National University of Computer and Emerging Sciences, Islamabad

FAST School of Computing

Artificial Intelligence Project

AI-Powered Real-Time Game Bot

Fahid Imran (23i-0061) Umer Ahmad (22i-0582)

Abdullah Qadri (23i-0089) Adam Afridi (23i-0055)

Bachelors of Artificial Intelligence

Submission Date: 9 May, 2025

Lecturer: Dr. Hasan Mujtaba

Project Report

AI-Powered Real-Time Game Bot

Objective

The objective of this project is to train a neural network that watches player behavior in the fighting game (Street Fighter II Turbo) and then uses that knowledge to automatically control a character in a game by predicting what buttons to press at each frame.

Files Overview

File	Purpose
train_model.py	Trains the AI model using game state data
bot.py	Uses the trained model to control the bot during gameplay
controller.py	Connects the emulator with the bot code (game state communication)
command.py, buttons.py	Game command and button structure definitions
trained_model.h5, data.pkl	Saved model and scaler used at runtime

Libraries Used:

- Pandas/Numpy: Data handling
- Scikit-learn: Data preprocessing & splitting
- TensorFlow/Keras: Deep learning framework
- Joblib: Model and scaler saving/loading
- Matplotlib: Visualization

Part 1: Data Generation Process

To train an AI model, we needed initially data that truly represented how a human plays the game. Implemented steps were to:

Creating Data Step-By-Step:

- We changed the bot.py file and set player1 buttons according to the buttons pressed from keyboard.
- We manually play Street Fighter II Turbo utilizing our keyboard keys.
- While playing, the game state was being captured in real-time.
- We altered the fight() function in bot.py to capture which keys were pressed at every frame (using keyboard library).
- This input was converted to button values (True/False) for player1.
- The game states + player1 button presses were saved as a CSV file for every frame.

Part 2: Model Training (Used muti-layer Perseptron Model)

1. Data Cleaning & Preprocessing

Data Cleaning: Strips white space, removes missing/null values.
Boolean Mapping: Converts True/False to 1/0.
Label Encoding: Converts string categories (like result) to numeric values.

2. Feature Selection

Separate input columns and out put columns.

```
input_columns = ['frame', 'timer', 'result', 'round_started', 'round_over',
            'height_delta', 'width_delta', 'player2_character',
            'player2_health', 'player2_x', 'player2_y',
            'player2_jumping', 'player2_crouching',
            'player2_in_move', 'player2_move']

output_columns = ['player1_Up', 'player1_Down', 'player1_Left', 'player1_Right',
            'player1_Select', 'player1_Start', 'player1_Y', 'player1_B',
            'player1_X', 'player1_A', 'player1_L', 'player1_R']
```

3. Data Normalization

In order to stabilize the training of the neural network and ensure efficient training, we standardize the input features. Standardization is managed by the StandardScaler from scikit-learn and scales each feature so the mean is 0 and the standard deviation is 1. It is particularly important to standardize because the game state features have different possible value ranges (e.g., timer vs. position); if we do not, features with larger values can have a much larger impact on the learning processes. Standardization makes it more likely that all features will have similar contributions from the start and should allow for faster convergence and more accurate predictions. The same scaler will be used for both the training and gameplay to standardize the input in the same way.

4. Model Definition

We create a multi-layer perceptron (MLP) model based on sequential approach for simplicity and more clarity. It is a model consisting of hidden layers that allow the model to learn complex, nonlinear patterns in the game data.

We use the Relu activation in hidden layers because it is fast, simple, and effective for model training. The output layer uses Sigmoid since we solve it as a multi-label classification problem, meaning each button (up, down, A, B, etc.) is predicted and treated independently from other buttons as pressed or not pressed.

In order to reduce overfitting during training we utilize Dropout, which provides the ability to randomly turn off some neurons during training to prevent the model from simply memorizing patterns, forcing the model to learn more robust patterns.

5. Model Training

The Adam Optimizer is used to train the model; it's a sophisticated optimization method that automatically adjusts the learning rate throughout training. This enables the model to converge faster and more efficiently during training on larger and more complex data sets.

The loss function is Binary Crossentropy, which is appropriate for multi-label classification problems. In this case, the model predicts 12 outputs (1 for each button), and since each of these outputs is independent of one another, several buttons can be pressed at once. Binary

Crossentropy evaluates each output's predicted value (between 0-1) and its actual label (0 or 1) to calculate error. This error is then used to maximize the training process and improve the output prediction as the model improves.

6. Saving the Model

The first is the trained model. It gets saved to a file called trained_model.h5. The file stores everything the model has learned from the training dataset, such as the structure of the model, the weights used in the model and the parameters that detail how the model was trained. This file means we do not need to retrain the model each time we want to use it in practice.

The second thing that will be saved is the data normalizer (StandardScaler), which we have used to scale the input features as we trained the model. This was saved to a file called data.pkl, using joblib to serialize it to a file. This means that when we then use the model to make predictions in gameplay, we will transform the game input in exactly the same way as the training data was transformed.

part 3: Real-Time Prediction Bot (bot.py)

This file is responsible for making the AI bot, play the game during run time, meaning it uses the model to predict what buttons should be pushed in every frame of the game. Here is how the process takes places:

1. Load the Model and Scaler

First thing the bot does is load the following:

- The trained model (trained_model.h5) that has learned how to play the game.
- The scaler (data.pkl) that was used to scale the input data during training.

This is important since we want to make sure the input features we give to the model during the game are processed the same way they were during training.

2. Read Game State Information

The bot gathers information about the current state of the game frame. It collects information such as:

- Timer value
- Player positions
- Health
- Whether the player is jumping, crouching or moving

These 15 features are read into the same order as it was during training, which ensures the model knows what each number represents.

3. Normalize and Predict

The game data is normalized using the previously saved scaler to make the data ready for the model.

The model makes a prediction for each of the twelve buttons (up, down, A, B, etc.) giving a probability of 0 to 1 for each button.

To determine whether we will press a button or not we can use a threshold of 0.5:

- If probability > 0.5 → press the button (set to True)
- If probability < 0.5 → do not press the button (set to False)

4. Set the Button Presses

The model predictions are then connected to actual game buttons. For example:

if prediction[0] is 1 → press the up button
if prediction[3] is 0 → do not press the right button etc.

The bot provides the game controller with the new button states. The button presses are sent to the emulator, and the game character acts out the predictions automatically.