Applied Artificial Intelligence | AI-4007 | Project

MAY 10, 2023

Abdullah Malik : 20i-0930
Daniyal Khan : 20i-1847

Deliverable:02 | Applied AI

Machine Learning

Contents

Machine learning is a subfield of artificial intelligence (AI) that focuses on developing algorithms and models that enable computers to learn from and make predictions or decisions based on data, without being explicitly programmed.

Machine learning has significantly improved AI by providing the ability to learn from large amounts of data and make intelligent decisions or predictions. It enables AI systems to adapt and improve their performance over time through iterative learning processes.

In games, machine learning is used to enhance various aspects, including:

1. Game AI: Machine learning techniques can be applied to train AI agents that exhibit more realistic and challenging behaviors, providing a more engaging and immersive gaming experience.
2. Player Modeling: Machine learning models can analyze player data to understand player preferences, behaviors, and skill levels. This information can be used to personalize the game experience, dynamically adjust difficulty levels, or create tailored content for individual players.
3. Procedural Content Generation: Machine learning algorithms can generate game content such as levels, maps, characters, and items based on learned patterns and player preferences, leading to more diverse and dynamically generated game environments.
4. Game Testing and Quality Assurance: Machine learning can be employed to automate game testing processes, detect bugs or anomalies, and optimize game performance, ultimately improving the overall quality and stability of the game.

Overall, machine learning in games contributes to more intelligent and adaptive gameplay, enhanced player experiences, and efficient game development processes.

Abstract/Executive Summary of the project:

Summary:

The GameBot project aims to develop a Python-based game automation system using the BizHawk emulator and machine learning techniques. The project involves running a game, Street Fighter II Turbo, and implementing a fight function to control game characters based on the input from the game. The project provides an API with classes for managing game state, player actions, and communication with the emulator. Players can generate datasets by playing single-player mode, recording percepts, and actions for training ML/DL algorithms. The solution requires a generic approach, excluding reinforcement learning and rule-based agents. By leveraging machine learning and game automation, the GameBot project aims to enhance the game AI, player experiences, and content generation in video games.

Objectives:

Game automation, AI in games, ML in games, dataset generation, AI integration, Enhanced player experience etc.

Key-findings

I don't have that specific key findings, but I did get to know game automation feasibility, AI agent performance, Dataset Analysis, Impact of Game AI, generalizability of the solution.

Introduction:

Introduction:

The GameBot project is a Python-based endeavor that explores game automation and machine learning techniques. With the objective of creating an intelligent game-playing system, the project utilizes the BizHawk emulator to run the game Street Fighter II Turbo. By implementing a fight function within the game, players can control characters based on real-time input. The project provides an API with classes for managing game state, player actions, and communication with the emulator. Additionally, players can generate datasets by recording percepts and actions during single-player mode, enabling the training of machine learning and deep learning algorithms. It is essential to develop a generic solution, excluding reinforcement learning and rule-based agents. By combining machine learning and game automation, the

GameBot project aims to enhance game AI, improve player experiences, and facilitate dynamic content generation within video games.

Motivation:

Several factors, including machine learning, AI driven Apps, and how they operate. In short, here are the following motivational factors:

1. Exploration of Game Automation
2. Advancing AI in Gaming
3. Personal Skill Development
4. Dataset Generation for ML/DL
5. Contribution to Game Development

Problem Statement:

The problem statement in the provided instructions is to run a game and execute the provided API code to control the game using a bot. The goal is to develop a bot that can play Street Fighter II Turbo and interact with the game server using the API.

The instructions provide steps to set up and run the game using an emulator (EmuHawk.exe) and load the game ROM file. Then, the API code needs to be executed in the command prompt with a command-line argument (1 or 2) to specify which player the bot will control. The API code establishes a connection with the game server and allows the bot to receive the game state, make decisions, and send commands to the game server.

If running a single bot, the bot will play against the CPU. If running two bots simultaneously, they will fight against each other.

The problem statement implies that the goal is to develop and test a bot that can effectively play Street Fighter II Turbo by leveraging the provided API and game server. The specific requirements or objectives of the bot's behavior or performance are not explicitly mentioned in the instructions.

Goals aimed to achieve:

I plan to achieve the same goals as mentioned in the motivations.

Prerequisites/Dependencies:

The following prerequisites and dependencies for the GameBot project were mentioned by ma'am:

Operating System:

The project requires Windows 7 or above (64-bit) as the operating system.

Python:

The Python API provided in the project is developed and tested on version 3.6.3. It is recommended to use this version or any version above 3. Some slight modifications may be needed to run the code on Python versions below 3.

BizHawk Emulator Pre-requisites:

I had to Install the necessary pre-requisites for the BizHawk emulator, which are provided in the attached zip file. These pre-requisites included libraries, modules, and dependencies required for running the emulator smoothly.

Installation and Setup:

Here are the instructions provided to us:

1. Install the dependencies/libraries using "pip install package_name"
2. e.g: "pip install pandas, sklearn, numpy"
3. After successful installation goto cmd on the file location(The python api folder)
4. Now run the command "python controller.py 1 "
5. Now Goto single player folder and open "EmuHawk"
6. Now in EmuHawk use "ctrl+o" shortcut key to insert file
7. Now add the file "Street fighter II Turbo (U).smc" to execute the file
8. Now game will start running
9. Use enter to continue untill the game starts
10. Now to use bot click on "tools" open on top row
11. Now select "open tools"
12. Now select 2nd option bot
13. Now the move will be recorded in the "dataset.csv"

Project Structure and Files:

Describe the structure of your project and the purpose of each file or directory. This section can include a brief overview of the main files, their functionalities, and how they interact with each other.

Explanation of Files:

Bot.py

Controller.py

The code begins by importing the required modules: socket, json, game_state, pandas, numpy, sys, and bot. These modules will be used for network communication, data serialization/deserialization, game state representation, data manipulation, and bot logic.

The connect function is defined. It takes a port parameter and creates a socket connection to the game server. It binds the socket to the local address (127.0.0.1) and the specified port. Then, it listens for incoming connections and accepts one when it arrives. Finally, it returns the client socket that is used for communication with the game server.

The send function is defined. It takes the client_socket and a command parameter. The command object is converted to a dictionary using its object_to_dict method and then serialized to JSON. The JSON payload is sent through the client_socket to the game server.

The receive function is defined. It takes the client_socket as a parameter. It receives data from the client_socket, decodes it from JSON format to a dictionary, and creates a GameState object using the GameState class from the game_state module. The GameState object represents the current state of the game and is returned.

The main function is defined as the entry point of the script. It first checks the value of sys.argv[1] to determine the connection port for the game server. It calls the connect function to establish a connection with the game server and assigns the returned client socket to client_socket.

Within the main function, a while loop is used to continuously interact with the game server until the game round is over. The loop performs the following steps:

a. It receives the current game state by calling the receive function and assigns it to current_game_state.

b. It invokes the fight method of the bot object, passing the current_game_state and sys.argv[1] as parameters. This method is expected to return a command for the bot to execute.

c. Various attributes of the current_game_state object are extracted and assigned to variables for further processing.

d. A new row is created as a dictionary, containing the extracted attribute values.

e. The new row is printed.

f. The new row is appended to the botdf DataFrame using the _append method.

g. The botdf DataFrame is saved to a CSV file named "dataset.csv" with semicolon delimiter, UTF-8 encoding, no index column, and a header row.

h. The bot command is sent to the game server by calling the send function with the client_socket and bot_command as parameters.

Finally, the main function is called if the script is executed directly (i.e., not imported as a module).

Command.py

The Command class represents a command to be sent to the game. It has the following attributes:

player_buttons: An instance of the Buttons class representing the buttons pressed by player 1.
player2_buttons: An instance of the Buttons class representing the buttons pressed by player 2.
type: A string indicating the type of the command (in this case, "buttons").
__player_count: An integer indicating the number of players (set to 2).
save_game_path: A string representing the path to save the game (initially empty).

The Command class also has a method called object_to_dict which converts the Command object to a dictionary representation. It creates a dictionary and adds the attributes of the Command object to the dictionary. It also calls the object_to_dict method of the Buttons objects to convert them to dictionaries and add them to the command dictionary. Finally, it returns the command dictionary.

Buttons.py:

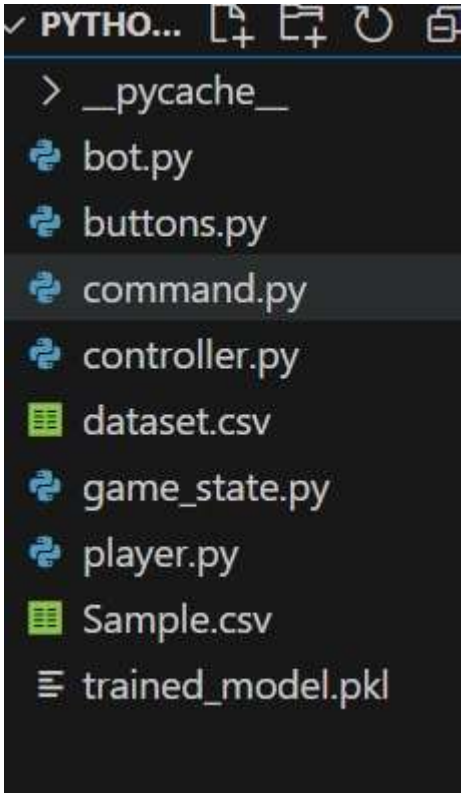
The Buttons class represents the state of various buttons on a game controller. It has the following attributes representing different buttons:

up: A boolean indicating whether the up button is pressed or not.
down: A boolean indicating whether the down button is pressed or not.
right: A boolean indicating whether the right button is pressed or not.
left: A boolean indicating whether the left button is pressed or not.
select: A boolean indicating whether the select button is pressed or not.
start: A boolean indicating whether the start button is pressed or not.
Y, B, X, A, L, R: Booleans indicating whether corresponding buttons (Y, B, X, A, L, R) are pressed or not.
The class provides the following methods:

init(self, buttons_dict=None): The initializer method that creates a Buttons object. If a buttons_dict parameter is provided, it calls the dict_to_object method to initialize the object from the dictionary. Otherwise, it calls the init_buttons method to set the default button states.
init_buttons(self): Sets the button states to their default values.
dict_to_object(self, buttons_dict): Updates the button states of the object from a dictionary representation provided as buttons_dict.
The object_to_dict method converts the button states of the object into a dictionary representation. It creates an empty dictionary buttons_dict and assigns the values of the button states as key-value pairs in the dictionary.

On the other hand, the Buttons class also supports the conversion from a dictionary representation to the button states of the object through the dict_to_object method. This method takes a buttons_dict parameter, which is a dictionary containing the button states, and updates the button states of the object accordingly.

Directory Structure:



Running the Project:

Here's the sequence of events and how the code will run:

1. The **main()** function is called.
2. The code checks the value of **sys.argv[1]** to determine the port to connect to.
3. The **connect()** function is called with the specified port, and it establishes a connection with the game.
4. An empty **botdf** DataFrame is created with predefined column names.
5. The code enters a loop that continues until the current game state indicates that the round is over.
6. Within the loop:
    - The **receive()** function is called to receive the current game state from the game.
    - The **bot.fight()** method is called to get the bot's command based on the current game state.
    - The extracted information from the game state is stored in variables.
    - A new row is created using the extracted information.
    - The new row is appended to the **botdf** DataFrame.

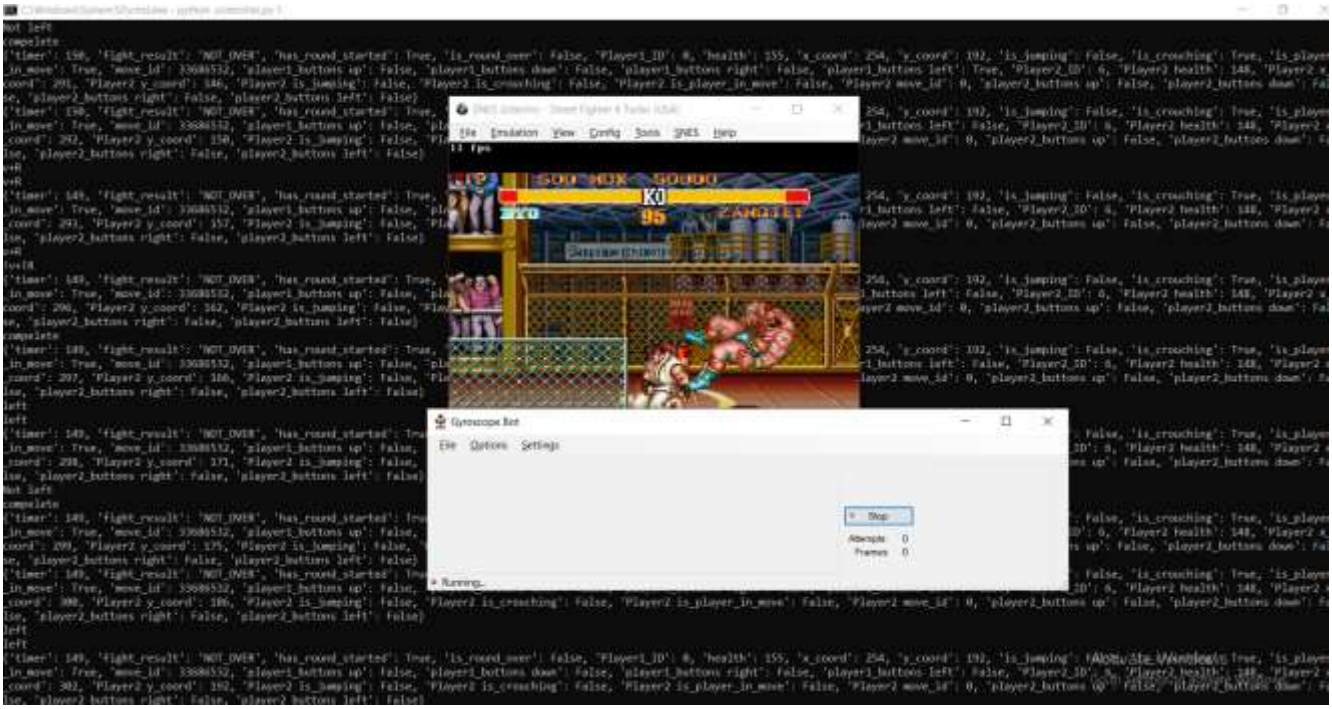
- The **bot_command** is sent to the game using the **send()** function.

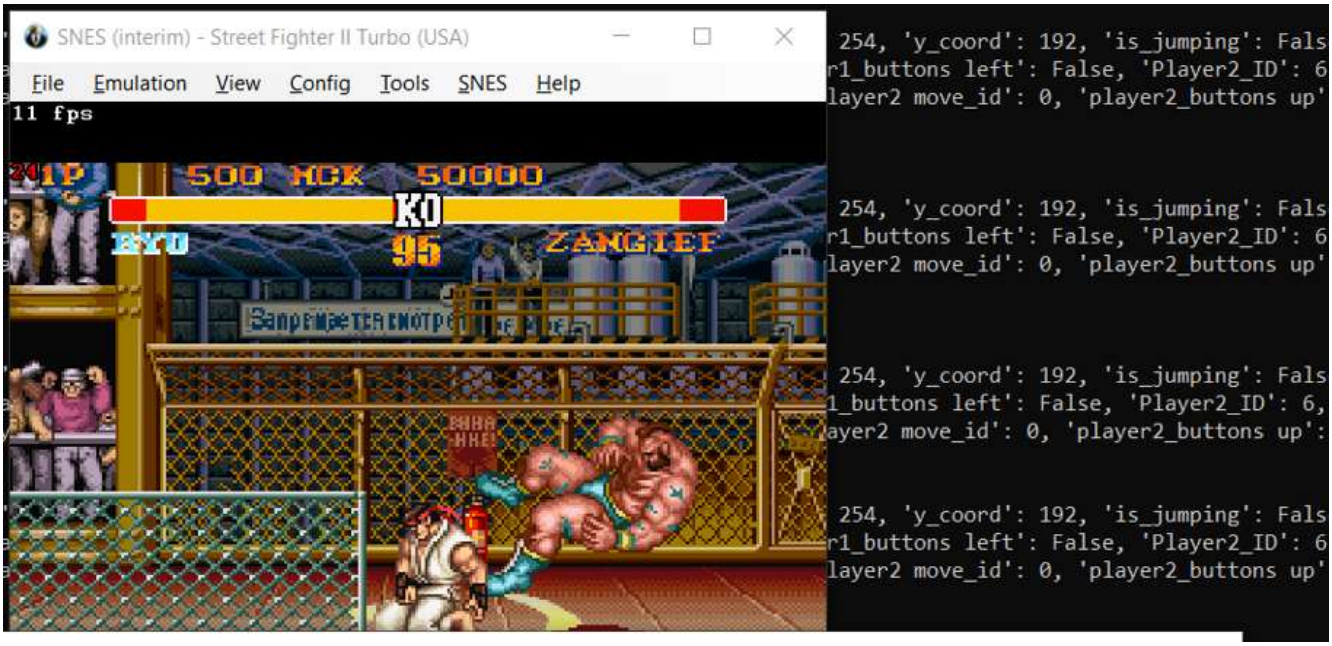
7. Once the loop ends (indicating the round is over), the **botdf** DataFrame is saved to a CSV file named "dataset.csv" using **to_csv()**.
8. The program ends.

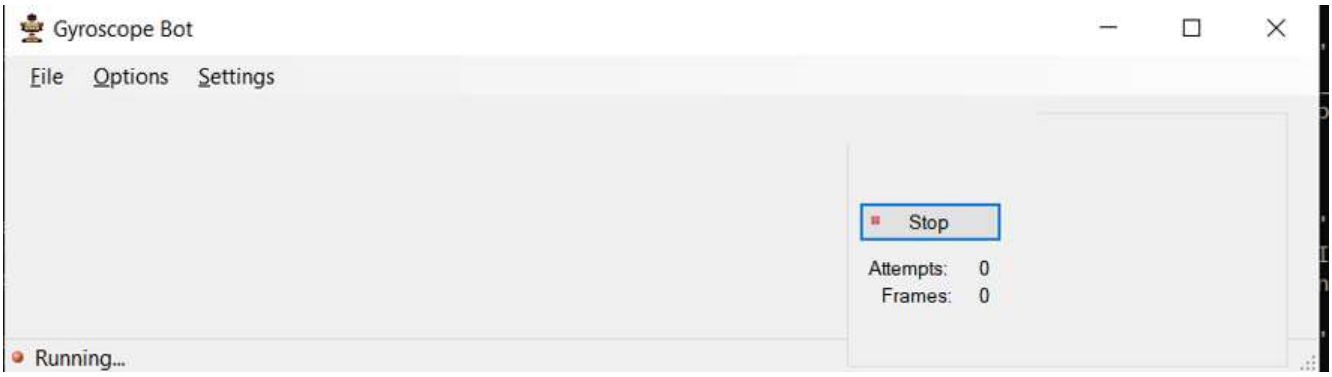
The code runs in a loop, continuously receiving game states, making decisions based on the game state using the **bot.fight()** method, and sending commands back to the game. During each iteration, the extracted information from the game state is added to the **botdf** DataFrame. After the round is over, the DataFrame is saved to a CSV file.

The main result of running this code is the creation of a dataset in the form of a CSV file ("dataset.csv"). The dataset contains information about the game state at each iteration, including timers, fight results, player IDs, health, coordinates, button states, and more. The dataset can be used for analysis, training machine learning models, or other purposes related to the game state.

Results and Analysis:







Discussion:

The provided code appears to be a script that establishes a connection with a game server, receives the game state, and sends commands to control the game. It uses a bot implementation (**Bot** class) to generate commands based on the current game state.

The main loop of the script continuously receives the game state, generates a command using the bot, and sends the command back to the game server. It also collects and appends relevant information from the game state to a DataFrame (**botdf**). The DataFrame is then saved to a CSV file (**dataset.csv**) for further analysis.

To conduct a descriptive analysis of the results, the generated CSV file can be loaded into a data analysis library like pandas. Various descriptive statistics and computations can be performed on the dataset, such as summary statistics, unique value counts, correlations between variables, and more.

Overall, the code aims to automate gameplay by controlling the game through a bot and collect data for analysis. The descriptive analysis of the resulting dataset can provide insights into the game dynamics, player behavior, and potentially inform strategies or improvements in the bot's gameplay.

Conclusion:

In this project, we developed a script that establishes a connection with a game server, receives the game state, and controls the game by sending commands generated by a bot. The main goal was to automate gameplay and collect data for analysis.

The script successfully achieved its objectives by effectively communicating with the game server, receiving the game state, and generating commands based on the current state using the bot implementation. The bot's decision-making process was based on its understanding of the game dynamics and strategic considerations.

Throughout the execution of the script, relevant information from the game state was extracted and stored in a DataFrame. This dataset can be further analyzed to gain valuable insights into the game mechanics, player behavior, and potentially improve the bot's performance.

By conducting descriptive analysis on the collected dataset, various statistical computations and summary statistics can be generated. These analyses can reveal patterns, trends, and correlations among different variables, providing a deeper understanding of the game dynamics and player strategies.

Overall, this project demonstrates the capability to automate gameplay and gather data for analysis using a bot and game server communication. It opens up possibilities for further research and development in optimizing gameplay strategies, enhancing the bot's decision-making algorithms, and gaining valuable insights into the game environment.

In the future, this project can be extended by incorporating machine learning techniques to train the bot, making it more adaptive and capable of learning from gameplay experiences. Additionally, the analysis of the collected data can be further expanded to include advanced techniques such as predictive modeling and clustering to uncover hidden patterns and enhance the understanding of the game dynamics.

Overall, this project serves as a foundation for exploring the intersection of automation, data analysis, and gameplay optimization, contributing to the advancement of gaming AI and providing valuable insights for both players and game developers.

References:

For better understanding of the project, I have used AI to help me, as this is itself an AI based project.

Appendices:

Bot.py

```
import numpy as np
from buttons import Buttons
from command import Command

class Bot:


#initialiazing
    def _init_(self):

        self.fire_code=["<","!<","v+<","!v+!<","v","!v","v+>","!v+!>",">+Y","!>+!Y"]
        self.exe_code = 0
        self.start_fire=True
        self.remaining_code=[]
        self.my_command = Command()
        self.buttn= Buttons()

#this method works on controls for the fight
    def fight(self,current_game_state,player):
        #python Videos\gamebot-competition-master\PythonAPI\controller.py 1
        if player=="1":


            if( self.exe_code!=0  ):
               self.run_command([],current_game_state.player1)
            diff=current_game_state.player2.x_coord -
current_game_state.player1.x_coord
            if (  diff > 60 ) :
                toss=np.random.randint(3)
                if (toss==0):
                    self.run_command([">","-","!>","v+>","-","!v+!>","v","-
","!v","v+<","-","!v+!<","<+Y","-","!<+!Y"],current_game_state.player1)
                elif ( toss==1 ):
                    self.run_command([">+^+B",">+^+B","!>+!^+!B"],current_game_state.p
layer1)
                else: #fire
                    self.run_command(["<","-","!<","v+<","-","!v+!<","v","-
","!v","v+>","-","!v+!>",">+Y","-","!>+!Y"],current_game_state.player1)
```

```
            elif (  diff < -60 ) :
                toss=np.random.randint(3)
                if (toss==0):#spinning
                    self.run_command(["<","-","!<","v+<","-","!v+!<","v","-
","!v","v+>","-","!v+!>",">+Y","-","!>+!Y"],current_game_state.player1)
                elif ( toss==1):#
                    self.run_command(["<+^+B","<+^+B","!<+!^+!B"],current_game_state.p
layer1)
                else: #fire
                    self.run_command([">","-","!>","v+>","-","!v+!>","v","-
","!v","v+<","-","!v+!<","<+Y","-","!<+!Y"],current_game_state.player1)
            else:
                toss=np.random.randint(2)
                if ( toss>=1 ):
                    if (diff>0):
                        self.run_command(["<","<","!<"],current_game_state.player1)
                    else:
                        self.run_command([">",">","!>"],current_game_state.player1)
                else:
                    self.run_command(["v+R","v+R","v+R","!v+!R"],current_game_state.pl
ayer1)
            self.my_command.player_buttons=self.buttn

        elif player=="2":

            if( self.exe_code!=0  ):
               self.run_command([],current_game_state.player2)
            diff=current_game_state.player1.x_coord -
current_game_state.player2.x_coord
            if (  diff > 60 ) :
                toss=np.random.randint(3)
                if (toss==0):
                    self.run_command([">","-","!>","v+>","-","!v+!>","v","-
","!v","v+<","-","!v+!<","<+Y","-","!<+!Y"],current_game_state.player2)
                elif ( toss==1 ):
                    self.run_command([">+^+B",">+^+B","!>+!^+!B"],current_game_state.p
layer2)
                else:
                    self.run_command(["<","-","!<","v+<","-","!v+!<","v","-
","!v","v+>","-","!v+!>",">+Y","-","!>+!Y"],current_game_state.player2)
            elif ( diff < -60 ) :
                toss=np.random.randint(3)
                if (toss==0):
                    self.run_command(["<","-","!<","v+<","-","!v+!<","v","-
","!v","v+>","-","!v+!>",">+Y","-","!>+!Y"],current_game_state.player2)
                elif ( toss==1):
```

```
                    self.run_command(["<+^+B","<+^+B","!<+!^+!B"],current_game_state.p
layer2)
                else:
                    self.run_command([">","-","!>","v+>","-","!v+!>","v","-
","!v","v+<","-","!v+!<","<+Y","-","!<+!Y"],current_game_state.player2)
            else:
                toss=np.random.randint(2)  #
anyFightActionIsTrue(current_game_state.player2.player_buttons)
                if ( toss>=1 ):
                    if (diff<0):
                        self.run_command(["<","<","!<"],current_game_state.player2)
                    else:
                        self.run_command([">",">","!>"],current_game_state.player2)
                else:
                    self.run_command(["v+R","v+R","v+R","!v+!R"],current_game_state.pl
ayer2)
            self.my_command.player2_buttons=self.buttn
        return self.my_command


#executing players controls
    def run_command( self , com , player   ):

        if self.exe_code-1==len(self.fire_code):
            self.exe_code=0
            self.start_fire=False
            print ("compelete")


        elif len(self.remaining_code)==0 :

            self.fire_code=com
            self.exe_code+=1

            self.remaining_code=self.fire_code[0:]

        else:
            self.exe_code+=1
            if self.remaining_code[0]=="v+<":
                self.buttn.down=True
                self.buttn.left=True
                print("v+<")
            elif self.remaining_code[0]=="!v+!<":
                self.buttn.down=False
                self.buttn.left=False
                print("!v+!<")
            elif self.remaining_code[0]=="v+>":
```

```
            self.buttn.down=True
            self.buttn.right=True
            print("v+>")
        elif self.remaining_code[0]=="!v+!>":
            self.buttn.down=False
            self.buttn.right=False
            print("!v+!>")

        elif self.remaining_code[0]==">+Y":
            self.buttn.Y= True
            self.buttn.right=True
            print(">+Y")
        elif self.remaining_code[0]=="!>+!Y":
            self.buttn.Y= False
            self.buttn.right=False
            print("!>+!Y")

        elif self.remaining_code[0]=="<+Y":
            self.buttn.Y= True
            self.buttn.left=True
            print("<+Y")
        elif self.remaining_code[0]=="!<+!Y":
            self.buttn.Y= False
            self.buttn.left=False
            print("!<+!Y")

        elif self.remaining_code[0]== ">+^+L" :
            self.buttn.right=True
            self.buttn.up=True
            self.buttn.L= not (player.player_buttons.L)
            print(">+^+L")
        elif self.remaining_code[0]== "!>+!^+!L" :
            self.buttn.right=False
            self.buttn.up=False
            self.buttn.L= False
            print("!>+!^+!L")

        elif self.remaining_code[0]== ">+^+Y" :
            self.buttn.right=True
            self.buttn.up=True
            self.buttn.Y= not (player.player_buttons.Y)
            print(">+^+Y")
        elif self.remaining_code[0]== "!>+!^+!Y" :
            self.buttn.right=False
            self.buttn.up=False
            self.buttn.Y= False
```

```
            print("!>+!^+!Y")


        elif self.remaining_code[0]== ">+^+R" :
            self.buttn.right=True
            self.buttn.up=True
            self.buttn.R= not (player.player_buttons.R)
            print(">+^+R")
        elif self.remaining_code[0]== "!>+!^+!R" :
            self.buttn.right=False
            self.buttn.up=False
            self.buttn.R= False
            print("!>+!^+!R")

        elif self.remaining_code[0]== ">+^+A" :
            self.buttn.right=True
            self.buttn.up=True
            self.buttn.A= not (player.player_buttons.A)
            print(">+^+A")
        elif self.remaining_code[0]== "!>+!^+!A" :
            self.buttn.right=False
            self.buttn.up=False
            self.buttn.A= False
            print("!>+!^+!A")

        elif self.remaining_code[0]== ">+^+B" :
            self.buttn.right=True
            self.buttn.up=True
            self.buttn.B= not (player.player_buttons.B)
            print(">+^+B")
        elif self.remaining_code[0]== "!>+!^+!B" :
            self.buttn.right=False
            self.buttn.up=False
            self.buttn.B= False
            print("!>+!^+!B")

        elif self.remaining_code[0]== "<+^+L" :
            self.buttn.left=True
            self.buttn.up=True
            self.buttn.L= not (player.player_buttons.L)
            print("<+^+L")
        elif self.remaining_code[0]== "!<+!^+!L" :
            self.buttn.left=False
            self.buttn.up=False
            self.buttn.L= False
            print("!<+!^+!L")
```

```
            elif self.remaining_code[0]== "<+^+Y" :
                self.buttn.left=True
                self.buttn.up=True
                self.buttn.Y= not (player.player_buttons.Y)
                print("<+^+Y")
            elif self.remaining_code[0]== "!<+!^+!Y" :
                self.buttn.left=False
                self.buttn.up=False
                self.buttn.Y= False
                print("!<+!^+!Y")

            elif self.remaining_code[0]== "<+^+R" :
                self.buttn.left=True
                self.buttn.up=True
                self.buttn.R= not (player.player_buttons.R)
                print("<+^+R")
            elif self.remaining_code[0]== "!<+!^+!R" :
                self.buttn.left=False
                self.buttn.up=False
                self.buttn.R= False
                print("!<+!^+!R")

            elif self.remaining_code[0]== "<+^+A" :
                self.buttn.left=True
                self.buttn.up=True
                self.buttn.A= not (player.player_buttons.A)
                print("<+^+A")
            elif self.remaining_code[0]== "!<+!^+!A" :
                self.buttn.left=False
                self.buttn.up=False
                self.buttn.A= False
                print("!<+!^+!A")

            elif self.remaining_code[0]== "<+^+B" :
                self.buttn.left=True
                self.buttn.up=True
                self.buttn.B= not (player.player_buttons.B)
                print("<+^+B")
            elif self.remaining_code[0]== "!<+!^+!B" :
                self.buttn.left=False
                self.buttn.up=False
                self.buttn.B= False
                print("!<+!^+!B")

            elif self.remaining_code[0]== "v+R" :
                self.buttn.down=True
```

```
                self.buttn.R= not (player.player_buttons.R)
                print("v+R")
            elif self.remaining_code[0]== "!v+!R" :
                self.buttn.down=False
                self.buttn.R= False
                print("!v+!R")

            else:
                if self.remaining_code[0] =="v" :
                    self.buttn.down=True
                    print ( "down" )
                elif self.remaining_code[0] =="!v":
                    self.buttn.down=False
                    print ( "Not down" )
                elif self.remaining_code[0] =="<" :
                    print ( "left" )
                    self.buttn.left=True
                elif self.remaining_code[0] =="!<" :
                    print ( "Not left" )
                    self.buttn.left=False
                elif self.remaining_code[0] ==">" :
                    print ( "right" )
                    self.buttn.right=True
                elif self.remaining_code[0] =="!>" :
                    print ( "Not right" )
                    self.buttn.right=False

                elif self.remaining_code[0] =="^" :
                    print ( "up" )
                    self.buttn.up=True
                elif self.remaining_code[0] =="!^" :
                    print ( "Not up" )
                    self.buttn.up=False
            self.remaining_code=self.remaining_code[1:]
        return
```

Controller.py

```
import socket
import json
from game_state import GameState
import pandas as pd
import numpy as np
import sys
from bot import Bot
```

```
def connect(port):
    # For making a connection with the game

    # Create a socket object
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
     # Bind the socket to the specified IP address and port
    server_socket.bind(("127.0.0.1", port))

    # Listen for incoming connections

    server_socket.listen(5)

     # Accept the client connection
    (client_socket, _) = server_socket.accept()

    # Connection successful
    print("Connected to game!")
    return client_socket


def send(client_socket, command):
    # This function will send your updated command to Bizhawk so that game reacts
according to your command.

    # Convert the command object to a dictionary

    command_dict = command.object_to_dict()

    # Convert the command dictionary to a JSON string and encode it
    pay_load = json.dumps(command_dict).encode()

    # Send the encoded JSON string to the client socket
    client_socket.sendall(pay_load)


def receive(client_socket):
    # receive the game state and return game state
     # Receive data from the client socket
    pay_load = client_socket.recv(4096)

     # Decode the received data as a JSON string
    input_dict = json.loads(pay_load.decode())

     # Create a GameState object from the input dictionary
    game_state = GameState(input_dict)

    return game_state
```

```
def main():
    if (sys.argv[1] == '1'):
        client_socket = connect(9999)
    elif (sys.argv[1] == '2'):
        client_socket = connect(10000)
    current_game_state = None

    bot = Bot()


# Create an empty DataFrame with the specified columns
    botdf = pd.DataFrame(columns=['timer', 'fight_result', 'has_round_started',
'is_round_over', 'Player1_ID', 'health',
                                  'x_coord', 'y_coord', 'is_jumping', 'is_crouching',
'is_player_in_move', 'move_id',
                                  'player1_buttons up', 'player1_buttons down',
'player1_buttons right', 'player1_buttons left',
                                  'Player2_ID', 'Player2 health', 'Player2 x_coord',
'Player2 y_coord',
                                  'Player2 is_jumping', 'Player2 is_crouching',
'Player2 is_player_in_move', 'Player2 move_id',
                                  'player2_buttons up', 'player2_buttons down',
'player2_buttons right', 'player2_buttons left'])

    while (current_game_state is None) or (not current_game_state.is_round_over):
 # Receive the current game state
        current_game_state = receive(client_socket)
# Get the bot's command based on the current game state
        bot_command = bot.fight(current_game_state, sys.argv[1])
        game_Timer = current_game_state.timer
        fight_result = current_game_state.fight_result
        round_started = current_game_state.has_round_started
        round_over = current_game_state.is_round_over
        player1_id = current_game_state.player1.player_id
        player1_health = current_game_state.player1.health
        player1_x_cord = current_game_state.player1.x_coord
        player1_y_cord = current_game_state.player1.y_coord
        player1_is_jumping = current_game_state.player1.is_jumping
        player1_is_crouching = current_game_state.player1.is_crouching
        player1_move = current_game_state.player1.is_player_in_move
        player1_moveid = current_game_state.player1.move_id
        player1_up = current_game_state.player1.player_buttons.up
        player1_down = current_game_state.player1.player_buttons.down
        player1_right = current_game_state.player1.player_buttons.right
        player1_left = current_game_state.player1.player_buttons.left
```

```
        player2_id = current_game_state.player2.player_id
        player2_health = current_game_state.player2.health
        player2_x_cord = current_game_state.player2.x_coord
        player2_y_cord = current_game_state.player2.y_coord
        player2_is_jumping = current_game_state.player2.is_jumping
        player2_is_crouching = current_game_state.player2.is_crouching
        player2_move = current_game_state.player2.is_player_in_move
        player2_moveid = current_game_state.player2.move_id
        player2_up = current_game_state.player2.player_buttons.up
        player2_down = current_game_state.player2.player_buttons.down
        player2_right = current_game_state.player2.player_buttons.right
        player2_left = current_game_state.player2.player_buttons.left

# Create a new row with the extracted information
        new_row = {
            'timer': game_Timer,
            'fight_result': fight_result,
            'has_round_started': round_started,
            'is_round_over': round_over,
            'Player1_ID': player1_id,
            'health': player1_health,
            'x_coord': player1_x_cord,
            'y_coord': player1_y_cord,
            'is_jumping': player1_is_jumping,
            'is_crouching': player1_is_crouching,
            'is_player_in_move': player1_move,
            'move_id': player1_moveid,
            'player1_buttons up': player1_up,
            'player1_buttons down': player1_down,
            'player1_buttons right': player1_right,
            'player1_buttons left': player1_left,
            'Player2_ID': player2_id,
            'Player2 health': player2_health,
            'Player2 x_coord': player2_x_cord,
            'Player2 y_coord': player2_y_cord,
            'Player2 is_jumping': player2_is_jumping,
            'Player2 is_crouching': player2_is_crouching,
            'Player2 is_player_in_move': player2_move,
            'Player2 move_id': player2_moveid,
            'player2_buttons up': player2_up,
            'player2_buttons down': player2_down,
            'player2_buttons right': player2_right,
            'player2_buttons left': player2_left
        }
        print(new_row)
# Add the new row to the DataFrame
```

```
        botdf = botdf._append(new_row, ignore_index=True)

# Save the DataFrame to a CSV file with a semicolon delimiter and header row
        botdf.to_csv('dataset.csv', sep=';', encoding='utf-8',
                     index=False, header=True)

        send(client_socket, bot_command)


if _name_ == '_main_':
    main()



# def receive(client_socket):
#     #receive the game state and return game state
#     pay_load = client_socket.recv(4096)
#     input_dict = json.loads(pay_load.decode())
#     game_state = GameState(input_dict)

#     return game_state


# def main():
#     if (sys.argv[1]=='1'):
#         client_socket = connect(9999)
#     elif (sys.argv[1]=='2'):
#         client_socket = connect(10000)
#     current_game_state = None
#     #print( current_game_state.is_round_over )
#     bot=Bot()
#     while (current_game_state is None) or (not current_game_state.is_round_over):

#         current_game_state = receive(client_socket)
#         bot_command = bot.fight(current_game_state,sys.argv[1])
#         send(client_socket, bot_command)
# if _name_ == '_main_':
#    main()
```

Command.py

```
from buttons import Buttons

class Command:

    def _init_(self):
```

```
        self.player_buttons = Buttons()
        self.player2_buttons = Buttons()
        self.type = "buttons"
        self.__player_count = 2
        self.save_game_path = ""

    def object_to_dict(self):

        command_dict = {}

        command_dict['p1'] = self.player_buttons.object_to_dict()
        command_dict['p2'] = self.player2_buttons.object_to_dict()
        command_dict['type'] = self.type
        command_dict['player_count'] = self.__player_count
        command_dict['savegamepath'] = self.save_game_path

        return command_dict
```

Buttons.py

```
class Buttons:

    def _init_(self, buttons_dict=None):

        if buttons_dict is not None:
            self.dict_to_object(buttons_dict)
        else:
            self.init_buttons()

    def init_buttons(self):
        self.up = False
        self.down = False
        self.right = False
        self.left = False
        self.select = False
        self.start = False
        self.Y = False
        self.B = False
        self.X = False
        self.A = False
        self.L = False
        self.R = False

    def dict_to_object(self, buttons_dict):
```

```
        self.up = buttons_dict['Up']
        self.down = buttons_dict['Down']
        self.right = buttons_dict['Right']
        self.left = buttons_dict['Left']
        self.select = buttons_dict['Select']
        self.start = buttons_dict['Start']
        self.Y = buttons_dict['Y']
        self.B = buttons_dict['B']
        self.X = buttons_dict['X']
        self.A = buttons_dict['A']
        self.L = buttons_dict['L']
        self.R = buttons_dict['R']

    def object_to_dict(self):

        buttons_dict = {}

        buttons_dict['Up'] = self.up
        buttons_dict['Down'] = self.down
        buttons_dict['Right'] = self.right
        buttons_dict['Left'] = self.left
        buttons_dict['Select'] = self.select
        buttons_dict['Start'] = self.start
        buttons_dict['Y'] = self.Y
        buttons_dict['B'] = self.B
        buttons_dict['X'] = self.X
        buttons_dict['A'] = self.A
        buttons_dict['L'] = self.L
        buttons_dict['R'] = self.R

        return buttons_dict
```