Engineering**:

- Proper reward normalization and structure played a vital role in guiding the agents' behaviour, highlighting the importance of well-designed reward systems in reinforcement learning environments. This step seems to be the most critical when wanting an agent to do a particular thing.

4. **Human-AI Interaction Insights**:
   - Feedback from human testers revealed that while the AI was engaging and competitive. However, it occasionally exhibited repetitive behaviour and lacked human-like unpredictability, emphasizing the need for improvements in move diversity and decision-making realism as addressed in [4].
5. **Challenges with Multi-Agent Systems**:
   - The difficulty in training Player 2 to consistently outperform Player 1 highlighted the complexity of competitive multi-agent reinforcement learning and the potential need for advanced techniques like multi-agent exploration or game-theoretic approaches.

Future Work:

1. **Improving Agent Intelligence:**
   - Implement multi-agent reinforcement learning to allow Player 2 to adapt more dynamically to Player 1's strategies.
   - Experimenting with different techniques to allow the agent to adapt dynamically to varying player styles, including defensive, aggressive, and unpredictable strategies.
2. **Using screen pixels in observation space:** The observation space currently used included the meta-data such as players' position, velocity etc. Another way to do this could be by using images to represent a state in the game. Some refinements to the game engine were made to do the same by capturing the screens in every step and making the size of the screen captures very small and in grayscale with intention of feeding it to the CCN models later. Due to time constraints of this dissertation, this approach could not be completed but is worth to be tried out.
3. **Reward Function Refinement:**
   - Introduce more granular rewards for advanced strategies like counterattacks, blocking, and combos to explore on defensive strategies as well.
   - Penalize repetitive or unrealistic behaviours to improve diversity in gameplay.
4. **Environment Enhancements:**
   - Expand the action space to include special moves and defensive strategies.
5. **Human-AI Interaction:**
   - Conduct further user testing with a larger, more diverse audience to refine the bot's human-like behaviours.
   - Develop adaptive difficulty levels to tailor the experience to individual player skill levels.
6. **Exploration of Other RL Techniques:**
   Test state-of-the-art RL algorithms like Soft Actor-Critic (SAC) or Hybrid RL models to benchmark performance against PPO and DQN in similar environments.
7. **Addressing issues accentuated by humans:**
   - Fix the stun lock in game by reducing the time for being in stunned state.
   - The model's performance was decreased a lot when the speed of the game was reduced. More analysis could be done to fix it.
   - Make the hitboxes smaller and giving more window for humans to hit.
8. **Improving agent performance in real-time:**

The game was tweaked a little during the training time to make the training faster. Once the agent was trained and the game changes were rolled back, the agent was made to play against humans. During that time, it was found out that the model's performance went down to some extent. It was still able to beat humans, but its performance wasn't up to par like how it was during the testing time. For example, if it was supposed to go to the middle of the screen, it would only move half of that distance. A workaround of this must be found so that even if the game is made faster to reduce the training time, the agent should perform the same in real-time as it did during the testing phase. One of the solutions could be to introduce a time scaling factor within the agent's observations or actions to normalise movements irrespective of the speed of the game.

References

[1] Google DeepMind, "Technologies," [Online]. Available: https://deepmind.google/technologies/alphago/.

[2] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang and W. Zaremba, "OpenAI Gym," 2016.

[3] D. Berry, "street-fighter," 06 2024. [Online]. Available: https://github.com/drewberry612/street-fighter.

[4] C. A. Cruz and . J. R. Uresti, "HRLB^ 2: A reinforcement learning based framework for believable bots," *Applied Sciences,* vol. 8, no. 12, pp. 1-24, 2018.

[5] B. Xia, X. Ye and A. O. Abuassba, "Recent Research on AI in Games," *2020 International Wireless Communications and Mobile Computing (IWCMC),* p. 506, 2020.

[6] S.-X. Go, Y. Jiang and D. K. Loke, "A Phase-Change Memristive Reinforcement Learning for Rapidly Outperforming Champion Street-Fighter Players," *Advanced Intelligent Systems,* vol. 5, no. 11, pp. 1-15, 2023.

[7] W. Li, Z. Ding, S. Karten and C. Jin, "FightLadder: A Benchmark for Competitive Multi-Agent Reinforcement Learning," 2024.

[8] A. Lohokare, A. Shah and M. Zyda, "Deep Learning Bot for League of Legends," in *Proceedings of the Sixteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-20),* Los Angeles, California, 2020.

[9] R. S. Sutton and A. G. Barto, Reinforcement Learning An Introduction, Cambridge: The MIT Press, 2018.

[10] S. Eldridge, "Science & Tech, Nash Equilibrium," 22 12 2022. [Online]. Available: https://www.britannica.com/science/Nash-equilibrium.

[11] Stable Baselines3, "Policy Networks," [Online]. Available: https://stable-baselines3.readthedocs.io/en/master/guide/custom_policy.html.

[12] S. Chandan, "Elo Rating System – Everything You Need to Know," [Online]. Available: https://chessklub.com/elo-rating-system/.

[13] B. Dagenais, "Py4J - A Bridge between Python and Java," 23 12 2009. [Online]. Available: https://www.py4j.org/.

[14] A. Brandolini, "j2py 0.3.3.6," 4 10 2022. [Online]. Available: https://pypi.org/project/j2py/.

[15] PyLessons, "Introduction to Reinforcement Learning," 22 09 2019. [Online]. Available: https://pylessons.com/CartPole-reinforcement-learning#google_vignette.

[16] T. Kapoor, "Street Fighter Game with Reinforcement Learning Proposal," 2024.

[17] Y. -t. Liu, J. -m. Yang, L. Chen, T. Guo and Y. Jiang, "Overview of Reinforcement Learning Based on Value and Policy," *2020 Chinese Control And Decision Conference (CCDC),* Hefei, China, 2020, pp. 598-603, doi: 10.1109/CCDC49329.2020.9164615.

Appendices

Complete resources and source code has been uploaded on GitLab: https://cseegit.essex.ac.uk/23-24-ce901-sl-ce902-su/23-24_CE901-SL_CE902-SU_kapoor_tania

Main Code:

Environment code

```
import time
import numpy as np
from gymnasium import spaces
import gymnasium as gym
from gymnasium.envs.registration import register
import warnings
from py4j.java_gateway import JavaGateway
from stable_baselines3 import PPO
from stable_baselines3.common.env_checker import check_env
from stable_baselines3.common.vec_env import DummyVecEnv
from stable_baselines3.common.monitor import Monitor

import torch as th
from torch import nn

warnings.filterwarnings("ignore")

# Register the environment
register(
    id='StreetFighter_tk1998-v0',
    entry_point='StreetFighter_tk:StreetFighterEnv',
)

class KeyEvent:
    VK_ENTER = 10
    VK_UP = 38
    VK_DOWN = 40
    VK_ESCAPE = 27
    VK_1 = 49
    VK_2 = 50

class StreetFighterEnv(gym.Env):
    def __init__(self, game=None):
        super(StreetFighterEnv, self).__init__()

        # Define action and observation spaces
        self.action_space = spaces.Discrete(
            6)  # Actions: 0=Left, 1=Right, 2=Light Attack, 3=Medium Attack, 4=Heavy Attack, 5=Crouch

        # Initialize PPO model for opponent (Player 1)
        self.ppo_old = PPO.load("ppo_street_fighter_revised")
        self.p_random = 0.8
        # Initialize game connection via Py4j
        if game is None:
            self.gateway = JavaGateway()
            self.game = self.gateway.entry_point.getGame()
        else:
            self.game = game

        if self.game is None:
            raise Exception("Failed to connect to the game.")

        # Initialize reward tracking variables
        self.cumulative_reward = 0
        self.hits_taken = 0  # Counter for the number of hits Player 2 has taken

        # Define observation space with normalization and relative positions
        self.observation_space = spaces.Box(
            low=np.array([
                0,  # Relative Player 1 position x (min)
                -1,  # Relative Player 1 velocity x (normalized)
                0,  # Player 1 health (normalized)
                0,  # Relative Player 2 position x (min)
                -1,  # Relative Player 2 velocity x (normalized)
```

```
            0,  # Player 2 health (normalized)
            0,  # Normalized Distance between players
            0, 0,  # Player 1 and Player 2 states (e.g., idle, attacking)
            0,  # Player 1 attack states: light, medium, heavy
            0  # Player 2 attack states: light, medium, heavy
        ]),
        high=np.array([
            1.0,  # Relative Player 1 position x (max normalized to 1)
            1.0,  # Relative Player 1 velocity x (max normalized to 1)
            1.0,  # Player 1 health (max normalized to 1)
            1.0,  # Relative Player 2 position x (max normalized to 1)
            1.0,  # Relative Player 2 velocity x (max normalized to 1)
            1.0,  # Player 2 health (max normalized to 1)
            1.0,  # Normalized Distance between players (max normalized to 1)
            1, 1,  # State encoding (0 = idle, 1 = attacking)
            4,  # Player 1 attack states: 0, 1, 2, 3
            4  # Player 2 attack states: 0, 1, 2, 3
        ]),
        dtype=np.float32
    )

    # Initialize previous state variables for reward calculation
    self.prev_p1_health = 1.0  # Normalized
    self.prev_p2_health = 1.0  # Normalized
    self.prev_distance = 1.0  # Normalized

    # Curriculum Learning Parameters
    self.total_steps = 0  # Initialize step counter

    # Define curriculum phases as a list of tuples (step_threshold, p_random)
    self.curriculum = [
        (50000, 0.8),
        (100000, 0.5),
        (float('inf'), 0.2)
    ]

    self.current_phase = 0
    self.hits_taken_low_health = 0
    self.hits_given_low_health = 0

def get_current_mixture_ratio(self):
    """
    Determine the current mixture ratio
    Returns the probability of taking a random action.
    """

    return self.p_random  # Default to the last phase's p_random
def set_current_mixture_ratio(self,phase):
    """
    sets the current mixture ratio
    Sets the probability of taking a random action.
    """
    if phase == 0:
        self.p_random = 0.8
    elif phase == 1:
        self.p_random = 0.5
    else:
        self.p_random = 0.2



def simulate_key_press(self, key_code):
    # Call the wrapper function in Java to handle the key press
    self.game.simulateKeyPress(key_code)

def start_game_auto(self):
    # Start the game by simulating key presses
    self.simulate_key_press(KeyEvent.VK_ENTER)
    time.sleep(1)  # Wait for the game to initialize
    self.simulate_key_press(KeyEvent.VK_ENTER)
    time.sleep(1)  # Wait for the game to start

    # Character selection
    self.simulate_key_press(KeyEvent.VK_2)
    time.sleep(0.5)
```

```
        self.simulate_key_press(KeyEvent.VK_2)
        time.sleep(0.5)

    def step(self, action_p2):
        """
        Execute one step in the environment.

        Parameters:
            action_p2 (int): Action taken by Player 2 (the agent).

        Returns:
            obs (np.array): Observation of the current state.
            reward (float): Reward obtained from taking the action.
            done (bool): Whether the episode has ended.
            truncated (bool): Whether the episode was truncated.
            info (dict): Additional information.
        """
        self.current_action = action_p2
        self._handle_action(action_p2, player=2)
        self.game.requestAdvance()
        obs_p1 = self._get_next_frame()
        # Determine Player 1's action based on the current mixture ratio
        p_random = self.get_current_mixture_ratio()
        if np.random.rand() < p_random:
            action_p1 = self.action_space.sample()  # Random action
        else:
            action_p1, _ = self.ppo_old.predict(obs_p1, deterministic=True)  # PPO action

        action_p1 = int(action_p1)
        self._handle_action(action_p1, player=1)
        self.game.requestAdvance()
        obs = self._get_next_frame()
        reward = self._compute_reward()
        done, result = self._check_done()
        truncated = False  #
        self.cumulative_reward += reward
        info = {"result": result} if result is not None else {}
        self.total_steps += 1

        return obs, reward, done, truncated, info

    def _handle_action(self, action, player=1):
        """
        Handle the action taken by a player.

        Parameters:
            action (int): The action to be taken.
            player (int): The player number (1 or 2).
        """
        action = int(action)
        controls = self.game.getPlayer1Controls() if player == 1 else self.game.getPlayer2Controls()
        actions = {
            0: controls[2],  # Left
            1: controls[3],  # Right
            2: controls[4],  # Light attack
            3: controls[5],  # Medium attack
            4: controls[6],  # Heavy attack
            5: controls[1]   # Crouch (down)
        }
        if action in actions and actions[action] is not None:
            if player == 1:
                self.game.player1Move(actions[action])
            else:
                self.game.player2Move(actions[action])

    def _get_next_frame(self):
        """
        Retrieve the current observation frame.

        Returns:
            np.array: The observation array.
        """
        # Get absolute positions, velocities, and health
        p1_pos = self.game.getPlayer1().getPositionX() / 1400  # Normalize to [0,1]
        p1_vel = self.game.getPlayer1().getVelocityX() / 600    # Normalize to [-1,1]
```

```
        p1_health = self.game.getPlayer1().getHealth() / 100    # Normalize to [0,1]
        p2_pos = self.game.getPlayer2().getPositionX() / 1400  # Normalize to [0,1]
        p2_vel = self.game.getPlayer2().getVelocityX() / 600    # Normalize to [-1,1]
        p2_health = self.game.getPlayer2().getHealth() / 100    # Normalize to [0,1]
        distance = abs(self.game.getPlayer1().getPositionX() - self.game.getPlayer2().getPositionX()) / 1800  #
Normalize to [0,1]


        combined_observation = self.game.getPlayer1().getObs()
        obs = np.array(combined_observation, dtype=np.float32)

        normalized_continuous_obs = np.array([
            p1_pos,
            p1_vel,
            p1_health,
            p2_pos,
            p2_vel,
            p2_health,
            distance
        ], dtype=np.float32)
        final_obs = np.concatenate([normalized_continuous_obs, obs[7:]])

        return final_obs

    def _compute_reward(self, max_health=100):
        """
        Compute the reward for the current step.

        Parameters:
            max_health (int): The maximum health a player can have.

        Returns:
            float: The computed reward.
        """
        # Get current health values (normalized)
        p1_health = self.game.getPlayer1().getHealth() / max_health
        p2_health = self.game.getPlayer2().getHealth() / max_health

        # Calculate health differences since the last step
        p2_health_diff = self.prev_p2_health - p2_health  # Positive if Player 2 lost health
        p1_health_diff = self.prev_p1_health - p1_health  # Positive if Player 1 lost health

        # Initialize reward
        reward = 0
        reward += (p1_health_diff) * 3  # Reward for damaging Player 1
        reward -= (p2_health_diff) / 3  # Penalty for taking damage

        # Reward for defeating Player 1
        if p1_health <= 0 and p2_health > 0:
            reward += 10.0  # Large reward for winning

        # Penalize for maintaining a large distance
        current_distance = abs(self.game.getPlayer1().getPositionX() - self.game.getPlayer2().getPositionX()) / 1800
        if current_distance > 0.277:  # Equivalent to 500 distance units
            reward -= 0.05 * (current_distance / 0.277)  # Penalty increases with distance

        # Reward for closing distance
        if current_distance < self.prev_distance:
            reward += 0.01 * ((self.prev_distance - current_distance) / 0.277)  # Small reward

        # Penalize for inactivity or not attacking
        if self.current_action not in [2, 3, 4]:  # Not an attack action
            reward -= 0.1  # Small penalty for inactivity
        elif p1_health_diff == 0 and self.current_action in [2, 3, 4]:  # Attack action taken but no damage
            reward -= 0.2

        if self.current_action in [2, 3, 4]:
            reward += 0.05

        if p2_health < 0.5:
            if p2_health_diff > 0:
                reward -= 0.3
            if p1_health_diff > 0:
                reward += 0.5
```

```
        self.prev_p1_health = p1_health
        self.prev_p2_health = p2_health
        self.prev_distance = current_distance

        return reward

    def _check_done(self):
        """
        Check if the episode is done.

        Returns:
            tuple: (done (bool), result (str or None))
        """
        p1_health = self.game.getPlayer1().getHealth() / 100
        p2_health = self.game.getPlayer2().getHealth() / 100
        p1_wins = self.game.getP1Wins()
        p2_wins = self.game.getP2Wins()

        result = None
        if p1_health <= 0 and p2_health > 0:
            result = "P1 Lost"  # Player 1 lost
        elif p2_health <= 0 and p1_health > 0:
            result = "P1 Won"  # Player 1 won
        elif p2_health == 0 and p1_health == 0:
            result = "Draw"  # Draw

        # Check if time runs out
        elif self.game.getTimeLeft() <= 2:
            if p1_health > p2_health:
                result = "P1 Won"
            elif p2_health > p1_health:
                result = "P1 Lost"
            else:
                result = "Draw"
            return True, result

        # Check win counters
        elif p1_wins > 0:
            result = "P1 Won"
        elif p2_wins > 0:
            result = "P1 Lost"
        if result is not None:
            # Reseting health and win counters for the next episode
            self.game.getPlayer1().setHealth(100)
            self.game.getPlayer2().setHealth(100)
            self.game.setP2Wins(0)
            self.game.setP1Wins(0)
            self.hits_taken = 0

            return True, result

        return False, None

    def reset(self, seed=None, options=None):
        """
        Reset the environment to its initial state.

        Returns:
            tuple: (observation (np.array), info (dict))
        """
        super().reset(seed=seed)
        self.game.resetGame()
        self.game.setP2Wins(0)
        self.game.setP1Wins(0)
        self.cumulative_reward = 0
        self.hits_taken = 0  # Reset hits_taken counter
        self.hits_taken_low_health = 0
        self.hits_given_low_health = 0
        self.prev_p1_health = self.game.getPlayer1().getHealth() / 100
        self.prev_p2_health = self.game.getPlayer2().getHealth() / 100
        self.prev_distance = abs(self.game.getPlayer1().getPositionX() - self.game.getPlayer2().getPositionX()) / 1800
        obs = self._get_next_frame()


        return obs, {}
```

```
    def render(self):
        """
        Render the environment. Not implemented.
        """
        pass
```

Training code

```
import gymnasium as gym
from stable_baselines3 import PPO
from stable_baselines3.common.env_checker import check_env
from stable_baselines3.common.callbacks import BaseCallback, EvalCallback, CallbackList
from stable_baselines3.common.evaluation import evaluate_policy
from stable_baselines3.common.torch_layers import BaseFeaturesExtractor
from stable_baselines3.common.vec_env import DummyVecEnv
from stable_baselines3.common.monitor import Monitor
from py4j.java_gateway import JavaGateway
import torch as th
from torch import nn
import numpy as np
import matplotlib.pyplot as plt
from StreetFighter_with_ppo import StreetFighterEnv


class ObservationBufferWrapper(gym.Wrapper):
    def __init__(self, env, buffer_size=10):
        super(ObservationBufferWrapper, self).__init__(env)
        self.buffer_size = buffer_size
        self.observation_buffer = []

        original_obs_shape = env.observation_space.shape
        new_low = np.repeat(env.observation_space.low, self.buffer_size, axis=0)
        new_high = np.repeat(env.observation_space.high, self.buffer_size, axis=0)
        self.observation_space = gym.spaces.Box(low=new_low, high=new_high, dtype=env.observation_space.dtype)

    def reset(self, **kwargs):
        observation, info = self.env.reset(**kwargs)
        self.observation_buffer = [observation] * self.buffer_size
        stacked_observation = self._get_stacked_observation()
        return stacked_observation, info

    def step(self, action):
        observation, reward, done, truncated, info = self.env.step(action)
        self.observation_buffer.pop(0)
        self.observation_buffer.append(observation)
        stacked_observation = self._get_stacked_observation()
        return stacked_observation, reward, done, truncated, info

    def _get_stacked_observation(self):
        return np.concatenate(self.observation_buffer, axis=0)


class CustomLSTMFeaturesExtractor(BaseFeaturesExtractor):
    def __init__(self, observation_space: gym.Space, features_dim: int = 256, buffer_size: int = 10):
        input_dim = observation_space.shape[0] // buffer_size
        super(CustomLSTMFeaturesExtractor, self).__init__(observation_space, features_dim)
        self.buffer_size = buffer_size
        self.lstm = nn.LSTM(input_dim, 256, batch_first=True)
        self.fc = nn.Linear(256, features_dim)

    def forward(self, observations: th.Tensor) -> th.Tensor:
        batch_size = observations.size(0)
        sequence_length = self.buffer_size
        input_dim = observations.size(1) // sequence_length
        observations = observations.view(batch_size, sequence_length, input_dim)
        lstm_out, _ = self.lstm(observations)
        lstm_out = lstm_out[:, -1, :]
        features = self.fc(lstm_out)

        return features


class CurriculumLearningCallback(BaseCallback):
    def __init__(self, env: StreetFighterEnv, patience: int = 500, threshold: float = 10.0, verbose=1):
        super(CurriculumLearningCallback, self).__init__(verbose)
        self.env = env
```

```
        self.patience = patience
        self.threshold = threshold
        self.best_mean_reward = -np.inf
        self.no_improvement_count = 0
        self.episode_rewards = []
        self.phase_change_episodes = []

    def _on_step(self) -> bool:
        if "episode" in self.locals["infos"][0]:
            reward = self.locals["infos"][0]["episode"]["r"]
            self.episode_rewards.append(reward)
            if len(self.episode_rewards) >= self.patience:
                mean_reward = np.mean(self.episode_rewards[-self.patience:])

                if mean_reward > self.best_mean_reward:
                    self.best_mean_reward = mean_reward
                    self.no_improvement_count = 0

                    if self.env.current_phase < len(self.env.curriculum) - 1:
                        self.env.current_phase += 1
                        self.env.set_current_mixture_ratio(self.env.current_phase)
                        current_episode = len(self.episode_rewards)
                        self.phase_change_episodes.append(current_episode)
                        if self.verbose:
                            print(f"Curriculum Phase Increased to {self.env.current_phase + 1} "
                                  f"at Episode {current_episode}, ratio: {self.env.get_current_mixture_ratio()}")

                else:
                    self.no_improvement_count += 1
                    if self.verbose:
                        print(f"No improvement for {self.no_improvement_count} episodes.")

                if self.no_improvement_count >= self.patience:
                    if self.verbose:
                        print("Stopping training due to no improvement.")
                    return True
        return True



def main():
    buffer_size = 10
    gateway = JavaGateway()
    game_instance = gateway.entry_point.getGame()

    env = StreetFighterEnv(game=game_instance)
    env = ObservationBufferWrapper(env, buffer_size=buffer_size)
    check_env(env)
    env.start_game_auto()
    env = Monitor(env)
    env = DummyVecEnv([lambda: env])

    eval_env_instance = StreetFighterEnv(game=game_instance)
    eval_env_instance = ObservationBufferWrapper(eval_env_instance, buffer_size=buffer_size)
    check_env(eval_env_instance)
    eval_env_instance.start_game_auto()
    eval_env = Monitor(eval_env_instance)
    eval_env = DummyVecEnv([lambda: eval_env])

    policy_kwargs = dict(
        features_extractor_class=CustomLSTMFeaturesExtractor,
        features_extractor_kwargs=dict(features_dim=256, buffer_size=buffer_size),
    )
   # Initializing the PPO model with the custom feature extractor
    model = PPO(
        "MlpPolicy",
        env,
        learning_rate=1e-4,
        n_steps=2048,
        batch_size=64,
        n_epochs=10,
        gamma=0.99,
        gae_lambda=0.95,
        clip_range=0.2,
        ent_coef=0.05,
```

```
        vf_coef=0.5,
        max_grad_norm=0.5,
        verbose=1,
        policy_kwargs=policy_kwargs,
        tensorboard_log="./ppo_sf_lstm_tensorboard/",
    )

    eval_callback = EvalCallback(
        eval_env=eval_env,
        best_model_save_path='./logs/best_model',
        log_path='./logs/evaluation',
        eval_freq=10000,
        n_eval_episodes=10,
        deterministic=True,
        render=False,
        verbose=1
    )

    curriculum_callback = CurriculumLearningCallback(env=env.envs[0], patience=500, threshold=12.0, verbose=1)

    callback = CallbackList([curriculum_callback, eval_callback])

    timesteps = 300000
    model.learn(total_timesteps=timesteps, callback=callback, log_interval=10)

    model.save("ppo_sf_lstm_v3")
    print("Model saved as 'ppo_sf_lstm_v3.zip'.")

    # Evaluating the model
    mean_reward, std_reward = evaluate_policy(model, eval_env, n_eval_episodes=10, deterministic=True)
    print(f"Mean reward: {mean_reward:.2f} +/- {std_reward:.2f}")

    plt.figure(figsize=(12, 6))
    plt.plot(range(1, len(curriculum_callback.episode_rewards) + 1),
             curriculum_callback.episode_rewards,
             marker='o', linestyle='-', color='b', label='Total Reward')

    for idx, phase_episode in enumerate(curriculum_callback.phase_change_episodes, start=1):
        plt.axvline(x=phase_episode, color='r', linestyle='--', linewidth=1)
        plt.text(phase_episode, max(curriculum_callback.episode_rewards) * 0.95,
                 f'Phase {idx}',
                 rotation=90, verticalalignment='top', color='r')

    plt.title('Training Rewards per Episode with Curriculum Phases')
    plt.xlabel('Episode')
    plt.ylabel('Total Reward')
    plt.ylim(-30, 30)
    plt.legend(['Rewards', 'Phase Changes'])
    plt.grid(True)

    plt.savefig('training_rewards_plot_curr.png')
    plt.show()


main()
```

Testing code

```
import time
import matplotlib.pyplot as plt
import pandas as pd
from gymnasium.utils.env_checker import check_env
from stable_baselines3 import PPO, DQN
from py4j.java_gateway import JavaGateway
from stable_baselines3.common.evaluation import evaluate_policy
from sb3_contrib import RecurrentPPO
import seaborn as sns
from stable_baselines3.common.vec_env import DummyVecEnv
from stable_baselines3.common.monitor import Monitor
from stable_baselines3.common.torch_layers import BaseFeaturesExtractor
import gymnasium as gym
import torch as th
from torch import nn
from StreetFighter_with_ppo import StreetFighterEnv, CustomLSTMFeaturesExtractor  # Import custom classes

import numpy as np
```

```
class ObservationBufferWrapper(gym.Wrapper):
    def __init__(self, env, buffer_size=10):
        super(ObservationBufferWrapper, self).__init__(env)
        self.buffer_size = buffer_size
        self.observation_buffer = []
        original_obs_shape = env.observation_space.shape
        new_low = np.repeat(env.observation_space.low, self.buffer_size, axis=0)
        new_high = np.repeat(env.observation_space.high, self.buffer_size, axis=0)
        self.observation_space = gym.spaces.Box(low=new_low, high=new_high, dtype=env.observation_space.dtype)

    def reset(self, **kwargs):
        observation, info = self.env.reset(**kwargs)
        self.observation_buffer = [observation] * self.buffer_size
        stacked_observation = self._get_stacked_observation()
        return stacked_observation, info

    def step(self, action):
        observation, reward, done, truncated, info = self.env.step(action)
        self.observation_buffer.pop(0)
        self.observation_buffer.append(observation)
        stacked_observation = self._get_stacked_observation()
        return stacked_observation, reward, done, truncated, info

    def _get_stacked_observation(self):
        return np.concatenate(self.observation_buffer, axis=0)

buffer_size = 10
gateway = JavaGateway()
game_instance = gateway.entry_point.getGame()
game_instance.setRenderingEnabled(True)
env = StreetFighterEnv(game=game_instance)

env = ObservationBufferWrapper(env, buffer_size=buffer_size)

check_env(env)
print('Environment created for testing.')
env.start_game_auto()

env = Monitor(env)
env = DummyVecEnv([lambda: env])

print('loading models....')

model = PPO.load("ppo_sf_lstm_v3.zip", env=env
# Initialize data tracking
num_test_episodes = 100
# Elo Rating Initialization
elo_ratings = {
    "PPO+LSTM": 1000,
    "DQN+LSTM": 1000,
    "Random": 1000
}
K_FACTOR = 32
elo_progression = {"PPO+LSTM": [], "DQN+LSTM": [], "Random": []}

# Data structures to store metrics
testing_rewards = []
old_model_rewards = []
random_rewards = []

results = {"P1 Won": 0, "P1 Lost": 0, "Draw": 0}
old_model_results = {"P1 Won": 0, "P1 Lost": 0, "Draw": 0}
random_results = {"P1 Won": 0, "P1 Lost": 0, "Draw": 0}

# Additional metrics
time_taken_current_model = []
action_efficiency_current_model = []

time_taken_old_model = []
action_efficiency_old_model = []

time_taken_random = []
action_efficiency_random = []

# Define attack actions
```

```
attack_actions = [2, 3, 4]  # Light, Medium, Heavy Attack



def calculate_expected_outcome(rating_A, rating_B):
    return 1 / (1 + 10 ** ((rating_B - rating_A) / 400))


def update_elo(rating_A, rating_B, score_A, score_B):
    expected_A = calculate_expected_outcome(rating_A, rating_B)
    expected_B = calculate_expected_outcome(rating_B, rating_A)

    new_rating_A = rating_A + K_FACTOR * (score_A - expected_A)
    new_rating_B = rating_B + K_FACTOR * (score_B - expected_B)

    return new_rating_A, new_rating_B


attack_frequency_current_model = []
attack_frequency_old_model = []
attack_frequency_random = []

def test_model(model, env, num_episodes, results_dict, rewards_list, time_taken_list, action_efficiency_list,
               attack_frequency_list, elo_name, model_name="Model"):
    print(f'Testing {model_name}...')
    for episode in range(num_episodes):
        reset_output = env.reset()
        if isinstance(reset_output, tuple):
            obs, _ = reset_output
        else:
            obs = reset_output
        total_reward = 0
        done = False
        steps = 0
        attack_count = 0

        while not done:
            if model_name == "Random Actions":

                action = env.action_space.sample()
            elif model_name == "Current PPO+LSTM Model":

                actions, _ = model.predict(obs, deterministic=True)
                action = int(actions[0])
            else:
                action, _ = model.predict(obs, deterministic=True)
                action = int(action)

            if action in attack_actions:
                attack_count += 1


            if  model_name == "Current PPO+LSTM Model":
                obs, rewards, dones, infos = env.step([action])
                total_reward += rewards[0]
                done = dones[0]
                steps += 1
            else:
                obs, reward, done, truncated, info = env.step(action)
                total_reward += reward
                steps += 1



            if done:
                if model_name == "Current PPO+LSTM Model":
                    result = infos[0].get('result', 'Draw')
                else:
                    result = info.get('result')
                if result in results_dict:
                    results_dict[result] += 1

                time_taken_list.append(steps)
                efficiency = attack_count / steps if steps > 0 else 0
                action_efficiency_list.append(efficiency)
```

```
            # Update Elo Ratings
            if result == "P1 Won":
                score_A, score_B = 1, 0
            elif result == "P1 Lost":
                score_A, score_B = 0, 1
            else:  # Draw or undefined
                score_A, score_B = 0.5, 0.5

            if model_name != "Random Actions":
                elo_ratings[elo_name], elo_ratings["Random"] = update_elo(
                    elo_ratings[elo_name], elo_ratings["Random"], score_A, score_B
                )
                elo_progression[elo_name].append(elo_ratings[elo_name])
            else:
                elo_ratings["Random"], elo_ratings[elo_name] = update_elo(
                    elo_ratings["Random"], elo_ratings[elo_name], score_A, score_B
                )
                elo_progression["Random"].append(elo_ratings["Random"])

            break

        rewards_list.append(total_reward)
        attack_frequency_list.append(attack_count)
        if (episode + 1) % 10 == 0:
            print(f"Completed Episode {episode + 1}/{num_episodes}")


test_model(
    model=model,
    env=env,
    num_episodes=num_test_episodes,
    results_dict=results,
    rewards_list=testing_rewards,
    time_taken_list=time_taken_current_model,
    action_efficiency_list=action_efficiency_current_model,
    attack_frequency_list=attack_frequency_current_model,
    elo_name="PPO+LSTM",
    model_name="Current PPO+LSTM Model"
)
print(f"\nWin/Loss/Draw for Current PPO+LSTM Model: {results}")

env = StreetFighterEnv(game=game_instance)

model_old = DQN.load("dqn_sf_lstm_v2.zip")
# Test the old DQN+LSTM model
test_model(
    model=model_old,
    env=env,
    num_episodes=num_test_episodes,
    results_dict=old_model_results,
    rewards_list=old_model_rewards,
    time_taken_list=time_taken_old_model,
    action_efficiency_list=action_efficiency_old_model,
    attack_frequency_list=attack_frequency_old_model,
    elo_name="DQN+LSTM",
    model_name="Old DQN+LSTM Model"
)

# Test random actions
test_model(
    model=None,  # Random actions do not require a model
    env=env,
    num_episodes=num_test_episodes,
    results_dict=random_results,
    rewards_list=random_rewards,
    time_taken_list=time_taken_random,
    action_efficiency_list=action_efficiency_random,
    attack_frequency_list=attack_frequency_random,
    elo_name="Random",
    model_name="Random Actions"
)

# Display Elo ratings
print("\nFinal Elo Ratings:")
```

```
for model_name, rating in elo_ratings.items():
    print(f"{model_name}: {rating}")

# Plot Elo rating progression
plt.figure(figsize=(10, 6))
for model_name, ratings in elo_progression.items():
    plt.plot(range(1, len(ratings) + 1), ratings, label=model_name)

plt.title("Elo Ratings Progression")
plt.xlabel("Match Number")
plt.ylabel("Elo Rating")
plt.legend()
plt.grid(True)
plt.savefig("elo_ratings_progression_fixed.png")
# plt.show()

# Display win/loss/draw results for all tests
print(f"\nWin/Loss/Draw for Current PPO+LSTM Model: {results}")
print(f"Win/Loss/Draw for Old DQN+LSTM Model: {old_model_results}")
print(f"Win/Loss/Draw for Random Actions: {random_results}")

# Plot Win/Loss/Draw results for all comparisons
fig, axes = plt.subplots(1, 3, figsize=(18, 5))

# Current Model Win/Loss/Draw
axes[0].bar(results.keys(), results.values(), color=['green', 'red', 'blue'])
axes[0].set_title('Win/Loss/Draw (Current PPO+LSTM Model)')
axes[0].set_xlabel('Result')
axes[0].set_ylabel('Number of Episodes')
axes[0].grid(True)

# Old Model Win/Loss/Draw
axes[1].bar(old_model_results.keys(), old_model_results.values(), color=['green', 'red', 'blue'])
axes[1].set_title('Win/Loss/Draw (Old DQN+LSTM Model)')
axes[1].set_xlabel('Result')
axes[1].set_ylabel('Number of Episodes')
axes[1].grid(True)

# Random Actions Win/Loss/Draw
axes[2].bar(random_results.keys(), random_results.values(), color=['green', 'red', 'blue'])
axes[2].set_title('Win/Loss/Draw (Random Actions)')
axes[2].set_xlabel('Result')
axes[2].set_ylabel('Number of Episodes')
axes[2].grid(True)

plt.tight_layout()
plt.savefig('comparison_win_loss_draw_modelVsmodel.png')
# plt.show()

# Box Plot Comparison of Rewards for all three methods
data_rewards = [testing_rewards, old_model_rewards, random_rewards]
labels_rewards = ['PPO+LSTM Model', 'DQN+LSTM Model', 'Random Actions']

plt.figure(figsize=(12, 6))
plt.boxplot(data_rewards, patch_artist=True, labels=labels_rewards,
            boxprops=dict(facecolor="lightgreen", color="black"),
            medianprops=dict(color="orange"))
plt.title('Reward Distribution: PPO+LSTM vs DQN+LSTM vs Random Actions')
plt.xlabel('Model')
plt.ylabel('Total Reward per Episode')
plt.grid(True)
plt.savefig('comparison_rewards_boxplot_modelVsmodel.png')
# plt.show()


data_time = [time_taken_current_model, time_taken_old_model]
labels_time = ['PPO+LSTM Model', 'DQN+LSTM Model']

plt.figure(figsize=(12, 6))
plt.boxplot(data_time, patch_artist=True, labels=labels_time,
            boxprops=dict(facecolor="lightblue", color="black"),
            medianprops=dict(color="orange"))
plt.title('Time Taken to Win: PPO+LSTM vs DQN+LSTM Models')
plt.xlabel('Model')
plt.ylabel('Number of Steps to Win')
```

	plt.grid(True)
	plt.savefig('comparison_time_taken_modelVsmodel.png')
	# plt.show()
	
	# Box Plot Comparison of Action Efficiency for all three methods
	data_efficiency = [action_efficiency_current_model, action_efficiency_old_model, action_efficiency_random]
	labels_efficiency = ['PPO+LSTM Model', 'DQN+LSTM Model', 'Random Actions']
	
	plt.figure(figsize=(12, 6))
	plt.boxplot(data_efficiency, patch_artist=True, labels=labels_efficiency,
	boxprops=dict(facecolor="lightcoral", color="black"),
	medianprops=dict(color="orange"))
	plt.title('Action Efficiency: PPO+LSTM vs DQN+LSTM vs Random Actions')
	plt.xlabel('Model')
	plt.ylabel('Action Efficiency (Attack Actions / Total Actions)')
	plt.grid(True)
	plt.savefig('comparison_action_efficiency_modelVsmodel.png')
	# plt.show()
	
	action_reward_data = {
	'Action Frequency': attack_frequency_current_model + attack_frequency_old_model + attack_frequency_random,
	'Reward': testing_rewards + old_model_rewards + random_rewards,
	'Model': (['PPO+LSTM'] * len(attack_frequency_current_model) +
	['DQN+LSTM'] * len(attack_frequency_old_model) +
	['Random'] * len(attack_frequency_random))
	}
	
	assert len(action_reward_data['Action Frequency']) == len(action_reward_data['Reward'])
	assert len(action_reward_data['Reward']) == len(action_reward_data['Model'])
	
	action_reward_df = pd.DataFrame(action_reward_data)
	
	pivot_table = action_reward_df.pivot_table(
	values='Reward',
	index='Model',
	columns='Action Frequency',
	aggfunc='mean',
	)
	
	# Plot the heatmap
	plt.figure(figsize=(10, 6))
	sns.heatmap(pivot_table, cmap='coolwarm', annot=True, fmt='.2f', cbar_kws={'label': 'Average Reward'})
	plt.title('Heatmap of Action Frequency vs Reward')
	plt.xlabel('Action Frequency')
	plt.ylabel('Model')
	plt.savefig('heatmap_action_frequency_vs_reward_fixed.png')
	# plt.show()
	
	time_data = {
	'Model': ['PPO+LSTM'] * len(time_taken_current_model) + ['DQN+LSTM'] * len(time_taken_old_model) + ['Random'] * len(time_taken_random),
	'Steps': time_taken_current_model + time_taken_old_model + time_taken_random
	}
	
	time_df = pd.DataFrame(time_data)
	
	# Plot
	plt.figure(figsize=(10, 6))
	sns.violinplot(data=time_df, x='Model', y='Steps', inner="quartile", palette='muted')
	plt.title('Episode Length Distribution: PPO+LSTM vs DQN+LSTM vs Random Actions')
	plt.xlabel('Model')
	plt.ylabel('Number of Steps')
	plt.grid(True)
	plt.savefig('episode_length_violin_modelVsmodel.png')
	# plt.show()

Model vs human	import time import matplotlib.pyplot as plt import numpy as np from gymnasium.utils.env_checker import check_env

	from stable_baselines3 import PPO
	from py4j.java_gateway import JavaGateway
	from stable_baselines3.common.evaluation import evaluate_policy
	
	from StreetFighter_tk import StreetFighterEnv
	gateway = JavaGateway()
	game_instance = gateway.entry_point.getGame()
	game_instance.setRenderingEnabled(True)
	env = StreetFighterEnv(game=game_instance)
	print('Environment created for testing.')
	
	model = PPO.load("ppo_street_fighter_ppo")
	env.start_game_auto()
	
	id = 'az2'
	num_test_episodes = 10
	testing_rewards = []
	old_model_rewards = []
	random_rewards = []
	results = {"P1 Won": 0, "P1 Lost": 0, "Draw": 0}
	
	for episode in range(num_test_episodes):
	  obs, _ = env.reset()
	  total_reward = 0
	  done = False
	  while not done:
	  if np.random.rand() < 0.2:
	  action = env.action_space.sample()
	  else:
	  action, _ = model.predict(obs, deterministic=True)
	  obs, reward, done, truncated, info = env.step(int(action))
	  total_reward += reward
	  if done:
	  result = info.get('result')
	  if result in results:
	  results[result] += 1
	  testing_rewards.append(total_reward)
	
	# Plotting Results
	# Bar graph for win/loss/draw
	plt.figure(figsize=(10, 6))
	plt.bar(results.keys(), results.values(), color=['green', 'red', 'blue'], alpha=0.7)
	plt.title("Win/Loss/Draw Results for Current Model")
	plt.ylabel("Number of Episodes")
	plt.xlabel("Result")
	plt.grid(axis='y', linestyle='--', alpha=0.7)
	plt.savefig(f'winLoss_subject_{id}.png')
	plt.show()
	
	# Box plot for rewards
	
	plt.figure(figsize=(10, 6))
	plt.boxplot(testing_rewards, vert=True, patch_artist=True,
	  boxprops=dict(facecolor='orange', color='black'),
	  medianprops=dict(color='black'))
	plt.title("Box Plot of Rewards for Current Model")
	plt.ylabel("Rewards")
	
	plt.savefig(f'rew_subject_{id}.png')
	plt.show()
	
	print("Win/Loss/Draw Results:", results)
	print("Reward Summary:")
	print(f" Mean Reward: {np.mean(testing_rewards):.2f}")
	print(f" Max Reward: {np.max(testing_rewards):.2f}")
	print(f" Min Reward: {np.min(testing_rewards):.2f}")

Elo ratings	import time import matplotlib.pyplot as plt from gymnasium.utils.env_checker import check_env from stable_baselines3 import PPO, DQN from py4j.java_gateway import JavaGateway from StreetFighter_with_ppo import import seaborn as sns

```
gateway = JavaGateway()
game_instance = gateway.entry_point.getGame()
game_instance.setRenderingEnabled(True)
env = StreetFighterEnv(game=game_instance)
check_env(env)
print('Environment created for testing.')
num_test_episodes = 100

model = PPO.load("best_model.zip")
model_old = DQN.load("dqn_sf_lstm_v2.zip")
env.start_game_auto()
K_FACTOR = 32


# Function to calculate expected outcome
def calculate_expected_outcome(rating_A, rating_B):
    return 1 / (1 + 10 ** ((rating_B - rating_A) / 400))


# Function to update Elo ratings
def update_elo(rating_A, rating_B, score_A, score_B):
    expected_A = calculate_expected_outcome(rating_A, rating_B)
    expected_B = calculate_expected_outcome(rating_B, rating_A)

    new_rating_A = rating_A + K_FACTOR * (score_A - expected_A)
    new_rating_B = rating_B + K_FACTOR * (score_B - expected_B)

    return new_rating_A, new_rating_B


def test_model_v3(model_p2, model_name, env, num_episodes):

    elo_ratings = {"P1": 1200, f"P2_{model_name}": 1000}
    elo_progression = {"P1": [], f"P2_{model_name}": []}

    print(f"Testing Player 2 ({model_name}) vs Player 1...")
    for episode in range(num_episodes):
        obs, _ = env.reset()
        total_reward = 0
        done = False

        while not done:
            # Player 2 takes an action
            action_p2 = env.action_space.sample() if model_p2 is None else model_p2.predict(obs, deterministic=True)[0]
            action_p2 = int(action_p2)

            obs, reward, done, truncated, info = env.step(action_p2)
            total_reward += reward

            if done:
                result = info.get('result')
                if result == "P1 Won":
                    score_p2, score_p1 = 0, 1
                elif result == "P1 Lost":
                    score_p2, score_p1 = 1, 0
                else:  # Draw
                    score_p2, score_p1 = 0.5, 0.5

                # Update Elo ratings
                elo_ratings[f"P2_{model_name}"], elo_ratings["P1"] = update_elo(
                    elo_ratings[f"P2_{model_name}"], elo_ratings["P1"], score_p2, score_p1
                )

                elo_progression[f"P2_{model_name}"].append(elo_ratings[f"P2_{model_name}"])
                elo_progression["P1"].append(elo_ratings["P1"])
                break

    # Plot Elo ratings for this configuration
    plt.figure(figsize=(10, 6))
    plt.plot(range(1, len(elo_progression[f"P2_{model_name}"]) + 1), elo_progression[f"P2_{model_name}"],
             label=f"{model_name} (Player 2)", color="blue")
    plt.plot(range(1, len(elo_progression["P1"]) + 1), elo_progression["P1"], label="Player 1 (Environment)",
             color="red", linestyle="--")
```

```
    plt.title(f"Elo Ratings Progression: {model_name} vs Player 1")
    plt.xlabel("Match Number")
    plt.ylabel("Elo Rating")
    plt.legend()
    plt.grid(True)
    plt.savefig(f"elo_ratings_{model_name}_vs_P1.png")
    # plt.show()

    return elo_progression



elo_progression_ppo = test_model_v3(model, "PPO+LSTM", env, num_test_episodes)
elo_progression_dqn = test_model_v3(model_old, "DQN+LSTM", env, num_test_episodes)
elo_progression_random = test_model_v3(None, "Random", env, num_test_episodes)


# Density plot function
def plot_elo_density(elo_ratings_p1, elo_ratings_p2, model_name):
    plt.figure(figsize=(10, 6))

    # KDE plots for Elo ratings
    sns.kdeplot(elo_ratings_p1, label="Player 1 (P1)", color="red", fill=True, alpha=0.4)
    sns.kdeplot(elo_ratings_p2, label=f"Player 2 ({model_name})", color="blue", fill=True, alpha=0.4)

    plt.title(f"Elo Rating Density: P1 vs {model_name}")
    plt.xlabel("Elo Rating")
    plt.ylabel("Density")
    plt.legend()
    plt.grid(True)
    plt.savefig(f"elo_density_P1_vs_{model_name}.png")
    # plt.show()


# Plot density after collecting data
plot_elo_density(elo_progression_ppo["P1"], elo_progression_ppo["P2_PPO+LSTM"], "PPO+LSTM")
plot_elo_density(elo_progression_dqn["P1"], elo_progression_dqn["P2_DQN+LSTM"], "DQN+LSTM")
plot_elo_density(elo_progression_random["P1"], elo_progression_random["P2_Random"], "Random")
```