Project part 2 report

Objective:

The objective of this report is to outline the steps involved in integrating a trained neural network model into a game bot for generating button actions. The integration process involves preprocessing the data, splitting it into training and test sets, training the model, saving the trained model, and modifying the bot.py file to load the model for generating button actions.

1. ***Preprocess data:***

The first step in integrating the trained neural network model into the game bot is to preprocess the data. This involves applying scaling and normalization techniques to the data file. Preprocessing the data helps to ensure that the input features are on a similar scale and reduces the impact of outliers or variations in the data. By performing scaling and normalization, we can enhance the performance and accuracy of the neural network model.

```
dropColumns=['player2_move_id','player1_move_id','player2_is_player_in_move','player1_is_player_in_move','player2_is_crouching','
data=data.drop(dropColumns,axis=1)
```

```
data
```

	player1_health	player1_x_coord	player1_y_coord	player1_button_up	player1_button_down	player1_button_right	player1_button_left	player1_button_Y	p
0	176	205	192	0	0	0	0	0	
1	176	205	192	0	0	0	0	0	
2	176	205	192	0	0	1	0	0	
3	176	205	192	0	1	1	0	0	
4	176	205	192	0	0	0	0	0	
...	...	...	...	...	...	...	...	...	
11959	2	369	192	0	0	0	0	0	
11960	2	369	192	0	0	1	0	1	
11961	2	369	192	0	0	0	0	0	
11962	2	369	192	0	0	0	0	0	
11963	255	404	192	1	0	1	0	0	

11964 rows × 26 columns

```
Player1['Player1_Movements'] = Player1['Player1_Movement'].apply(lambda x: int(x, 2))
Player1['Player1_Attacks'] = Player1['Player1_Attack'].apply(lambda x: int(x, 2))
```

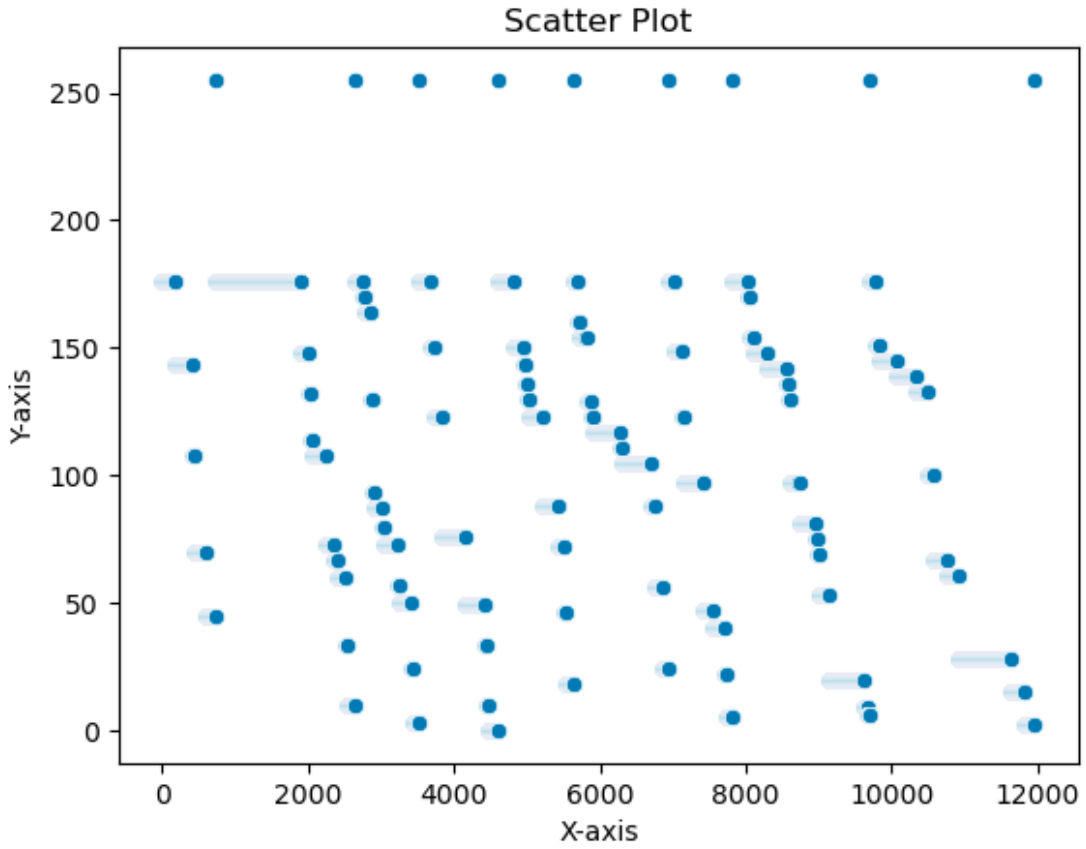
```
Player1
```

	player1_health	player1_x_coord	player1_y_coord	Player1_Movement	Player1_Attack	Player1_Movements	Player1_Attacks
0	176	205	192	0000	000000	0	0
1	176	205	192	0000	000000	0	0
2	176	205	192	0010	000000	2	0
3	176	205	192	0110	000000	6	0
4	176	205	192	0000	000000	0	0
...	...	...	...	...	...	...	...
11959	2	369	192	0000	000000	0	0
11960	2	369	192	0010	100000	2	32
11961	2	369	192	0000	000000	0	0
11962	2	369	192	0000	000000	0	0
11963	255	404	192	1010	010000	10	16

11964 rows × 7 columns

2. ***<u>Plotting the data:</u>***

Plots provide a visual representation of data, making it easier to understand patterns, trends, and relationships that may not be apparent from raw data alone. Visualizing data can help identify outliers, clusters, correlations, and other insights.



3. ***<u>Split data into training and testing:</u>***

After preprocessing the data, the next step is to split the data into training and test sets. The training set will be used to train the neural network model, while the test set will be used to evaluate the model's performance. This step is crucial to assess the generalization ability of the model on unseen data.

```
# Load the dataset
X = FinalOne[feature_One]
y = FinalTwo[feature_Two]

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

4. ***Define a neural network model:***

Once the data is split, the next step is to define a neural network model. The model architecture is designed based on the requirements of the game and the input data. The neural network model is created using popular deep learning frameworks such as tensorflow. The model will be able to take the preprocessed data as input and generate appropriate button actions as output.

```
input_shape = (X_train.shape[1],)

inputs = tf.keras.Input(shape=input_shape)
x = tf.keras.layers.Dense(64, activation='relu')(inputs)
x = tf.keras.layers.Dense(64, activation='relu')(x)
outputs = tf.keras.layers.Dense(y_train.shape[1], activation='sigmoid')(x)

fightingModel = tf.keras.Model(inputs=inputs, outputs=outputs)

fightingModel.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])


fightingModel.fit(X_train, y_train, epochs=25, batch_size=32, validation_data=(X_test, y_test))
```

5. ***Train the model:***

After defining the neural network model, the next step is to train the model using the training set. Training involves iteratively updating the model's parameters to minimize the difference between the predicted button actions and the expected button actions. The model learns from the input data and adjusts its internal parameters to make accurate predictions. Training the model may require multiple epochs, and appropriate optimization algorithms can be applied to improve convergence and performance.

6. ***Save the trained model:***

Once the model is trained and achieves satisfactory performance, it needs to be saved for future use. Saving the trained model allows us to load and reuse it without having to retrain it from scratch. The saved model can be stored in a file format compatible with the chosen deep learning framework which is in PyTorch's .pt format for our case.

7. ***Modify bot.py File to Load the Trained Model:***
   The final step is to modify the bot.py file to load the trained neural network model and utilize it for generating button actions. The modified code includes the necessary functions to load the saved model and pass the preprocessed data as input to the model. The model will then generate predictions for the button actions based on the provided input. The bot.py file is updated to incorporate these predictions into the game's button action logic.

Conclusion:

In conclusion, the integration of a trained neural network model into a game bot involves several steps, including data preprocessing, splitting the data, defining the neural network model, training the model, saving the trained model, and modifying the bot.py file to utilize the model for generating button actions. These steps ensure the seamless integration of the trained model into the game bot, enhancing its functionality and performance.