SC4079 Final Year Project

Title:

dAI_GO: Artificial Intelligence for Fighting Games

Final Report

October 2024

Supervisor: Dr Seah Hock Soon

Done By: Ryan Teo Cher Kean (U2122540D)

Table of Contents

Table of Figures

Abstract

dAI_GO investigates the implementation of an Actor-Critic (AC) model for training an artificial intelligence (AI) agent in a fighting game environment using the IkemenGO engine. The purpose is to develop agents capable of engaging in human-like adaptive behaviour during gameplay. The agent is intended to balance short-term rewards such as successful attacks with longer-term rewards such as winning a round. The AC model leverages policy optimization through the Actor, while also taking advantage of value estimation via the Critic, making the AC model suitable for handling the dual objectives of the agent.

Training was conducted using GPU-accelerated PyTorch models while incorporating logit manipulation techniques such as temperature scaling and bias addition to influence the agent's behaviour to create human-like agents. Additional experiments were conducted to investigate agent performance under various conditions, such as introducing biases towards certain actions or introducing behavioural asymmetries to the agents. Rewards were structured to reflect relevant changes in the game state such as health percentage, positioning and combos.

Experiments were conducted to explore how different training conditions—such as biases towards specific behaviours or segregated movement actions—affected performance. The results demonstrated notable progress, with agents showing promising adaptive play against built-in CPUs but sometimes struggling against human opponents due to certain exploitable patterns. Ultimately, the project highlights the potential for using actor-critic models in game AI and offers insights into future improvements, such as enhancing real-time decision-making, refining rewards, and incorporating imitation learning for more complex behaviours.

1 Introduction

1.1 Background and motivation

Fighting games constitute one of the most intricate and strategically demanding genres within the landscape of video games. Renowned for their combination of technical execution and strategic depth, titles such as the *Street Fighter*, *Super Smash Brothers*, and *Tekken* series have established themselves as cornerstones of competitive e-sports (ESPORTS CHARTS, n.d.). These games not only offer fast-paced, one-on-one combat at the highest levels of competition but also have become deeply ingrained in global pop culture, captivating millions of players worldwide (Famutimi, 2023).

The appeal of fighting games lies in their ability to foster dynamic and competitive environments where players must continually adapt their strategies to counteract their opponents. Unlike other genres, where enemies may follow predictable attack patterns, fighting games require players to anticipate, react, and execute with split-second precision. This demands an elevated level of technical skill, particularly in games like *Street Fighter*, where the execution of complex inputs within narrow time frames can determine the outcome of a match. Consequently, the learning curve for these games is often steep, necessitating extensive practice to master both the mechanical and strategic elements.

To aid in skill development, fighting games typically include practice modes, enabling players to refine their technical abilities in a controlled setting. Additionally, these games often feature computer-controlled opponents, commonly referred to as "CPU opponents," designed to simulate the experience of competing against another player. However, despite their utility, CPU opponents frequently fall short in replicating the nuanced decision-making processes of human players. Experienced human players are adept at recognizing and exploiting their opponent's habits, making strategic choices that are influenced by factors such as timing, psychological pressure, and risk assessment. Conversely, CPU opponents, while technically flawless, often lack the ability to mimic the adaptive and sometimes unpredictable nature of human gameplay. Furthermore, human players are constrained by biological factors, such as reaction time and executional consistency, which CPU opponents do not have to contend with. Additionally, CPU opponents tend to fall into certain patterns, where players can take certain actions to exploit those patterns. This disparity creates a gap in the training tools available to players, limiting their ability to practice and hone their competitive skills effectively.

In recent years, the rapid advancements in artificial intelligence (AI) have sparked significant interest in its potential applications to the domain of fighting games. AI has the capacity to bridge the gap between the rigid behaviour of CPU opponents and the adaptive, strategic play of human opponents. By developing AI that can more accurately emulate the experience of playing against a human opponent, it is possible to create training tools that are both challenging and beneficial for players seeking to improve their competitive performance.

This project, *dAI_GO* (Named after a famous *Street Fighter* player), is motivated by the desire to enhance the realism and effectiveness of AI opponents in fighting games. The goal is to develop an AI system capable of replicating the complex decision-making and adaptive strategies characteristic of human players. By achieving this, *dAI_GO* aims to provide players with a more authentic and valuable practice experience, ultimately contributing to the broader field of AI in gaming and offering new opportunities for innovation in both game design and competitive training.

1.2 Objective and Scope

The objective of this project is to design and implement an AI powered CPU opponent that as accurately as possible reflects the experience of engaging a human opponent. The AI will be based on an Actor-Critic model, which is suited for real-time decision-making processes. The AI is intended to be able to replicate complex strategies and adaptations that are characteristic of human players.

To achieve this, the project will focus on:

1. **Behavioural modelling:** The AI must be able to demonstrate human like characteristics such as pattern recognition and adaptation.
2. **Real-time adaptation:** Decisions must be made in real-time and in various scenarios.
3. **Human constraints:** The AI should display some form of human constraints, such as reactions time and non-frame-perfect inputs.

The project will be done based on a fighting game called IkemenGO, based off another fighting game called M.U.G.E.N. IkemenGO is lightweight, open source, and written using the GO programming language, making it suitable for the project The scope of the project includes the development of the said AI system, mainly using Python and GO.

2 Related works

The application of AI in video games has been explored in the past, with various methods evolving over time to enhance gameplay and player experience. Early video games relied heavily on rule-based AI systems, where agents followed predefined rules to make decisions. A well-known example is the behaviour of the ghosts in *Pac-Man*, where each ghost's movement is determined based on the player's position on the screen, showcasing a basic but effective form of game AI (Pittman, 2015). These systems, while limited, laid the foundation for future developments in game AI.

Over time, game AI progressed towards more sophisticated techniques such as finite state machines (FSMs), where agents transition between different predefined states based on specific triggers. FSMs allowed for more complex decision-making, as seen in games like DOOM, where enemies switch between states like patrolling, chasing, or attacking based on the player's actions (Thompson, 2022). FSMs represented a shift from static rules to more adaptive behaviour that reacts to player inputs.

More recently, machine learning techniques gained prominence. These models enable agents to learn by interacting with their environment and receiving rewards or penalties based on the outcomes of their actions. This shift reflects the desire for AI that can dynamically adjust, offering deeper and more engaging gameplay experiences. Additionally, methods such as player imitation aim to create AI agents that mimic human behaviours by analysing past gameplay data. However, these systems still face limitations in adaptability and struggle to respond effectively to unexpected changes in human strategies.

2.1 Reinforcement Learning (RL) Approach

Mendonca et al (2015) explore the use of RL techniques to simulate human behaviour. RL refers to a technique where an agent learns to solve a problem based on experiences and past actions. Mendonca et al developed a special reward function that allowed the agent to present human-like behaviour.

2.2 Genetic Programming (GP) Approach

Martínez-Arellano et al (2017) took a GP approach to building an AI model for fighting games. GP is a AI training method inspired by the principles of biological evolution. At its core, it operates on the idea of "survival of the fittest"—selecting and evolving solutions over successive generations. In the context of AI, agents are initially assigned random behaviours or actions. These agents then explore their environment, and those that perform well, according to a fitness function, are retained and combined to generate new, improved offspring solutions. Through mutation, crossover, and selection mechanisms, GP refines agents' performance over time, driving them closer to the desired outcome or optimal behaviour. This iterative process mimics natural evolution, ensuring that only the most effective strategies persist and evolve in subsequent generations.

Martínez-Arellano et al made use of the M.U.G.E.N engine engine to implement their AI. Each agent is made by a set of sequential blocks that determine the fighting strategy Again, these methods are intended to create optimal agents rather than agents that simulate human-like behaviour.

2.3 K-Nearest-Neighbours (KNN) Approach

Yamamoto et al (2014) explore the use of the KNN algorithm to develop AI that is able to choose actions from a variety of action patterns. The KNN approach is a machine learning technique commonly used for classification and pattern recognition. It works by identifying the k nearest data points (neighbours) to a given input and assigning the most frequent label or averaging the values from those neighbours. Yamamoto et al utilised the FightingICE platform, originally developed for usage in an international competition that competes for the performance of fighting game AI. The team surmised that the KNN approach would be an effective way to predict an opponent's next attack and deduce the most reasonable countermeasure.

2.4 Player Imitation Approach

Griebsch (2022) explores the use of player imitation as a technique to simulate human-like behaviour in AI agents. Player imitation involves training an agent to replicate the behaviour of a specific player by mimicking their actions in similar situations. In essence, the AI aims to make the same decisions that the player would have made when presented with identical circumstances. Griebsch utilizes the IkemenGO engine to implement this concept, employing case-based reasoning (CBR) as the core mechanism. While CBR is simpler than neural networks, it remains an effective approach for guiding agents in decision-making based on previous experiences. However, one key limitation of player imitation is that it lacks adaptability; the AI agent selects actions purely based on the current situation without adjusting dynamically to specific opponents or evolving strategies over time.

3 System design

3.1 IkemenGO environment analysis

IkemenGO is an open source 2D fighting game engine, derived from the original M.U.G.E.N engine. It allows for extensive customization, including character creation, stage design, and gameplay mechanics.

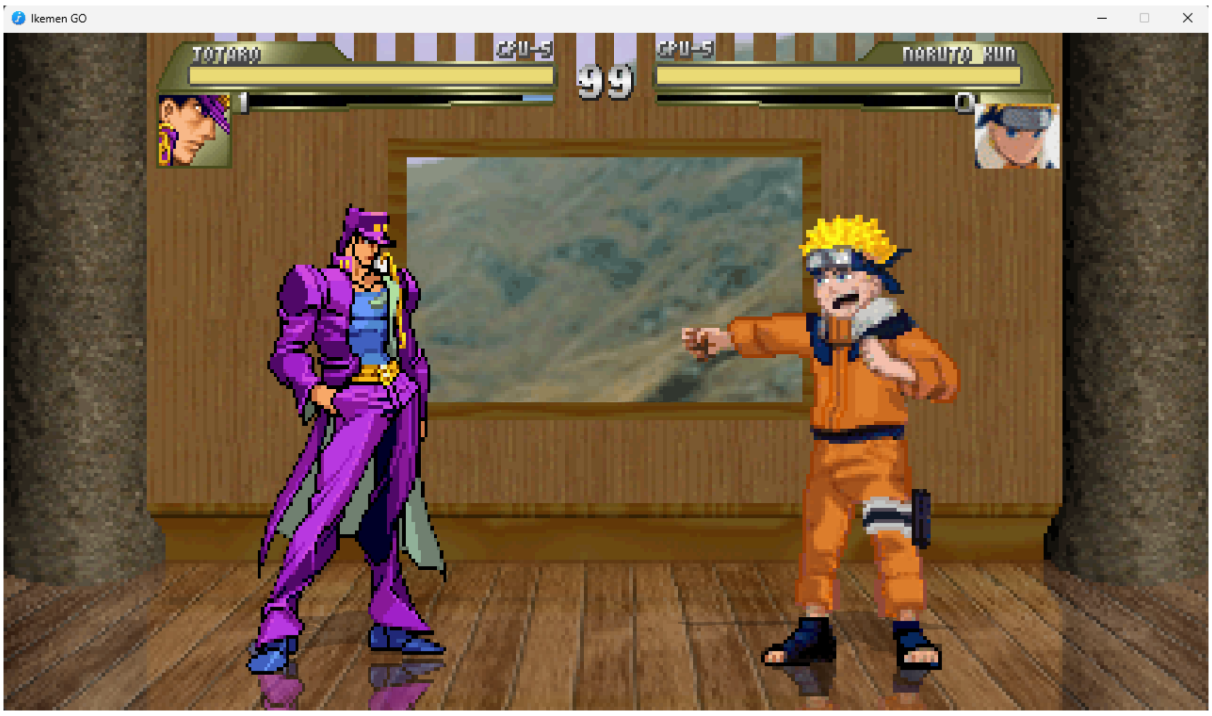


Figure 1 - Screenshot of IkemenGO featuring custom-made characters

3.1.1 Gameplay mechanics

IkemenGO supports a wide range of character attributes, including move sets, animations, and hitboxes. Custom character creation is also supported, allowing for players to design their own unique characters and move sets. Users can define character behaviour using scripts, which makes it possible to define CPU opponents.

IkemenGO offers the basic fighting game mechanics, such as combo systems, special moves, and power gauges (meter). At the same time, it also supports custom mechanics which can be tailored to the specific needs of the character or stage being developed. For example, if you want your character to be able to perform special actions such as transforming mid-game, it is possible to implement.

IkemenGO's action space typically involves some combination of button inputs. These button inputs are translated into actions on the screen, including but not limited to:

- **Movements:** Basic actions like moving forward, backward, jumping, dashing and crouching.
- **Attacks:** Various attack types, including punches, kicks, special moves, and combos.

- **Defensive Actions:** Blocking and dodging.
- **Advanced actions:** Various advanced actions such as input buffering and attack cancelling. There are also character specific advanced actions
- **Special directional inputs:**
    - **Quarter Circle Forward (QCF):** Joystick is moved from neutral > down > down forward > forward.

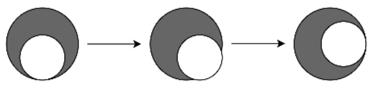


Figure 2 - Quarter Circle Forward input

    - **Quarter Circle Backward (QCB):** Joystick is moved from neutral > down > down backward > backward.

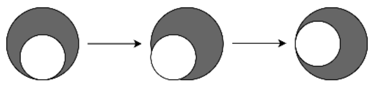


Figure 3 - Quarter Circle Backward input

    - **Dragon Punch (DP):** Joystick is moved from neutral > forward > down > down forward.

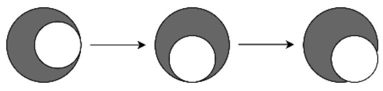


Figure 4 - Dragon Punch input

3.1.2 Characters

For this project, only the base character, Kung Fu Man (KFM), was used. KFM excels at chaining moves smoothly into one another, offering a versatile combat style. This flexibility enables dynamic gameplay and allows for creative combo routes, making KFM an ideal choice for testing and training AI agents to understand complex fighting mechanics.

Figure 5 - Kung Fu Man

KFM's move list comprises of the following actions:

Action name	Behaviour	Inputs
Standard Movement		
Right	KFM moves to the right.	Right
Left	KFM moves to the left.	Left
Jump	KFM jumps vertically.	Up
Crouch	KFM crouches in place.	Down
Forward Jump	KFM jumps diagonally forward.	[Forward Direction] + Up
Backward Jump	KFM jumps diagonally backwards.	[Backward Direction] + Up
Forward Dash	KFM dashes forward. KFM continues dashing if the input is held.	[Forward Direction] + [Forward Direction] in rapid succession
Backward Dash	KFM hops backwards.	[Backward Direction] + [Backward Direction] in rapid succession
Double Jump	KFM jumps, then performs a second jump mid-air.	Up + Up with varied timing
Hold	KFM returns to the neutral position.	No input / release all inputs
Basic Attacks		
Light Punch	KFM performs a quick punch directly forward.	X
Medium Punch	KFM performs a hook.	Y
Light Kick	KFM performs a quick low kick.	A
Medium Kick	KFM performs a kick directly forward.	B
Crouching Light Punch	KFM crouches and performs a quick punch directly forward.	Down + X

Crouching Medium Punch	KFM crouches and performs a punch diagonally upwards	Down + Y
Crouching Light Kick	KFM crouches and performs a low kick.	Down + A
Crouching Medium Kick (Sweep kick)	KFM crouches and performs a sweeping kick that topples opponents if successful.	Down + B
Air Light Punch	KFM performs a quick punch mid-air.	Up (+ Left / Right) + X
Air Medium Punch	KFM performs a karate chop mid-air.	Up (+ Left / Right) + Y
Air Light Kick	KFM performs a knee attack mid-air.	Up (+ Left / Right) + A
Air Medium Kick (Jumping Kick	KFM performs a kick mid-air.	Up (+ Left / Right) + B
Throw	KFM throws his opponent. This action bypasses guarding.	[Forward Direction]/ [Backward Direction] + y (when close)
Special attacks		
Weak Kung Fu Palm	KFM performs a quick palm attack forward.	QCF + X
Strong Kung Fu Palm	KFM performs a palm attack forward.	QCF + Y
Fast Kung Fu Palm	KFM performs a quick palm attack forward. This action uses 1/3 of KFM's meter.	QCF + XY
Weak Kung Fu Upper	KFM performs a quick uppercut.	DP + X
Strong Kung Fu Upper	KFM performs an uppercut.	DP + Y
Fast Kung Fu Upper	KFM performs a quick uppercut. This action uses 1/3 of KFM's meter.	DP + XY

Weak Kung Fu Blow	KFM performs a palm attack forward and backward.	QCB + X
Strong Kung Fu Blow	KFM performs a slow palm attack forward and backward.	QCB + Y
Fast Kung Fu Blow	KFM performs a palm attack forward and backward. This action uses 1/3 of KFM’s meter.	QCB + XY
Weak Kung Fu Zankou	KFM performs a quick shoulder attack forward.	QCF + A
Strong Kung Fu Zankou	KFM performs a shoulder attack forward.	QCF + B
Fast Kung Fu Zankou	KFM performs a quick shoulder attack forward. This action uses 1/3 of KFM’s meter.	QCF + AB
Weak Kung Fu Knee	KFM performs a quick knee attack forward. If A or B is pressed during this attack, KFM performs an extra kick.	Dash + A (+ A / B)
Strong Kung Fu Knee	KFM performs a knee attack forward. If A or B is pressed during this attack, KFM performs an extra kick.	Dash + B (+ A / B)
Fast Kung Fu Knee	KFM performs a quick knee attack forward. This action uses 1/3 of KFM’s meter. If A or B is pressed during this attack, KFM performs an extra kick.	Dash + AB (+ A / B)
Super Attacks		
Triple Kung Fu Palm	KFM performs three palm attacks forward.	QCF + QCF + X / Y

Smash Kung Fu Upper	KFM performs a powerful uppercut.	QCB + QCB + X/ Y
Defensive actions		
Standing block	KFM guards against high and middle attacks.	[Backwards direction]
Crouching block	KFM guards against middle and low attacks.	Down + [Backwards direction]
Ukemi (Technical)	KFM recovers quickly from getting hit.	XY

Figure 6 - KFM action list

This list of actions defines the action space for the AI agent to take. Button inputs (Up, Down, A, B, etc.) are mapped to specific keys on a standard keyboard. Special inputs such as the QCF and DP are simulated using arrow keys.

3.1.3 Stages

The base IkemenGO game features 4 stages:

1. **Stage 1: Training Stage**

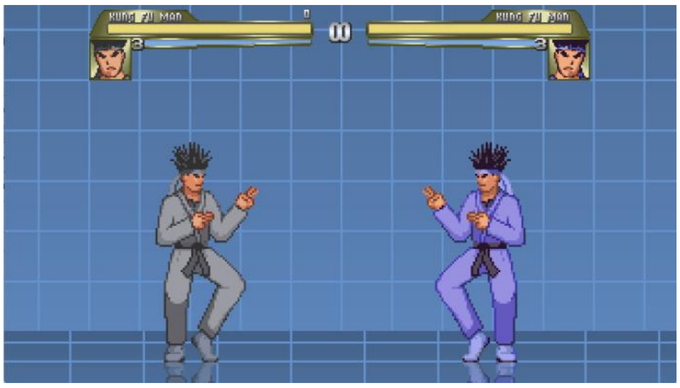


Figure 7 - Stage 1: Training Stage

2. **Stage 2: Mountainside Temple**

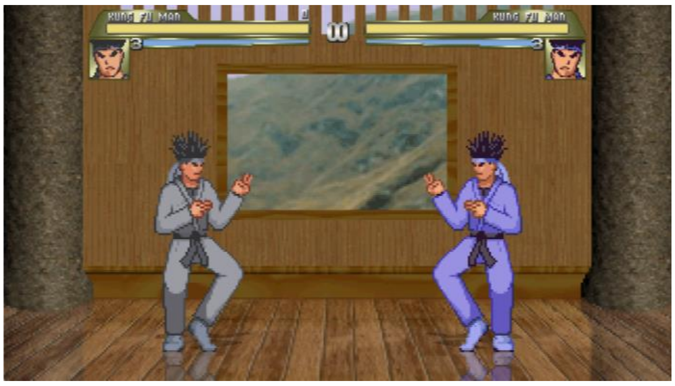


Figure 8 - Stage 2: Mountainside Temple

3. **Stage 3: Training Room**

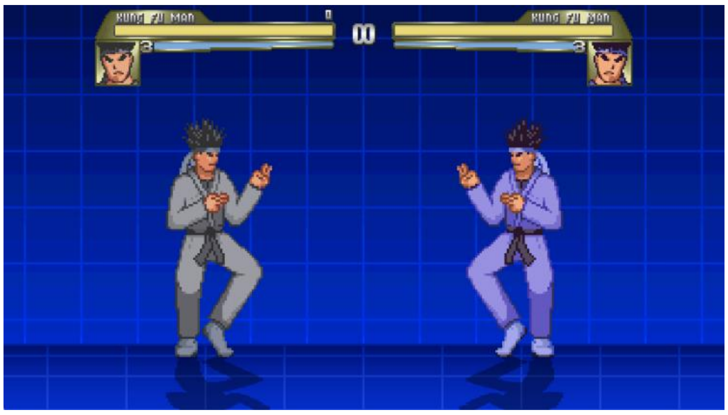


Figure 9 - Stage 3: Training Room

4. **Stage 4: Training Room HD**

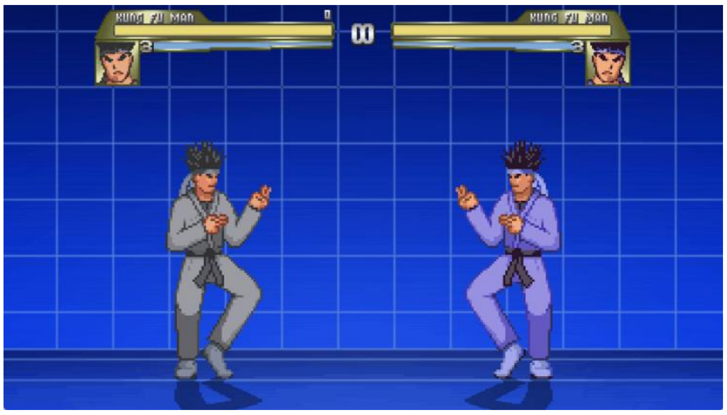


Figure 10 - Stage 4: Training Room HD

Each has the same boundaries in the X and Y directions, and unlike some other games (such as *Mortal Kombat* and *Tekken*), the stages do not offer any special interactions. Essentially, the different stages only have cosmetic differences. While this may seem trivial, it is essential for ensuring a symmetric environment for training. For example, if there was some sort of stage element that allowed for the agent on the left to have some sort of advantage (e.g. some item that could be used for combos), then training may be heavily skewed in that agent's favour. This symmetry also allows for the introduction of asymmetry in the model, which will be covered in a later section.

3.2 Architecture

The architecture of the system is as follows:

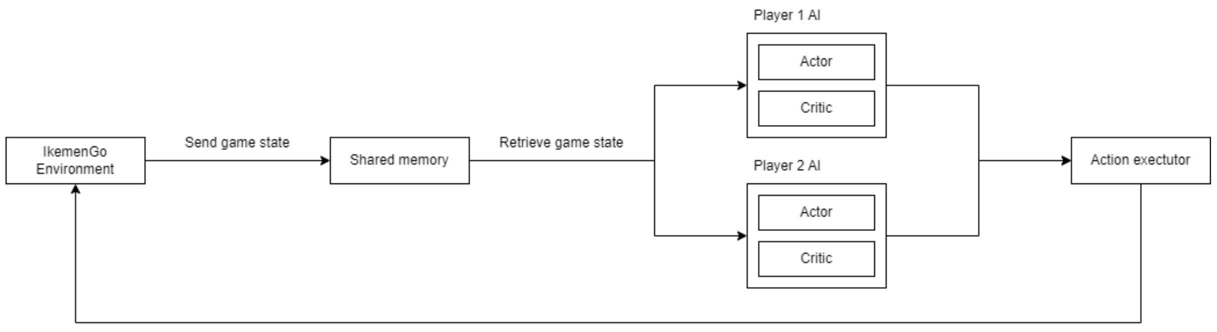


Figure 11 - System architecture diagram

The system features 2 agents running independent AI model. The agents play against each other, refining and optimising their policy as episodes go on. This style of training is known as **self-play**.

Firstly, a round is started in the IkemenGO game environment. A round in the context of a standard IkemenGO game refers to a 100 second match between 2 characters (Player or CPU). A round ends once:

1. One character's life meter is reduced to zero or
2. The 100 second timer reaches 0.

If the timer runs out, the side with the higher remaining HP is considered as the winner. Throughout the duration of the round, the state of the game is continuously updated in a shared memory space, allowing the AI system instant access to the games state at any given moment. The shared memory space is controlled by a mutex, preventing race conditions during reading and writing.

The shared memory space allows both AI agents to access detailed information about the two characters in the game. The game state is represented through several structured components, capturing the relevant gameplay details:

- **CharacterState:** Tracks the player's number, movement state, attack state, and whether the character is controllable.
- **PlayerInput:** Records the player's inputs and the character's facing direction.
- **FrameData:** Captures the frame of the current move and a reference ID for tracking ongoing animations.
- **Meters:** Represents essential status values such as life percentage, meter (energy) percentage, guard points, and dizzy levels.
- **Velocity:** Monitors both horizontal and vertical movement speeds of the characters.

- **AttackHit:** Indicates whether a move has hit, was guarded, and the current combo count.
- **Position:** Stores the character's X and Y coordinates within the game environment.

The AI system is built on an Actor-Critic model, which consists of two key components: the Actor and the Critic. The Actor observes the current game state and selects the appropriate action to take, based on learned policies. Meanwhile, the Critic evaluates the Actor's chosen action by estimating the value of the resulting state, helping the system understand how favourable the action was.

The action chosen by the agent is then sent to the action executor code, which simulates keystrokes to control the characters in game. These keystrokes are sent back to the IkemenGO environment, and the process repeats for the duration of the training cycle.

3.3 Software and Tools Used

3.3.1 Python

Python was the primary programming language used for developing and implementing the AI framework. Its versatility and extensive ecosystem of libraries made it suitable for building, training, and testing reinforcement learning agents. Below are some key libraries used:

1. **PyTorch**
   - A powerful open-source machine learning library for deep learning tasks, providing tools for building dynamic computational graphs and high-level neural network modules.
   - PyTorch enabled the design and training of the Actor-Critic model, with its ability to seamlessly integrate GPU acceleration.
   - Used for implementing loss functions, backpropagation, and model optimization.
   - Its flexibility allowed for quick prototyping and debugging, essential for the iterative nature of reinforcement learning.
2. **PyAutoGUI**
   - A cross-platform GUI automation library used to simulate keystrokes and mouse movements.
   - PyAutoGUI allowed the agents to interact with the game by sending real-time input commands (e.g., simulating keypresses for movement and attacks).
   - This enabled seamless coordination between the reinforcement learning model and the game environment.
3. **CUDA**
   - CUDA provided access to GPU acceleration on an NVIDIA RTX 3060, significantly improving model training speeds.
   - By enabling parallel computation, CUDA reduced the time required to compute gradients and perform matrix operations.

3.3.2 Golang

Golang was the primary language used for the IkemenGO game engine. As IkemenGO serves as the testbed for the AI agents, minor modifications were made to the engine:

1. **State Management:** Modifications were implemented to allow real-time writing and reading of the game's internal state to shared memory.
2. **Performance Optimization:** The language's efficient concurrency model ensured that game logic and interactions with the AI system ran smoothly.
3. **Interfacing with Python:** Data exchanges between Golang and Python occurred via shared memory, enabling synchronization between the game engine and AI processes.

3.3.3 Windows API

The Windows API was extensively used for interacting with game processes and managing shared memory. Key functionalities included:

1. **Shared Memory Management:** The AI agents accessed and modified the game state by reading from and writing to a shared memory space allocated using the Windows API.
2. **Process Control:** Used to detect and manipulate windows processes, including launching IkemenGO and managing key inputs.
3. **Keyboard and Input Simulation:** In conjunction with PyAutoGUI, the Windows API ensured precise control over in-game actions.

3.3.4 Git

Git was employed for version control to manage different versions of the AI models and codebase

3.3.5 VSCode

VSCode was the primary Integrated Development Environment (IDE) for this project:

1. **Code Development and Debugging:** The built-in debugging tools helped identify and resolve issues efficiently.
2. **Extensions:** Extensions for Python and Golang were utilized to streamline the coding process.
3. **Git Integration:** Version control through Git was integrated directly within the IDE for ease of use.

4 Model Implementation and Training

4.1 Actor-Critic model in depth

The AC model is suitable for this project as it combines policy learning and value estimation. This is particularly useful in fighting games, where both short- and long-term rewards are necessary to ensure the agent performs well. The Actor helps to balance the exploration of short-term rewards such as attacking and defending, while the Critic long term rewards like positioning and eventually winning the game.

4.1.1 Mathematical Framework

The AC model is defined by the following mathematical framework (Mendonca et al., 2015):

1. **Policy Gradient**

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta \log \pi_\theta(a_i|s_i) \cdot A(s_i, a_i)$$

   a. $J(\theta)$ defines the expected return of an action.
   b. $\pi_\theta(a_i|s_i)$ defines the probability of taking the action a given the state s.
   c. N defines the number of sampled experiences.
   d. $A(s_i, a_i)$ defines the advantage function. It represents the advantage of taking an action a given the state s.
   e. i defines the index of the sample.

   The policy gradient in the context of the AC model is handled by the Actor. The Actor's aim is to maximise the expected return $J(\theta)$.

2. **Value Function**

$$\nabla_w J(w) \approx \frac{1}{N}\sum_{i=1}^{N} \nabla_w (V_w(s_i) - Q_w(s_i, a_i))^2$$

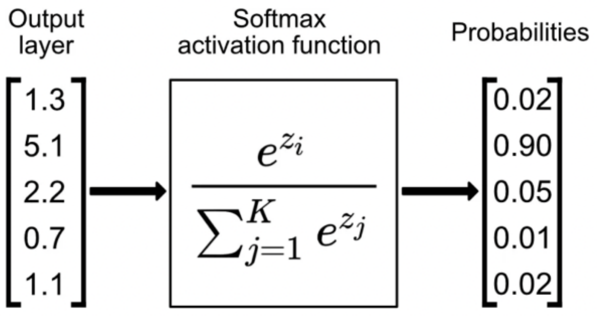
   a. $\nabla_w J(w)$ defines the gradient of the loss function with respect to w
   b. N defines the number of sampled experiences.
   c. $V_w(s_i)$ defines the estimate of the value of the state s
   d. $Q_w(s_i, a_i)$ defines the estimate of the value of taking the action a
   e. i defines the index of the sample.

   The value function in the context of the AC model is handled by the Critic. The goal of the Critic is to minimize the difference between the estimated value of $V_w(s_i)$ and the action value $Q_w(s_i, a_i)$.

The earlier mention policy function $\pi_\theta(a_i|s_i)$ is defined using the SoftMax function:

$$\pi_\theta(a_i|s_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

The SoftMax function takes raw output scores (logits) from the final layers of a neural network and converts them into probabilities.



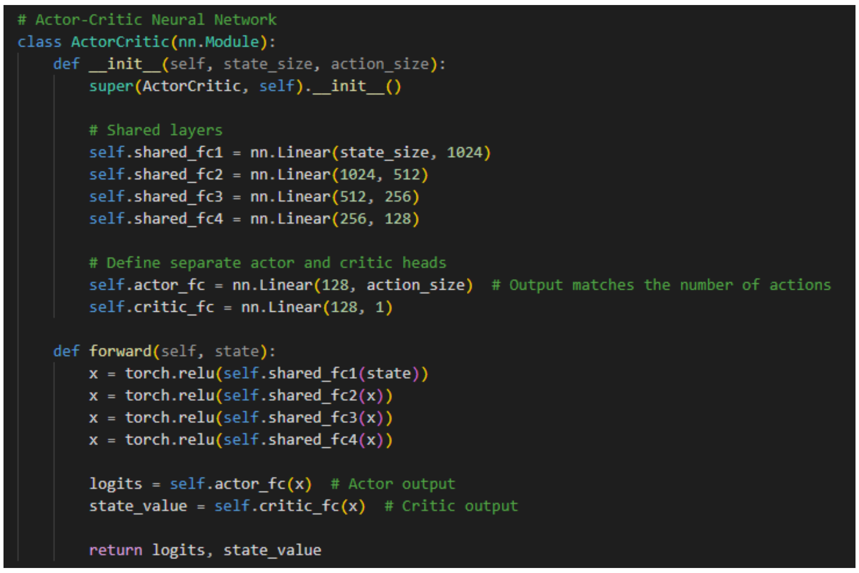
Output layer
Softmax activation function
Probabilities
1.3
5.1
2.2
0.7
1.1
$\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$
0.02
0.90
0.05
0.01
0.02

Figure 12 - Visual representation of SoftMax function

4.1.2 Model design

The Actor-Critic model is designed with shared fully connected layers followed by separate heads for the Actor and Critic. It takes the state size as input and outputs action probabilities and a state value. The shared layers consist of four linear transformations, progressively reducing the dimensionality from 1024 to 128 neurons. The Actor predicts action logits corresponding to the available actions, while the Critic outputs a scalar value representing the estimated value of the current state. This structure allows for simultaneous policy improvement and value estimation, facilitating effective learning in reinforcement tasks.

The code snippet below shows the implementation for the model:

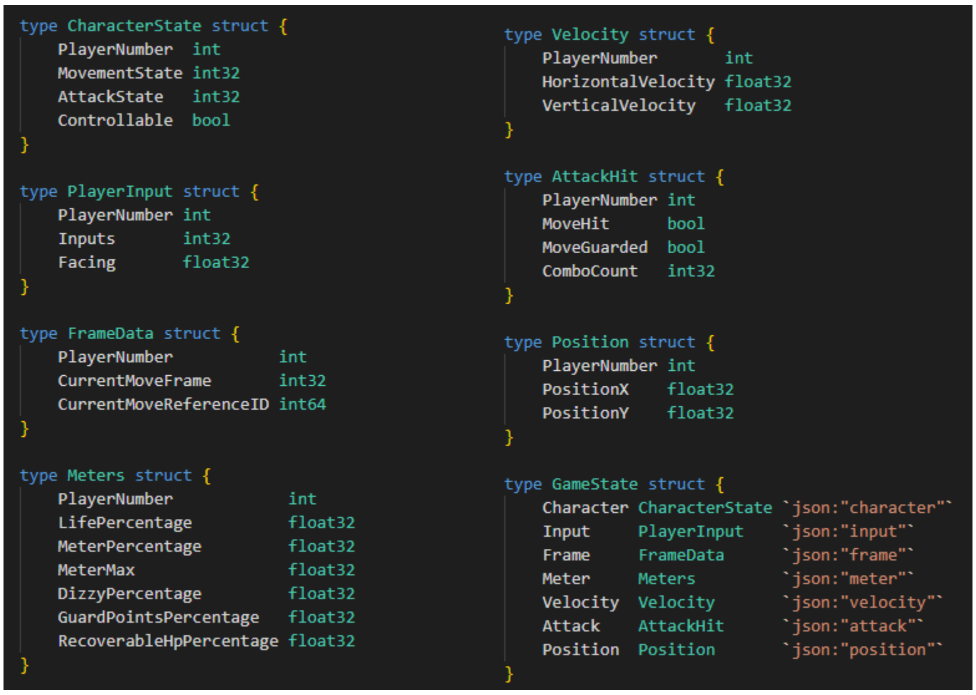


```python
# Actor-Critic Neural Network
class ActorCritic(nn.Module):
    def __init__(self, state_size, action_size):
        super(ActorCritic, self).__init__()

        # Shared layers
        self.shared_fc1 = nn.Linear(state_size, 1024)
        self.shared_fc2 = nn.Linear(1024, 512)
        self.shared_fc3 = nn.Linear(512, 256)
        self.shared_fc4 = nn.Linear(256, 128)

        # Define separate actor and critic heads
        self.actor_fc = nn.Linear(128, action_size)  # Output matches the number of actions
        self.critic_fc = nn.Linear(128, 1)

    def forward(self, state):
        x = torch.relu(self.shared_fc1(state))
        x = torch.relu(self.shared_fc2(x))
        x = torch.relu(self.shared_fc3(x))
        x = torch.relu(self.shared_fc4(x))

        logits = self.actor_fc(x)  # Actor output
        state_value = self.critic_fc(x)  # Critic output

        return logits, state_value
```

Figure 13 - Model design code snippet

4.2 Preprocessing steps

As with any AI system, before training, there is a need for preprocessing of the information that is going to be fed into the model. In this case, the game state extracted must be processed such that it is converted into a numerical representation of the model that is suitable for the neural network. The following code snippet shows the aspects of the game's state that were fed to the model:



```go
type CharacterState struct {
	PlayerNumber  int
	MovementState int32
	AttackState   int32
	Controllable  bool
}

type PlayerInput struct {
	PlayerNumber int
	Inputs       int32
	Facing       float32
}

type FrameData struct {
	PlayerNumber           int
	CurrentMoveFrame       int32
	CurrentMoveReferenceID int64
}

type Meters struct {
	PlayerNumber           int
	LifePercentage         float32
	MeterPercentage        float32
	MeterMax               float32
	DizzyPercentage        float32
	GuardPointsPercentage  float32
	RecoverableHpPercentage float32
}
```

```go
type Velocity struct {
	PlayerNumber       int
	HorizontalVelocity float32
	VerticalVelocity   float32
}

type AttackHit struct {
	PlayerNumber int
	MoveHit      bool
	MoveGuarded  bool
	ComboCount   int32
}

type Position struct {
	PlayerNumber int
	PositionX    float32
	PositionY    float32
}

type GameState struct {
	Character CharacterState `json:"character"`
	Input     PlayerInput    `json:"input"`
	Frame     FrameData      `json:"frame"`
	Meter     Meters         `json:"meter"`
	Velocity  Velocity       `json:"velocity"`
	Attack    AttackHit      `json:"attack"`
	Position  Position       `json:"position"`
}
```

Figure 14 - Game state code snippet

The preprocessing steps were as follows:

1. **Reading Shared Memory**: The game state is retrieved from shared memory to maintain synchronization with the game environment.
2. **Extracting Round Status**: The IsRoundOver status is checked to determine if the current round has concluded.
3. **Hashing Categorical Values**: Various game states (e.g., movement, attack, inputs) are hashed to convert them into fixed-size representations, allowing for consistent processing in neural networks. This also allows for unknown states to be handled more efficiently.
4. **Using Raw Values**: The player's facing direction, hit status, and guarded status are recorded for real-time decision-making.
5. **Normalizing Metrics**: Percentage values (e.g., life and meter) are used directly to ensure they are between 0 and 1.
6. **Velocity and Position Inclusion**: The player's current velocity and position data are included to provide spatial context to the model.

7. **Action History Handling**: The last few actions taken by the player are recorded to capture behavioural patterns. This history is trimmed to a fixed length to maintain consistency in input size.
8. **Creating State Vectors**: All processed data is concatenated into a single state vector for each player, representing their current state in the game.
9. **Returning State Information**: Finally, the function returns a NumPy array of state vectors and the round status, ready for use in the AC model.

After the data preprocessing, the state vector has a size of 118, with 59 unique values for each player.

4.3 Reward structure

The reward function evaluates the performance of the agent based on state changes during gameplay. It incorporates multiple factors to encourage desired behaviours:

1. **Combo Count**: Rewards are increased exponentially with the hit count of combos executed, promoting intelligent use of different attacking options.
2. **Health Management**: The agent gains rewards for damaging the opponent while being penalised, incentivizing effective offense and defence.
3. **Meter Management**: Rewards are given for increasing meter percentage, encouraging resource management. Meter is used to perform powered up special moves.
4. **Attacking Behaviour**: Successful hits yield a positive reward, especially when the opponent is vulnerable in the air.
5. **Guarding Behaviour**: Rewards for successful guarding and penalties for being guarded or broken, encouraging strategic defence.
6. **Proximity Management**: Rewards are adjusted based on the distance to the opponent, promoting movement strategies.
7. **Round Outcome**: Rewards are adjusted based on the round's outcome, encouraging the agent to play for the win.
8. **Time Consideration**: A penalty is applied based on elapsed time, discouraging passive play. This encourages dynamic gameplay.

This reward structure approach promotes balanced offensive and defensive strategies while adapting to changing game dynamics such as time and distance.

The table below shows the detailed reward structure for the agent:

Condition	**Reward**	**Remarks**
Combo count > 1.	+ min (2^combo_count, 10)	Combo rewards capped at 10
Opponent's health decreases.	+ (opponent_health_change * 10)	-
Agent's health decreases.	- (agent_health_change * 10)	-
Agent's meter increases.	+ (agent_meter_change * 10)	-
Agent lands a hit.	+ 5	-
Agent misses a hit	- 1	
Opponent lands a hit.	- 3	-
Agent guards an attack.	+ 2	Small reward for proper defensive play.
Opponent guards an attack.	- 1	Minor reward to encourage mix-ups in attacking play.
Agent's guard broken.	- 8	Heavy penalty to discourage getting stuck in block.
Distance between players > 90.	- 1	Specified without a hit to avoid double counting for hits and being within range.
Agent near screen boundary (x in [239, 271] or [-271, -239]).	- 2	Discourage getting stuck in the corner.
Opponent near screen boundary.	+ 2	Encourage cornering the opponent.
Agent wins round.	+ 10	-
Opponent wins round.	- 10	-
Elapsed time > 40s.	- (elapsed_time / 100)	Discourage passive play and stalling.
Action takes is the same as previous action	- (2 * number of times repeated)	Discourage spamming the same move.

Figure 15 - Reward structure for agent

Rewards for the agents were clamped between -10 and 10. This was done to ensure that the SoftMax function did not run into issues when dealing with large variance in the logits outputted by the model.

4.3.1 Handling logits

Due to the nature of the SoftMax function, it was common to run into issues where the action probabilities would be highly skewed towards certain actions to the point where there was a zero chance for most actions to occur in favour of one action. This could have been due to large variations in rewards, especially in extreme cases (e.g. using the same move 10 times in a row). As a result, several measures were implemented to reduce the variation within the logits:

1. **Bias addition**
   a. Bias addition applies a constant shift to specific logits to increase or decrease the final probabilities. In this case, all the logits were shifted by subtracting the mean of all the logits from each logit.
2. **Temperature scaling**
   a. Temperature scaling adjusts the sharpness of probabilities to favour higher logits or make logits more uniform (Davis et al., 2020). Essentially, it divides each logit z by some temperature T such that the exponential scaling of the numerator in the SoftMax function would be less extreme.

To evaluate the impact of logit adjustments on action selection, sample action probabilities were recorded both before and after the introduction of logit handling measures. These are a sample of the action probabilities after a single episode of training ***before*** the addition of the logit handling measures:

```
[9.9071421e-08 1.4658482e-06 3.8619010e-07 4.1452428e-09 1.0661253e-06
 2.9475539e-04 9.1549718e-01 9.0396171e-09 6.2331803e-02 4.4540706e-04
 3.9467950e-06 8.6898886e-04 2.1672859e-10 7.8041795e-09 1.2126155e-06
 9.8248798e-10 1.6645959e-14 1.1007945e-06 4.1609451e-06 3.7419985e-04
 8.1988425e-09 2.5183579e-06 1.6407425e-02 1.3306127e-03 3.3190431e-10
 1.2068915e-05 3.9322067e-08 3.5276972e-09 2.5387715e-05 2.5692350e-12
 1.1316604e-10 3.3636524e-11 1.9213738e-05 2.4298770e-13 6.4793210e-05
 4.8947819e-05 8.3409197e-04 2.3189256e-11 1.4291655e-03]
```

Figure 16 - Action probabilities before logit handling

The initial results demonstrate extreme disparity between actions. For example, the probability of selecting a single action was above **91.5%** (highlighted in green), while many other actions had **near-zero probabilities**, effectively making them irrelevant in the model's decision-making process. This kind of skewed behaviour is undesirable since it prevents exploration and limits the learning process.

This is a sample of the action probabilities ***after*** implementing logit handling measures:

```
[0.01454852 0.01540432 0.00801798 0.02050919 0.01464932 0.01226265
0.00358456 0.00073752 0.03877868 0.00252912 0.00790959 0.10962541
0.04863671 0.04432818 0.02625623 0.06211027 0.03869212 0.05942908
0.01104631 0.06963979 0.01388650 0.02143534 0.02176128 0.00514611
0.00607909 0.07916046 0.00717884 0.01795687 0.00703318 0.00123341
0.01251210 0.01837284 0.02551082 0.01555084 0.00178913 0.10604511
0.01198841 0.00305670 0.01560739]
```

Figure 17 - Action probabilities after logit handling

After implementing logit handling measures, the distribution of action probabilities became much more balanced. All actions are now considered to some extent, with the most probable action having a 10.96% chance and the least probable action still registering a small but non-negligible 0.07% chance.

Evidently, the logit handling was effective in reducing the disparity between the most likely option and least likely option. While this was the first episode of training and not necessarily an accurate representation of the final long-term action probabilities, it was far more promising to see that the agent was considering all actions to a certain degree instead of prioritising a single action. This is essential for continuous improvement for the model.

4.4 Training process

The training process for the AC model generally involves several key steps:

1. **Model Initialisation**
    a. Two independent agents are initialized, each running their own Actor-Critic models. The models are initialized with 1024–128 hidden units across shared layers, followed by separate actor and critic heads.
    b. CUDA support is leveraged if available to ensure faster computation with GPUs.
    c. An Action Space object is created to manage possible moves, translating them into keys and actions recognized by the game.
2. **State Sampling and Forward Pass**
    a. During every frame of the game, state data is captured via the shared memory space. The state is then pre-processed into a 118-dimensional vector.
    b. Each state vector contains character information (health, position, attack status) and the action history of both agents.
    c. The game state is passed as input to the Actor-Critic model to determine the next move.

3. **Action Execution**
    a. The Actor head of the model outputs logits, which are normalized into action probabilities using the SoftMax function.
    b. Temperature scaling controls the level of exploration: higher temperatures result in more uniform probabilities, while lower temperatures push the model toward high-value actions.
    c. Example: $\mathrm{temp} = \max(2.0, \mathrm{temp} * (0.99 ** \mathrm{episode}))$ gradually lowers the exploration as training progresses by decreasing the temperature over time.
    d. Actions are either randomly chosen (with probability epsilon) or sampled probabilistically based on the logits. This ensures a balance between exploration and exploitation. The probability epsilon also decays over time, moving towards a more exploitation focused training phase.
    e. The selected action is executed within the game, and key inputs are simulated.
    f. Each agent's recent actions are recorded to detect patterns.
    g. After actions are executed, rewards are calculated.
4. **Model Updates**
    a. The Critic head evaluates the state value, estimating future rewards.
    b. The Actor's loss encourages selecting high-reward actions, and the Critic's loss ensures value estimates are accurate over time
    c. Entropy regularization is used to encourage exploration by preventing the policy from becoming overly deterministic.
5. **Optimization and Backpropagation**
    a. The Adam optimizer is used with a small learning rate to update both Actor and Critic parameters
6. **Saving and Evaluation**
    a. The training loop runs for 500 episodes, with each episode lasting 100 seconds. After each episode, the game resets, and the training continues.
    b. The model is then tested by loading saved weights in an evaluation mode, either against a CPU or human player.

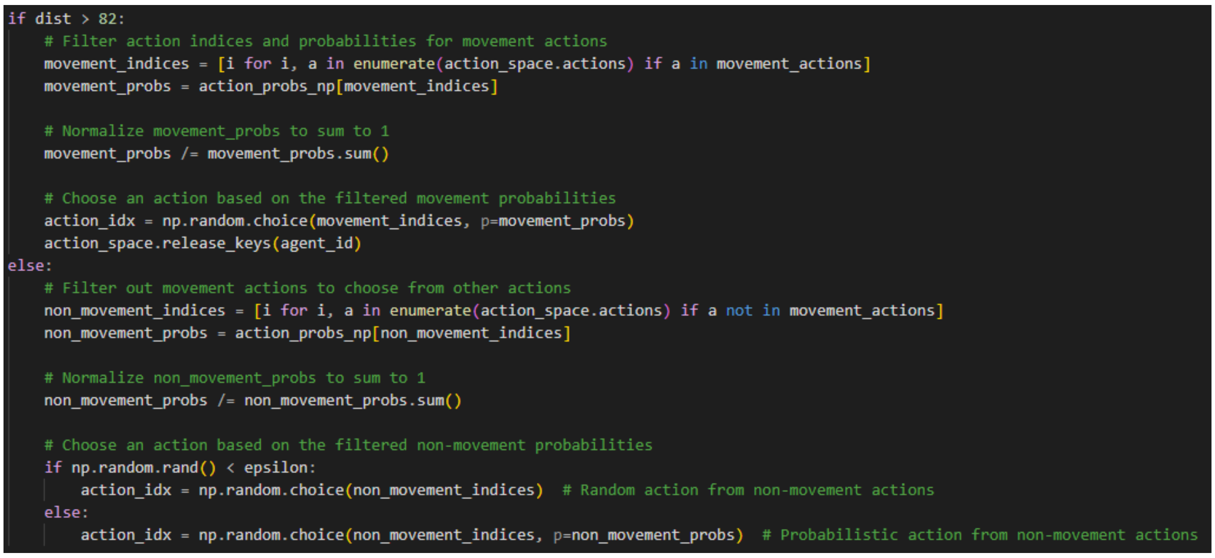
The overall flow of the model training remained relatively standard throughout the development process. Extra experiments were conducted to see how the final model would turn out, which were met with varying degrees of success. All models used a learning rate (gamma) of 0.99 to encourage long-term learning (Alto, 2019).

4.4.1 Experiment 1: Separating Movement Options

The first experiment involved isolating movement options such that the agent would only use those actions while they were outside of a certain distance threshold. Since KFM does not have any projectile-based attacks (e.g. Ryu's Hadoken), it is essential for KFM to stay within a certain range to utilise his kit effectively. As a result, robust movement policies are necessary to allow for dynamic gameplay.

It was also observed over earlier training rounds that the agents would frequently use certain special and super moves to get closer to the opponent rather than as combo tools of for actual offense. This was undesirable it showed that the agent was learning suboptimal policies, both in terms of movement and in terms of meter management.

To facilitate the isolation of the movement actions, the following code was implemented:



```
if dist > 82:
    # Filter action indices and probabilities for movement actions
    movement_indices = [i for i, a in enumerate(action_space.actions) if a in movement_actions]
    movement_probs = action_probs_np[movement_indices]

    # Normalize movement_probs to sum to 1
    movement_probs /= movement_probs.sum()

    # Choose an action based on the filtered movement probabilities
    action_idx = np.random.choice(movement_indices, p=movement_probs)
    action_space.release_keys(agent_id)
else:
    # Filter out movement actions to choose from other actions
    non_movement_indices = [i for i, a in enumerate(action_space.actions) if a not in movement_actions]
    non_movement_probs = action_probs_np[non_movement_indices]

    # Normalize non_movement_probs to sum to 1
    non_movement_probs /= non_movement_probs.sum()

    # Choose an action based on the filtered non-movement probabilities
    if np.random.rand() < epsilon:
        action_idx = np.random.choice(non_movement_indices)  # Random action from non-movement actions
    else:
        action_idx = np.random.choice(non_movement_indices, p=non_movement_probs)  # Probabilistic action from non-movement actions
```

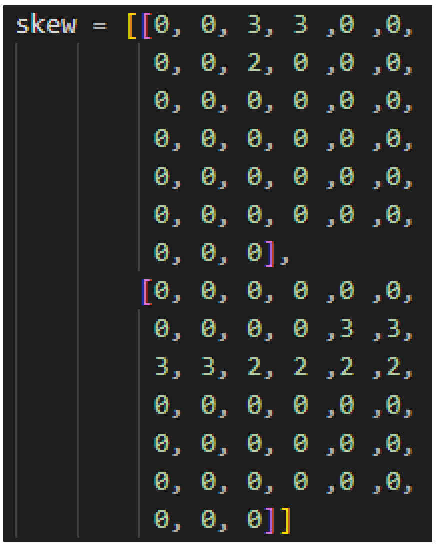
Figure 18 - Code snippet for movement action isolation

Basically, if the agents were not within 82 in-game distance units of each other, they would only choose movement actions with the intention of closing the gap. The distance was decided as a rough estimate of the natural position that a competent player would be comfortable playing in such that the player could avoid opponents moves and get hits in with minimal forward movement.

4.4.2 Experiment 2: Skewing Logits

The second experiment conducted was on introducing habits for each agent by skewing the logits and therefore increasing the probability that certain actions would be taken. In this case, one agent could be introduced with a bias such that they prefer to approach with aerial options, while the other agent prefers to use grounded attack options. The goal was to introduce asymmetry (as mentioned earlier) to observe how the model performs. This was also intended to simulate more human-like behaviour, as every many human players tend to develop habits during their gameplay.

To facilitate this, first, a skew matrix is defined:

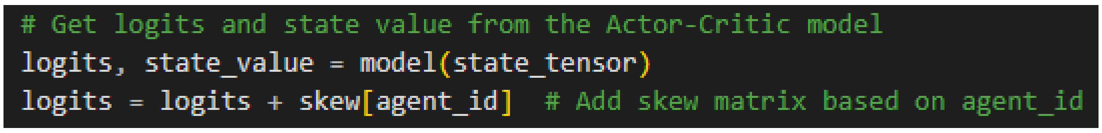


```
skew = [[0, 0, 3, 3 ,0 ,0,
         0, 0, 2, 0 ,0 ,0,
         0, 0, 0, 0 ,0 ,0,
         0, 0, 0, 0 ,0 ,0,
         0, 0, 0, 0 ,0 ,0,
         0, 0, 0, 0 ,0 ,0,
         0, 0, 0],
        [0, 0, 0, 0 ,0 ,0,
         0, 0, 0, 0 ,3 ,3,
         3, 3, 2, 2 ,2 ,2,
         0, 0, 0, 0 ,0 ,0,
         0, 0, 0, 0 ,0 ,0,
         0, 0, 0, 0 ,0 ,0,
         0, 0, 0]]
```

Figure 19 - Code snippet showing skew matrix

The skew matrix allows for the choosing of specific actions should be encourage and which should be discouraged. In this case, the skew matrix encourages jumping actions for the first agent and grounded attack options from the second agent.

Next, the skew matrix is added to the raw logits obtained from the model:



```
# Get logits and state value from the Actor-Critic model
logits, state_value = model(state_tensor)
logits = logits + skew[agent_id]  # Add skew matrix based on agent_id
```

Figure 20 - Code snippet for skewing implementation

This will eventually artificially bias the probabilities towards certain actions, thus introducing "habits".

4.5 Results

This section goes over the overall performance of the base model and experiments. The models are evaluated mainly on:

1. **Total rewards per episode.**
    a. Shows trend in rewards as training progresses. Ideally, rewards will become more positive as time goes on
2. **Mean, max and min total rewards over episodes.**
    a. General measures of how agent performed in general
3. **Observed performance (number of wins) against the in-built CPU.**
    a. Practical application model
4. **Observed performance against human controlled players.**
    a. Practical application model
5. **Subjective remarks from the human players.**
    a. Overall analysis of human-like behaviour modelling

It is also noted that the mean, max and min rewards tended to have negative values as the rewards structure was more biased towards giving penalties.

4.5.1 Base model Results

The system and performance analysis were first started by analysing a graph of total rewards per episode against episodes.

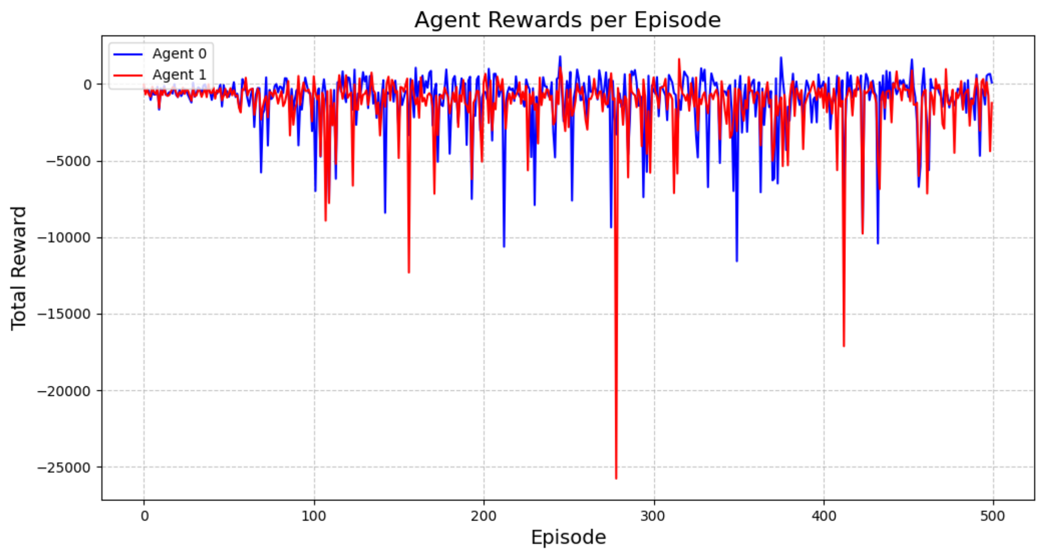
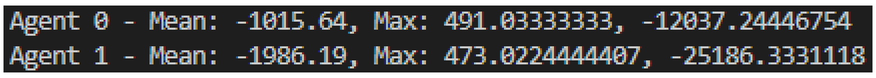


Figure 21 - Rewards per episode for base model

The performance of the base model was mostly acceptable, offering similar fast paced and dynamic gameplay between the agents. However, as seen by the very low lows of each agent, it was clear that there were some underlying issues which needed to be addressed. Observations included frequent "stalling" games where agents would jump around and attack, but nowhere near enough to each other to successfully land hits.

The screenshot below shows the mean, max and min rewards per episode for the agents:



```
Agent 0 - Mean: -1015.64, Max: 491.03333333, -12037.24446754
Agent 1 - Mean: -1986.19, Max: 473.0224444407, -25186.3331118
```

Figure 22 - Overall statistics for base model

Again, the overall low scores could be attributed to the occasional lack of interaction between agents, causing them to accumulate negative rewards quickly (missing attacks, not keeping within a good distance etc.).

The table below summarises each agent's performance against the CPU opponent and human opponent in 50 rounds:

Agent	Opponent	Rounds	Won	Lost
0	CPU	50	18	32
1	CPU	50	12	38
0	Human	50	5	45
1	Human	50	3	47

Figure 23 – Base model agents' performance against CPU and human opponents

Against the CPU, the agents performed reasonably well even with the occasional irrational behaviour. This could be attributed to the aggressive nature of KFM's CPU scripts as well, where the CPU tended to engage mode frequently than disengage.

Evidently, the agents produced by the base model did not function very well against human players, likely attributed to their tendency to start stalling in many situations. Human players could simply get a health advantage and then allow the agent to stall itself out to win a round.

Overall, the agents performed decently against CPU opponents but remained exploitable by adaptable human opponents. The agents did not show a high degree of human-like behaviour, prompting further tweaks and thus inspiring the experiments.

4.5.2 Experiment 1: Separating Movement Options Results

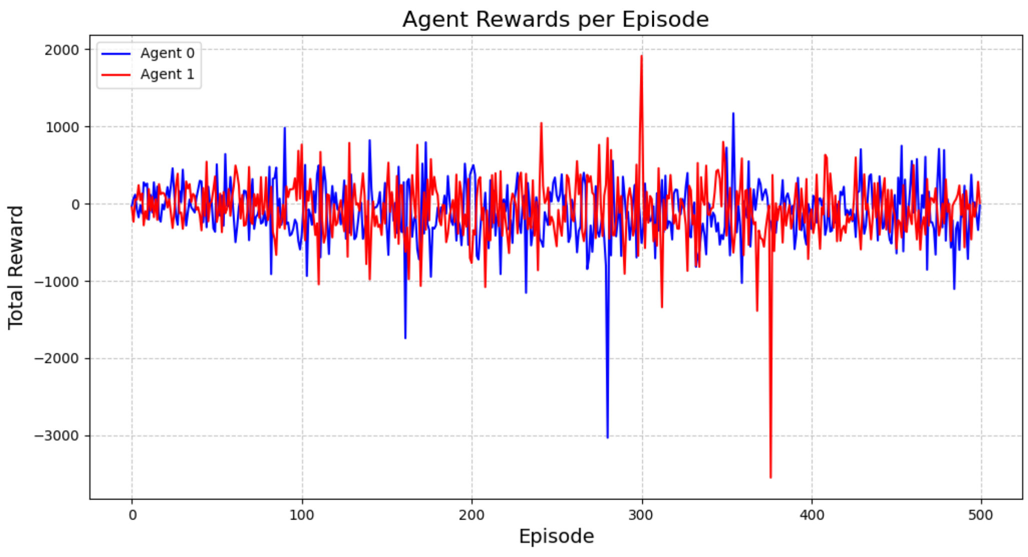


Figure 24 - Rewards per episode for Experiment 1

Like the base model, both agents performed similarly over 500 episodes of training, likely due to the symmetry in their training environment. The isolation of movement actions was helpful overall to drastically reduce unnecessary usage of moves for the agents which originally caused them to idle for long periods. This promoted faster and more aggressive playstyles from both agents. However, it was observed that the agents tended to have subpar defensive play.

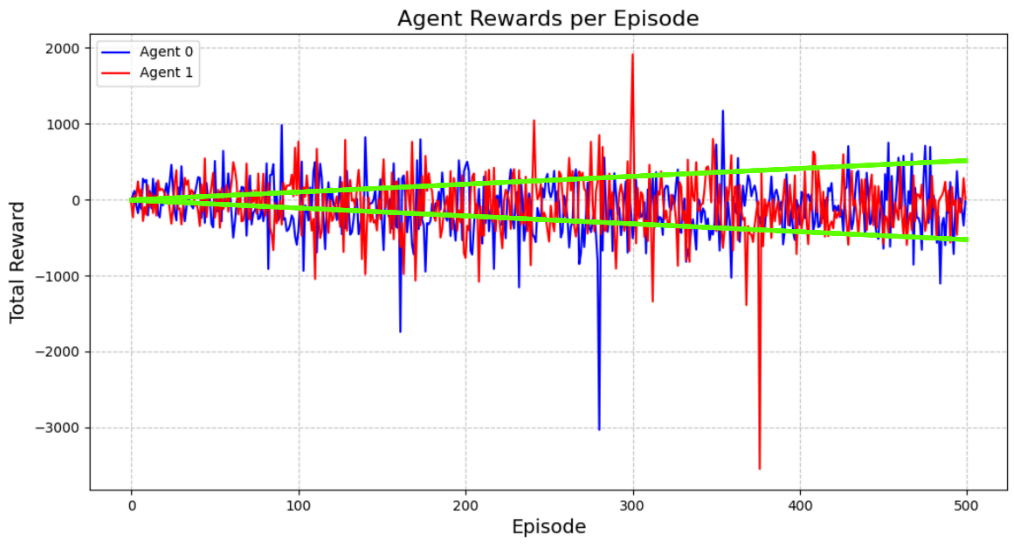
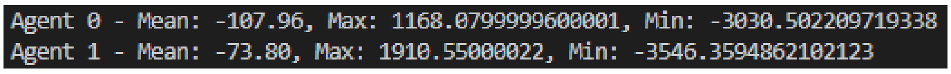


Figure 25 - Overall trend for Experiment 1

By plotting a trend line over the original data, we can see that both agents generally have increasing rewards over episodes as they began to perform better against the opponent (and corresponding higher penalties). This was promising as it confirmed that the agents were able to learn strategies to counteract the opponent's actions.

The screenshot below shows the mean, max and min rewards per episode for the agents:



```
Agent 0 - Mean: -107.96, Max: 1168.0799999600001, Min: -3030.502209719338
Agent 1 - Mean: -73.80, Max: 1910.55000022, Min: -3546.3594862102123
```

Figure 26 - Overall statistics for Experiment 1

While the average reward values are negative, they are not drastically far below zero. This suggests that the agents are making meaningful attempts at learning but may require further tuning in hyperparameters, reward functions, or training time to improve consistency. Additionally, given the nature of the negative mean, it indicates that the agents are occasionally making mistakes but are not perpetually stuck in poor states.

The maximum reward values suggest that both agents demonstrated exceptional performance in some episodes. A reward above 1000 implies that the agents were capable of executing effective strategies (such as landing high-damage combos or good defensive play and positioning) during certain rounds. This means that the current model is capable of high-level decision-making, even though it might not be consistent across all episodes.

The table below summarises each agent's performance against the CPU opponent and human opponent in 50 rounds:

Agent	Opponent	Rounds	Won	Lost
0	CPU	50	27	23
1	CPU	50	29	21
0	Human	50	16	34
1	Human	50	20	30

Figure 27 - Experiment 1 agents' performance against CPU and human opponents

While not perfect, the agents show better performance as compared to the base model. More aggressive play was shown, though the CPU opponent had the edge in overall damage defensive play. The human player also noticed that the agents tended to have easily exploitable defensive patterns. Overall, the model performed better than the base model in terms of aggressive play, but ultimately failed to capture the balance in defensive play.

4.5.3 Experiment 2: Skewing Logits

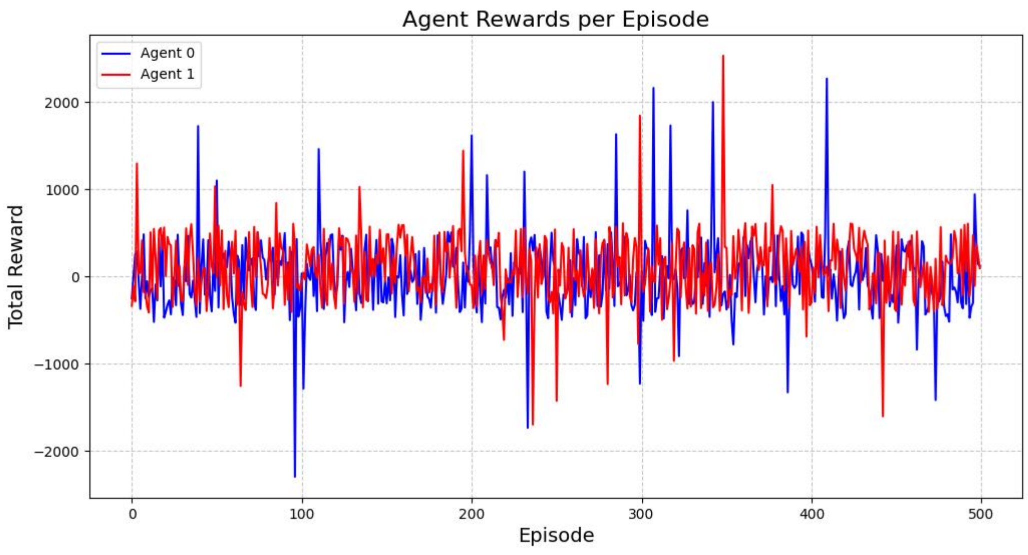
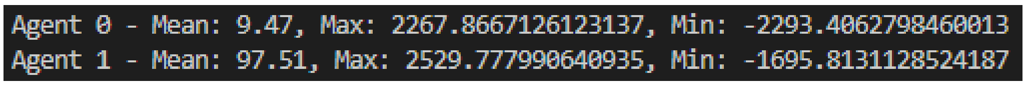


Figure 28 - Rewards per episode for Experiment 2

Overall, the agents in Experiment 2 performed very consistently over the course of training, displaying adherence to their given strategies. It was observed that the agents would sometimes tend to lean into their strategies too much to the point where they would only perform the skewed actions, leading to the observed peaks and dips in the graph. However, after a few rounds of training, this would usually wear off, allowing for dynamic fights to continue.

The screenshot below shows the mean, max and min rewards per episode for the agents:



```
Agent 0 - Mean: 9.47, Max: 2267.8667126123137, Min: -2293.4062798460013
Agent 1 - Mean: 97.51, Max: 2529.777990640935, Min: -1695.8131128524187
```

Figure 29 - Overall statistics for Experiment 2

Overall, Agent 1 tended to perform better. This was expected, as generally in fighting games, using the same approach tools (such as jumping in this case) is easily recognisable and punishable. In the case of Agent 1, the skew encourages it to use more moves, therefore encouraging more mix-ups in its playstyle, making it harder to adapt to.

The table below summarises each agent's performance against the CPU opponent and human opponent in 50 rounds:

Agent	Opponent	Rounds	Won	Lost
0	CPU	50	21	29
1	CPU	50	29	21
0	Human	50	17	33
1	Human	50	25	25

Figure 30 - Experiment 2 agents' performance against CPU and Human opponents

Against the in-built CPU, the agent performed well, with Agent 0 winning almost half its games and Agent 1 coming out on top in its 50 bouts. This success can likely be attributed to Agent 1's bias towards a broader set of actions, enabling more dynamic and varied gameplay. By having access to a wider range of strategic options, Agent 1 was better equipped to adapt to the CPU's behaviour.

In contrast, Agent 0 struggled in matches against human players. It became evident that its primary strategy—favouring jumps—was easily countered. Jumping introduces a longer frame commitment than walking, making it easier for opponents to predict and respond. Furthermore, frequent jumps limited Agent 0’s defensive capabilities, leaving it vulnerable to attacks. On the other hand, Agent 1 fared much better, holding its own against human players and achieving an even split in matches. Its success was attributed not only to its diverse gameplay but also to the agent's rapid input speed, which made it challenging for opponents to react quickly.

Overall, the agents in Experiment 2 outperformed their previous iterations, indicating significant progress. While Agent 1’s performance was especially noteworthy, the experiment highlighted the importance of balancing input speed, action variety, and strategic flexibility in building effective agents. These findings point towards promising avenues for further refinement and optimization.

5 Limitations and Next Steps

5.1 Limitations

While the implementation of AI was successful to an extent, it remains that there are certain issues that could not be overcome due to either time constraints or computational power:

5.1.1 Real-Time Processing

Implementing an AI that can operate in real-time within the constraints of a fast-paced fighting game requires careful optimization of both the AI model and the data exchange mechanisms. Furthermore, the model needs to be able to process inputs fast enough for the agent to receive instructions to react to the given state. The IkemenGO engine runs at approximate 60 frames per second (FPS), leaving very little time for the model to do processing in a frame-by-frame window. While GPU acceleration did greatly alleviate this issue, it was not to the point where frame by frame processing was possible. As a result, there was always some form disparity in the recording of the immediate state after the agent took an action.

5.1.2 Reward misattribution

As with any fighting game, each character has its own specific frame data. Frame data the punch may not land immediately but will only activate after several frames. For example, when KFM's Light Punch input is asserted, the actual attack may come out a few frames after the input is completed. As a result, there was some difficulty in determining when the effects of an input should end and when the next input should start. For example, the input for a super attack (which usually takes more frames before the actual hit comes out) may be sent to the agent followed by a jump. In the next state retrieved by the agent after executing the special move, the in-game KFM may be in the process of executing the move, but the agent has already assigned rewards to the jump instead of the super attack to based off the newly retrieved state, resulting in reward misattribution. If the special move connects, jump action may receive rewards it wasn't supposed to receive, thereby unintentionally increasing the probability of the agent jumping.

This frame-based complexity presents a challenge for the AI, as it needs to synchronize input-output timing with the game's internal mechanics. Without precise handling of this, the agent's learning performance suffers, and it struggles to develop effective strategies, especially for longer and more complex attack sequences.

5.1.3 Limited Generalisability

One major limitation of training the AI only using KFM is that the behaviours learned may not easily generalize to other characters with different attributes, moves, and frame data. There are several reasons why this may occur:

1. **Character-Specific Frame Data and Move Properties:**

- Each character in a fighting game has unique frame data. Since the AI only learns KFM’s frame data, it might struggle with other characters whose moves are slower, faster, or have longer recovery windows.
- For example, KFM’s fast, chainable attacks may teach the AI to adopt aggressive playstyles, but this strategy might not be viable for characters with slower or more defensive moves.

2. **Varying Ranges and Hitboxes:**
   - Different characters often have varying attack ranges and hitbox placements, which directly affect spacing, zoning, and footwork. An AI optimized for KFM might position itself too close or far when playing other characters, leading to suboptimal spacing or missed attacks. Furthermore, other characters often employ the use of projectile attacks, which KFM simply does not have.
3. **Movement Differences Across Characters:**
   - Characters may differ in mobility, such as dash speed, jump height, or air options. For example, an AI that learns to rely on KFM’s movement options may struggle with other characters who need to rely on different movement tactics. This reduces the AI's ability to transfer learned behaviours effectively.
4. **Inconsistent Combo Opportunities:**
   - Each character in a fighting game often has unique combo routes and optimal move sequences. Furthermore, different characters often have different inputs. For example, KFM only employs quarter circle and DP inputs, but other special inputs such as full circle turns and half circle turns are also present in fighting games. If the AI only practices KFM’s combos, it may fail to recognize or adapt to other characters' combo trees, resulting in ineffective or poorly executed attacks when playing as or against other characters.
5. **Adaptation to Different Matchups:**
   - Fighting games involve matchup-specific strategies—for example, different counters, anti-airs, or approaches are required depending on the opponent. An AI trained only on KFM may be unprepared to handle unique challenges posed by specific characters with unconventional tools, such as grapplers or zoners.

5.1.4 Personal Observations

There were certain inconsistencies in the training process that were strange in nature. Sometimes, during training, one of the agents would completely dominate a round, demonstrating a solid understanding of the game state and the appropriate actions to take. However, in the following round, the same agent would exhibit seemingly irrational behaviour—such as repeatedly jumping or performing non-optimal actions without any apparent reason—leading to poor performance.

5.2 Next Steps

Building on the current implementation, there are several areas for future improvements to overcome the current limitations of the model.

5.2.1 Expand Character Roster

To address the limited scope of the current model, future iterations could focus on training multiple characters with varying move sets and unique mechanics. Additionally, training could be conducted differently such that each round is played by the same agents but on different character each time. This would lead to the AI inevitably being exposed to different playstyles and matchups. This would help the AI develop more general gameplay strategies. Another way to apply this would be to have agents dedicated to a specific character to learn character specific strategies.

5.2.2 Model Compression

Further optimizing the AI to handle real-time processing would make it more competitive in fast-paced environments. Techniques like model pruning and quantization (Malingan, 2023) can be applied to reduce the size and complexity of the neural network, enabling faster decision-making. It might also be worth experimenting with temporal abstraction, where the AI evaluates actions over several frames instead of frame-by-frame, reducing the computational burden.

1. **Pruning**
    a. Model pruning is a technique that removes unimportant or redundant model parameters. This can reduce the complexity and resource requirements of the model while still maintaining good performance.
2. **Quantisation**
    a. Model quantization is a technique that reduces the precision of the model's parameters, usually from a 32-bit floating point to 8-bit integers. This can significantly reduce the memory and computational requirements of the model while still maintaining good performance.

5.2.3 Improve Reward Attribution

Modifications to the reward function and/or IkemenGO could be explored to address misattribution issues. This could involve assigning delayed rewards—where the AI waits for confirmation of an attack landing—rather than immediately rewarding the action.

Another approach is credit assignment mechanisms, such as using TD(λ) or eligibility traces, which would help the AI correctly assign rewards to earlier actions in a sequence (Datta, 2024).

5.2.4 Integration of Imitation Learning

As previously discussed, imitation learning is an effective way to teach AI to perform complex actions. Future improvements could explore human-in-the-loop training, where human players guide the agent's learning process through demonstrations or feedback. This would help the AI learn complex manoeuvres that would otherwise take a long time to master autonomously. Behaviour cloning techniques could also be used to teach the AI from existing gameplay data.

6 Conclusion

The project demonstrated promising advancements toward developing AI agents capable of engaging in complex fighting game scenarios with adaptive, human-like behaviour. The tests demonstrated the benefits of the actor-critic model, since the agents could apply strategic action selection, learn from past mistakes, and optimize through skewed logits to produce a wider range of playstyles.

Subpar defence and stalling behaviour were among the areas of weakness identified by the base model; nevertheless, the follow-up trials demonstrated notable improvements. While skewing logits created asymmetry and encouraged faster, more aggressive games, splitting movement actions encouraged better adaptability and more interesting matches. The agents' victory over the AI opponent with built-in CPU demonstrates that reinforcement learning is a viable approach for gaming AI. But their inconsistent performance against human players brought to light important issues, such as the difficulty of reproducing subtle decision-based counterplay

Despite progress, a number of limitations were discovered, such as rewards misattribution, processing limitations in real-time, and limited generalizability as a result of training with a single character (KFM). These difficulties show how difficult it is to create competitive AI in dynamic settings such as fighting games.

Future iterations will need to address these limitations through broadened character training, model compression, and improved reward functions. The introduction of imitation learning—where agents learn from human demonstrations—could also improve their capabilities, allowing for more complex gameplay while decreasing the time needed for training.

In conclusion, the study successfully established a framework for additional research into fighting game adaptive AI. With refinements in training methodology, reward attribution, and real-time performance, these agents could evolve into formidable opponents, providing important insights into the intersection of AI and gaming. The progress made offers an intriguing glimpse into how AI can improve both gameplay and player engagement in future fighting games.

References

Alto, V. (2019, July 6). Neural networks: Parameters, hyperparameters and optimization strategies. Medium. https://towardsdatascience.com/neural-networks-parameters-hyperparameters-and-optimization-strategies-3f0842fac0a5

Datta, S. (2024, March 18). What is the credit assignment problem?. Baeldung on Computer Science. https://www.baeldung.com/cs/credit-assignment-problem

Davis, J., Zhu, J., Oldfather, J., McDonald, S., Trzaskowski, M., & Kelsen, M. (2020, August). Temperature scaling - AWS prescriptive guidance. Amazon Web Services. https://docs.aws.amazon.com/prescriptive-guidance/latest/ml-quantifying-uncertainty/temp-scaling.html

Famutimi, F. (2023, December 22). From niche to thriving: The Fighting Game Renaissance - Esports Insider. Esports Insider. https://esportsinsider.com/2023/12/from-niche-to-thriving-the-fighting-game-renaissance#:~:text=In%20a%20broad%20sense%2C%20fighting,firsts'%20in%20the%20gaming%20scene.

Griebsch, L. (2022). Player Imitation | How to make your players redundant [White Paper]. Hochschule Der Medien. https://ai.hdm-stuttgart.de/downloads/student-white-paper/Sommer-2022/Player_Imitation.pdf

K4Thos. (2023, February 28). IkemenGO: An open-source fighting game engine that supports Mugen Resources. GitHub. https://github.com/ikemen-engine/Ikemen-GO

Malingan, N. (2023, May 11). Quantization and pruning. Scaler. https://www.scaler.com/topics/quantization-and-pruning/

Martinez-Arellano, G., Cant, R., & Woods, D. (2017). Creating AI characters for fighting games using genetic programming. IEEE Transactions on Computational Intelligence and AI in Games, 9(4), 423–434. https://doi.org/10.1109/tciaig.2016.2642158

Mendonca, M. R., Bernardino, H. S., & Neto, R. F. (2015). Simulating human behavior in fighting games using reinforcement learning and Artificial Neural Networks. 2015 14th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames). https://doi.org/10.1109/sbgames.2015.25

Most popular esports games 2023 | esports charts. ESPORTS CHARTS. (2024). https://escharts.com/top-games?order=peak

Pittman, J. (2015, August 11). The Pac-Man Dossier | Chapter 4. The Pac-Man Dossier. https://pacman.holenet.info/#Chapter_4

Sutton, R. S., Bach, F., & Barto, A. G. (2018). Reinforcement learning: An introduction. MIT Press Ltd.

Thompson, T. (2022, May 2). The Ai of Doom (1993). The AI of DOOM (1993). https://www.gamedeveloper.com/game-platforms/the-ai-of-doom-1993

Yamamoto, K., Mizuno, S., Chun Yin Chu, & Thawonmas, R. (2014). Deduction of fighting-game countermeasures using the K-nearest neighbor algorithm and a game simulator. 2014 IEEE Conference on Computational Intelligence and Games, 1–5. https://doi.org/10.1109/cig.2014.6932915