Final Project

Group 28

109550092, 111550038

Introduction
The competition described focuses on the analysis of real game data in badminton. It aims to collect badminton data to predict player performance in future strokes and rallies. The competition consists of two tracks.

Track 1: Automatic Annotation of Technical Data for Badminton Match Videos
The goal of this track is to design solutions using computer vision techniques to automatically annotate shot-by-shot data from badminton match videos.

Track 2: Forecasting of Future Turn-Based Strokes in Badminton Rallies
In this track, participants are challenged to design predictive models that can forecast future strokes in badminton rallies. Given the observed past strokes (4 strokes with type-area pairs) and player information, the task is to predict the future strokes(type), including shot types and area coordinates(landing_x,landing_y), for the next steps in the rally. The length of the predicted sequence varies based on the length of the rally. This task requires the development of advanced predictive models to anticipate player actions based on historical data.

Here, the final project is mainly on the practice in building models for the prediction of future strokes in **Track 2**.

Referenced Works
Multivariate Multi-step Time Series Forecasting using Stacked LSTM sequence to sequence Autoencoder in Tensorflow 2.0 / Keras

Suggula Jagadeesh — Published On October 29, 2020

This work on building a time-series predictive model provides easy understanding and guidance on predicting future sequences using existing sequences, including the prediction of future features.

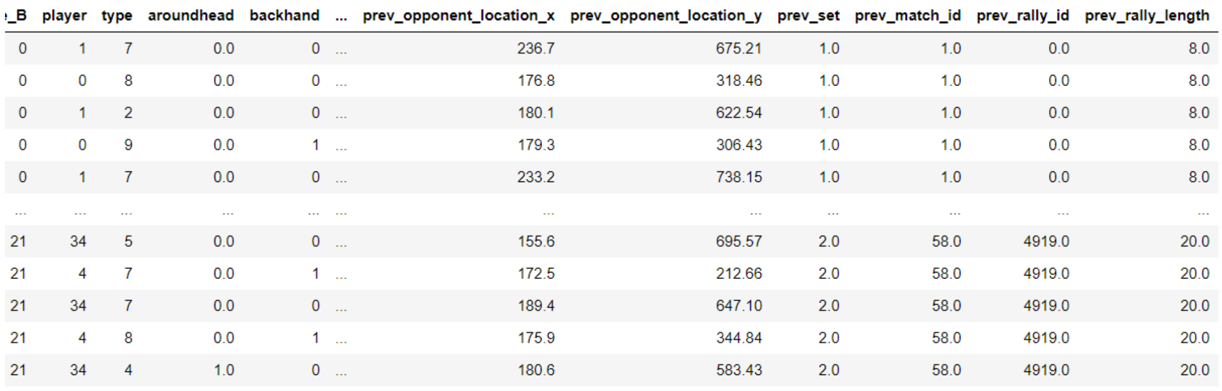
This article discusses the implementation of a stacked sequence-to-sequence LSTM model for time series forecasting using Keras and TensorFlow 2.0. The sequence-to-sequence model consists of an encoder and a decoder, with LSTMs used as the recurrent neural network units. The encoder converts the input sequence into a fixed-length context vector, which is then fed into the decoder along with the final encoder state to predict the output sequence. The article demonstrates how to add additional layers, such as a repeat vector layer and a time-distributed dense layer, to the architecture to enable multi-step forecasting, the step by step guidance is clear and that we follow the implementation and adapt a similar approach in parts of our model. Two model architectures are presented: E1D1 with one encoder layer and one decoder layer, and E2D2 with two encoder layers and two decoder layers. The models are trained using the Adam optimizer and Huber loss, and the predictions are evaluated using mean absolute error. The difference between this work and the final work is that we only use the encoder-decoder system in the first part of our model, predicting future landing coordinates, then we added a final classification NN model for the prediction of types of stroke. One of the reasons to use this approach is considering the correlation and feature importance of the datasets, which will be mentioned more later. The other reason is the failure of building a robust all-in-one model that will also be mentioned in future improvements later.

Data Preprocessing

The preprocessing is mainly on feature selection. The feature selection can be viewed as Inter-stroke, and Intra-stroke, in which we observe the influence using mathematical correlations and random forest feature importance to capture the relationship between different features.

Inter-stroke

Inter-stroke refers to the relationship between different strokes within a given time series. Since the dataset is based on sport data, the previous strokes may affect the next stroke significantly. Here, we decided to merge the data with a shifted dataframe. The merged results contain the features of the previous stroke.
On the screenshot below, you can see shifted columns denoted as “prev_”attribute_name”", for further investigation.



_B	player	type	aroundhead	backhand	...	prev_opponent_location_x	prev_opponent_location_y	prev_set	prev_match_id	prev_rally_id	prev_rally_length
0	1	7	0.0	0	...	236.7	675.21	1.0	1.0	0.0	8.0
0	0	8	0.0	0	...	176.8	318.46	1.0	1.0	0.0	8.0
0	1	2	0.0	0	...	180.1	622.54	1.0	1.0	0.0	8.0
0	0	9	0.0	1	...	179.3	306.43	1.0	1.0	0.0	8.0
0	1	7	0.0	0	...	233.2	738.15	1.0	1.0	0.0	8.0
...	...	...	...	...	...	...	...	...	...	...	...
21	34	5	0.0	0	...	155.6	695.57	2.0	58.0	4919.0	20.0
21	4	7	0.0	1	...	172.5	212.66	2.0	58.0	4919.0	20.0
21	34	7	0.0	0	...	189.4	647.10	2.0	58.0	4919.0	20.0
21	4	8	0.0	1	...	175.9	344.84	2.0	58.0	4919.0	20.0
21	34	4	1.0	0	...	180.6	583.43	2.0	58.0	4919.0	20.0

Intra-stroke

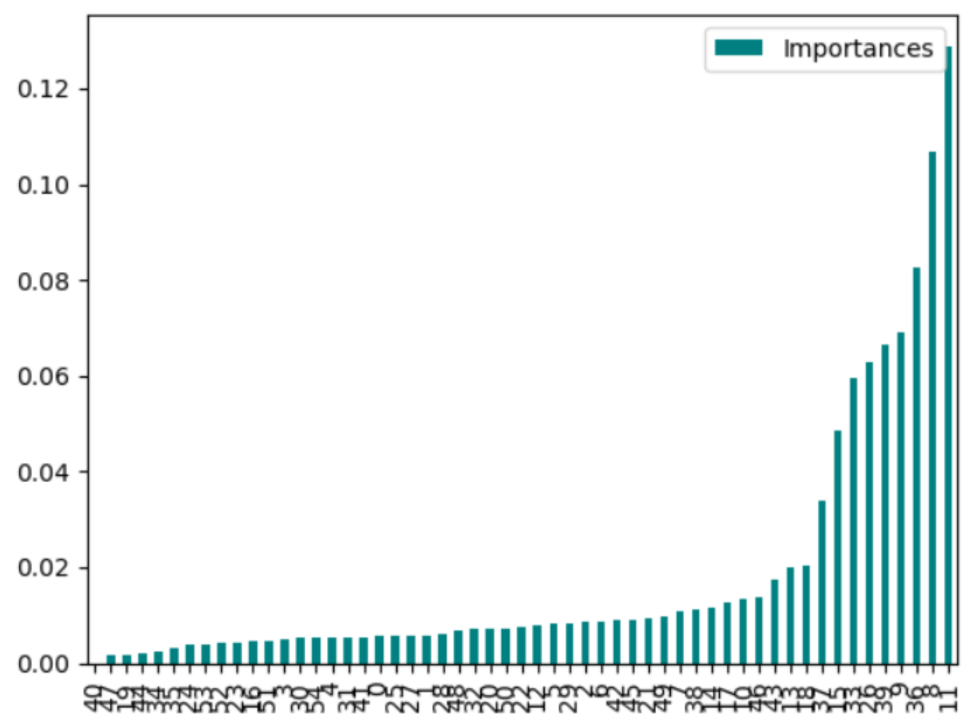
Intra-stroke refers to the relationship of different attributes within a specific stroke, some of the features may have direct relationship and are highly contributed in the classification of the label within a stroke, which is usually included even if the datasets are not on a time series.

A combination of Horizontal and Vertical Feature Selection

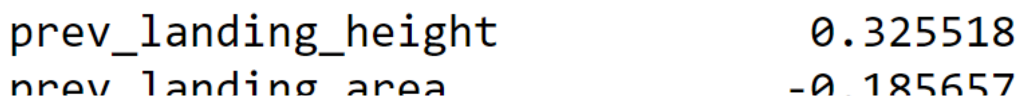
The main purpose here is to try to observe what influenced the future landing coordinates and type of stroke the most. So, first we apply Random Forest Classification on the different type of stroke to observe the importance of features. The “type” attribute is encoded into values from 0 to 9 for the ease of processing. than, we combine the Horizontal features(the original

data columns) and the Vertical features(shifted columns), and plot the feature importance on a RFC with n_estimators = 200
In the plot below, features that are possibly significant are landing_height, landing_y , prev_landing_y, landing_area, prev_landing_height, prev_type and player_location_y.

```
<AxesSubplot: >
```



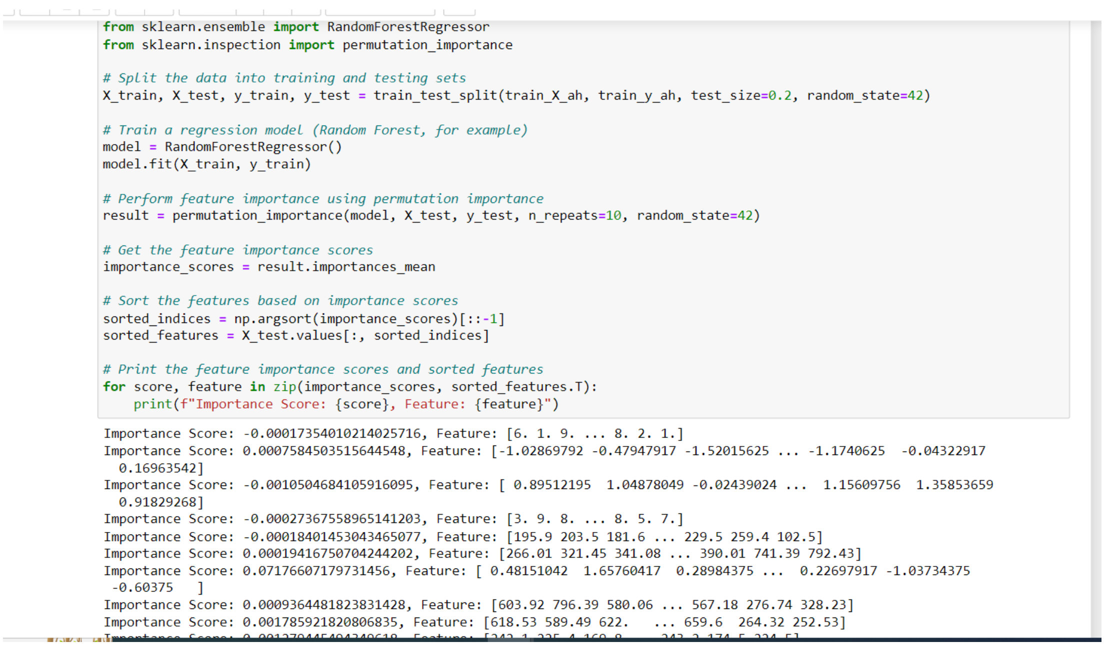
Using mathematical correlations computations, previous landing height is also the most correlated.



```
prev_landing_height             0.325518
prev_landing_area              -0.185657
```

The final choice on the prediction of “type” attributes for us are:
'prev_landing_height', 'prev_landing_area', 'prev_player_location_y','prev_landing_y', 'prev_type','landing_height', 'landing_area', 'player_location_y','landing_y'
Which is mostly determined in the feature selection. For other data such as the set, match, and the metadata, we found it not as significantly correlated with type of strokes, but it may be included with further digging in the future.

For Landing_x and Landing_y coordinates, since it’s not a good approach to apply random forest classifiers on continuous target data, Random Forest Regressor are used instead. Below is part of the coding and results, the importance score indicates the influencing on the coordinates, the Feature is the specific column with values.



```
from sklearn.ensemble import RandomForestRegressor
from sklearn.inspection import permutation_importance

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(train_X_ah, train_y_ah, test_size=0.2, random_state=42)

# Train a regression model (Random Forest, for example)
model = RandomForestRegressor()
model.fit(X_train, y_train)

# Perform feature importance using permutation importance
result = permutation_importance(model, X_test, y_test, n_repeats=10, random_state=42)

# Get the feature importance scores
importance_scores = result.importances_mean

# Sort the features based on importance scores
sorted_indices = np.argsort(importance_scores)[::-1]
sorted_features = X_test.values[:, sorted_indices]

# Print the feature importance scores and sorted features
for score, feature in zip(importance_scores, sorted_features.T):
    print(f"Importance Score: {score}, Feature: {feature}")
```

```
Importance Score: -0.00017354010214025716, Feature: [6. 1. 9. ... 8. 2. 1.]
Importance Score: 0.0007584503515644548, Feature: [-1.02869792 -0.47947917 -1.52015625 ... -1.1740625  -0.04322917
  0.16963542]
Importance Score: -0.0010504684105916095, Feature: [ 0.89512195  1.04878049 -0.02439024 ...  1.15609756  1.35853659
  0.91829268]
Importance Score: -0.00027367558965141203, Feature: [3. 9. 8. ... 8. 5. 7.]
Importance Score: -0.00018401453043465077, Feature: [195.9 203.5 181.6 ... 229.5 259.4 102.5]
Importance Score: 0.00019416750704244202, Feature: [266.01 321.45 341.08 ... 390.01 741.39 792.43]
Importance Score: 0.07176607179731456, Feature: [ 0.48151042  1.65760417  0.28984375 ...  0.22697917 -1.03734375
 -0.60375   ]
Importance Score: 0.0009364481823831428, Feature: [603.92 796.39 580.06 ... 567.18 276.74 328.23]
Importance Score: 0.001785921820806835, Feature: [618.53 589.49 622.   ... 659.6  264.32 252.53]
```

After analyzing the results, the feature we choose for the landing coordinates are:
landing_x landing_y landing_height landing_area player_location_x, player_location_y opponent_location_x opponent_location_y.
We did not mention the previous features here because in later implementation, a encoder-decoder system considers an entire sequence of the 8 features with past 4 strokes is trained together, the LSTM model will conclude the relationship for past and future sequence itself, no need to use lagged features.
Also, the reason we did not put type attributes here is because we found it not the most significant feature in the landing_coordinates, although it does have some correlation, we were struggle to incorporate a robust model with different loss functions for different type of data without the result of gradient exploding and a poor performance. On the other hand, we found landing coordinates have huge influence on type of stroke, so a different approach is conducted, which is, by predicting future sequence of landing coordinates and correlated features, use the predictions for a next layer classification on potential type of stroke, which means the prediction on stroke is on a different NN model.

Scaling

For all the scaling here, MinMax Scaling within range(-1,1) is used. The "type" attribute is not scaled, it's encoded into label value from (0,9) instead.

Splitting The Datasets

For all the models, separate the datasets into 70% training, 20% testing and 10% validating, and the index is shuffled, preventing the datasets all lying in some specific rallies or matches.

```
#split the datasets 70% training, 20% testing, 10% validating

from sklearn.model_selection import train_test_split

# Shuffle the data
indices = np.arange(X.shape[0])
np.random.shuffle(indices)
X_shuffled = X[indices]
y_shuffled = y[indices]

# Split the shuffled data into train, test, and validation sets
X_train, X_test, y_train, y_test = train_test_split(X_shuffled, y_shuffled, test_size=0.3, random_state=
X_test, X_val, y_test, y_val = train_test_split(X_test, y_test, test_size=0.33, random_state=42)
```

Implementation

The coding is mainly in three parts:

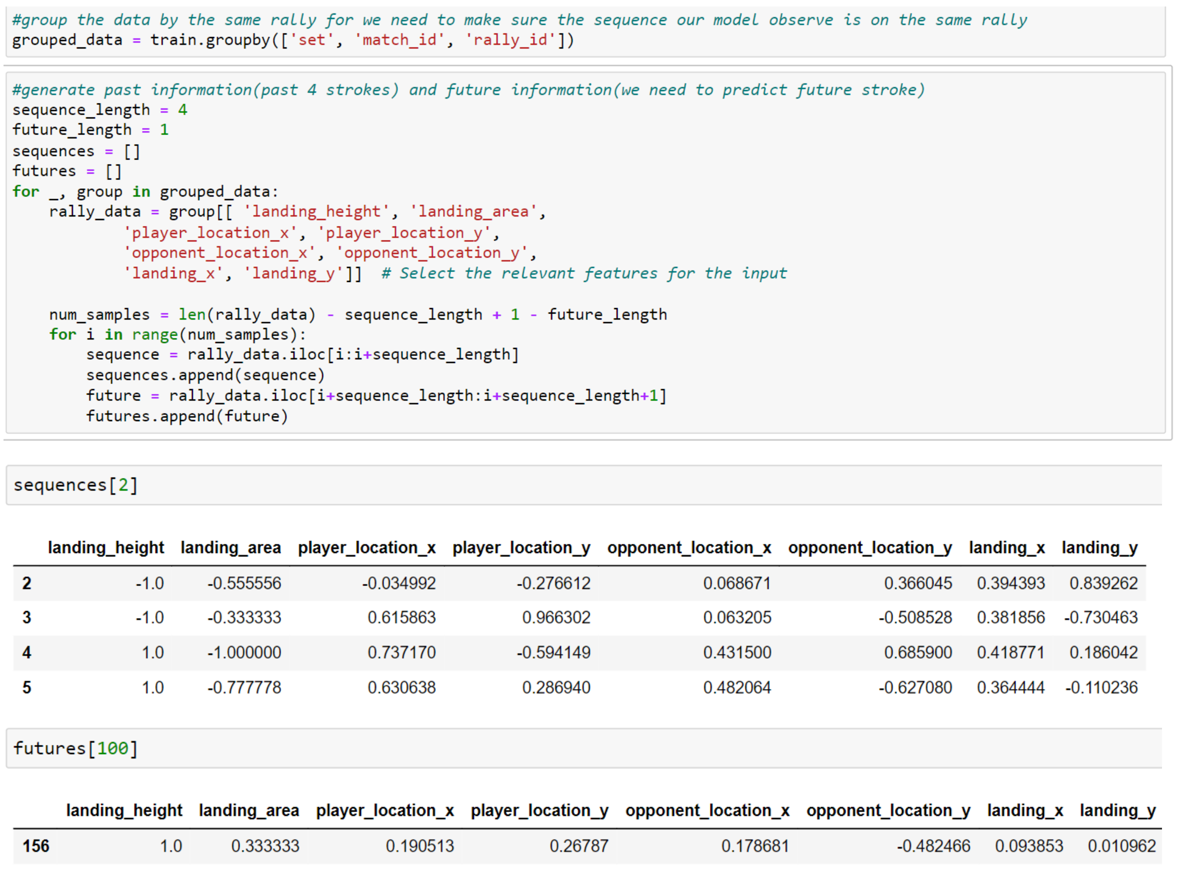
1.Future_Sequence_Coordinates_Predictions
2.Stroke_Type_Classification
3.Final_Inference

Future_Sequence_Coordinates_Predictions

In this part, we adapt an encoder-decoder system with an LSTM based model.
The model consider selected features in previous 4 strokes of a rally as inputs, the target output for training is a future sequence of a future stroke, so the approach is using the given past 4 strokes, we can produce the 5th stroke, with the 2nd, 3rd,4th and 5th stroke data(incorporating the future predictions as features), we predict the 6th strokes, and the prediction stop when the ideal ball round is reached as what the requirement in the data specified.

The code consists of several steps:

1. Grouping the data and generating sequences and futures: The training data is grouped by the set, match_id, and rally_id. This grouping ensures that the sequences observed by the model are from the same rally. The code here also defines the sequence length (4) and future length (1). It iterates over the grouped data and creates sequences of past information (past 4 strokes) and future information (to predict the next stroke). The sequences and corresponding futures are stored in separate lists.



```python
#group the data by the same rally for we need to make sure the sequence our model observe is on the same rally
grouped_data = train.groupby(['set', 'match_id', 'rally_id'])
```

```python
#generate past information(past 4 strokes) and future information(we need to predict future stroke)
sequence_length = 4
future_length = 1
sequences = []
futures = []
for _, group in grouped_data:
    rally_data = group[[ 'landing_height', 'landing_area',
            'player_location_x', 'player_location_y',
            'opponent_location_x', 'opponent_location_y',
            'landing_x', 'landing_y']]  # Select the relevant features for the input

    num_samples = len(rally_data) - sequence_length + 1 - future_length
    for i in range(num_samples):
        sequence = rally_data.iloc[i:i+sequence_length]
        sequences.append(sequence)
        future = rally_data.iloc[i+sequence_length:i+sequence_length+1]
        futures.append(future)
```

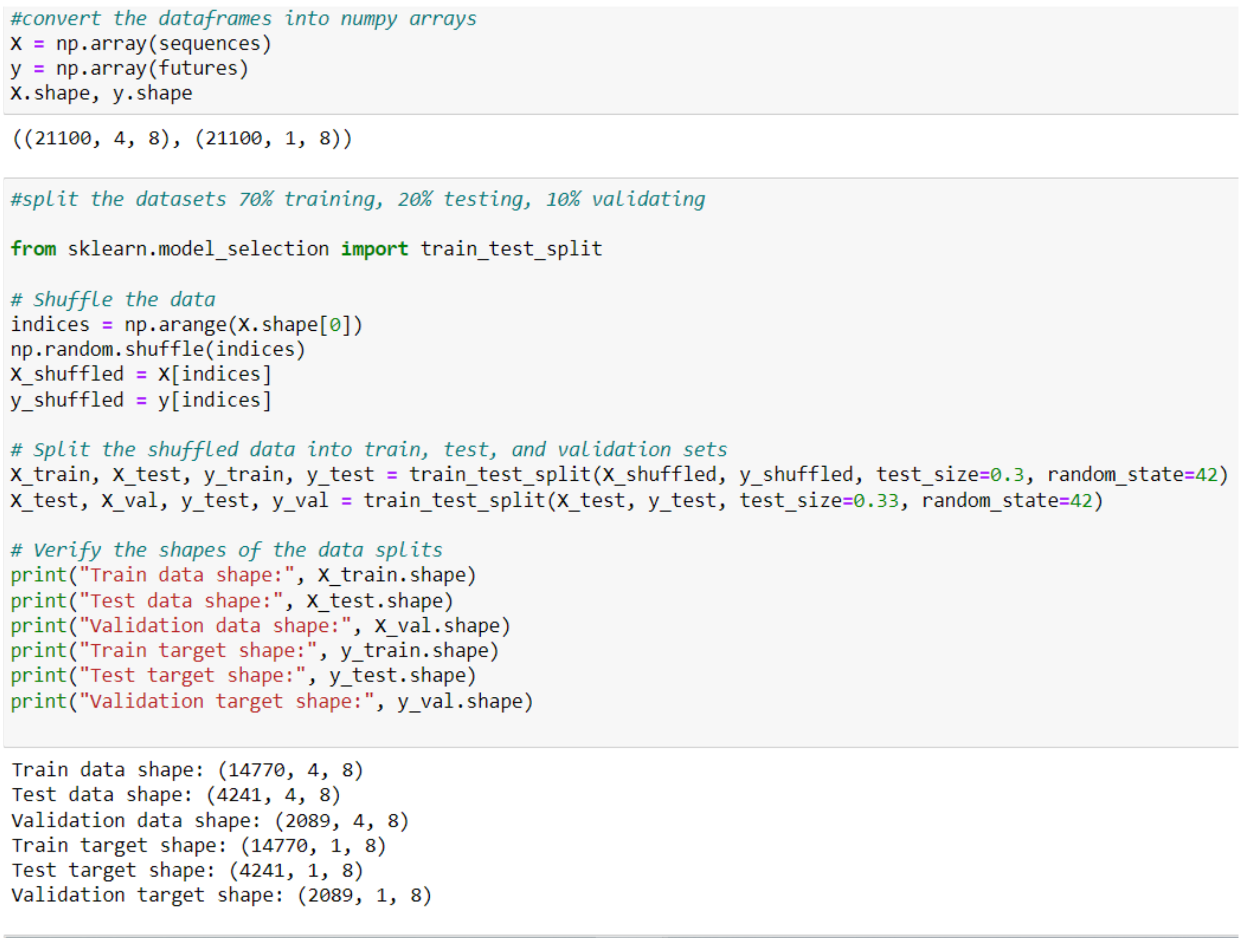
```python
sequences[2]
```

	landing_height	landing_area	player_location_x	player_location_y	opponent_location_x	opponent_location_y	landing_x	landing_y
2	-1.0	-0.555556	-0.034992	-0.276612	0.068671	0.366045	0.394393	0.839262
3	-1.0	-0.333333	0.615863	0.966302	0.063205	-0.508528	0.381856	-0.730463
4	1.0	-1.000000	0.737170	-0.594149	0.431500	0.685900	0.418771	0.186042
5	1.0	-0.777778	0.630638	0.286940	0.482064	-0.627080	0.364444	-0.110236

```python
futures[100]
```

	landing_height	landing_area	player_location_x	player_location_y	opponent_location_x	opponent_location_y	landing_x	landing_y
156	1.0	0.333333	0.190513	0.26787	0.178681	-0.482466	0.093853	0.010962

2. Converting to numpy arrays and splitting the datasets: The sequences and futures are converted from dataframes to numpy arrays. The data is split into training, testing, and validation sets. The code shuffles the data and then splits it into 70% training, 20% testing, and 10% validation using the `train_test_split` function from scikit-learn.



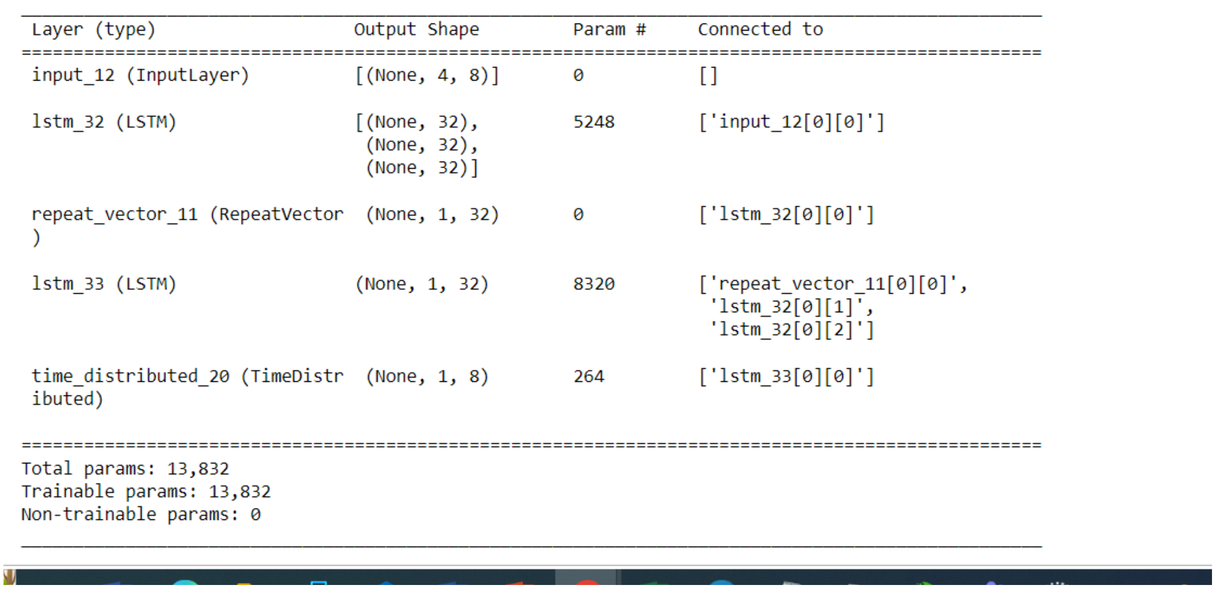
```python
#convert the dataframes into numpy arrays
X = np.array(sequences)
y = np.array(futures)
X.shape, y.shape
```

```
((21100, 4, 8), (21100, 1, 8))
```

```python
#split the datasets 70% training, 20% testing, 10% validating

from sklearn.model_selection import train_test_split

# Shuffle the data
indices = np.arange(X.shape[0])
np.random.shuffle(indices)
X_shuffled = X[indices]
y_shuffled = y[indices]

# Split the shuffled data into train, test, and validation sets
X_train, X_test, y_train, y_test = train_test_split(X_shuffled, y_shuffled, test_size=0.3, random_state=42)
X_test, X_val, y_test, y_val = train_test_split(X_test, y_test, test_size=0.33, random_state=42)

# Verify the shapes of the data splits
print("Train data shape:", X_train.shape)
print("Test data shape:", X_test.shape)
print("Validation data shape:", X_val.shape)
print("Train target shape:", y_train.shape)
print("Test target shape:", y_test.shape)
print("Validation target shape:", y_val.shape)
```

```
Train data shape: (14770, 4, 8)
Test data shape: (4241, 4, 8)
Validation data shape: (2089, 4, 8)
Train target shape: (14770, 1, 8)
Test target shape: (4241, 1, 8)
Validation target shape: (2089, 1, 8)
```

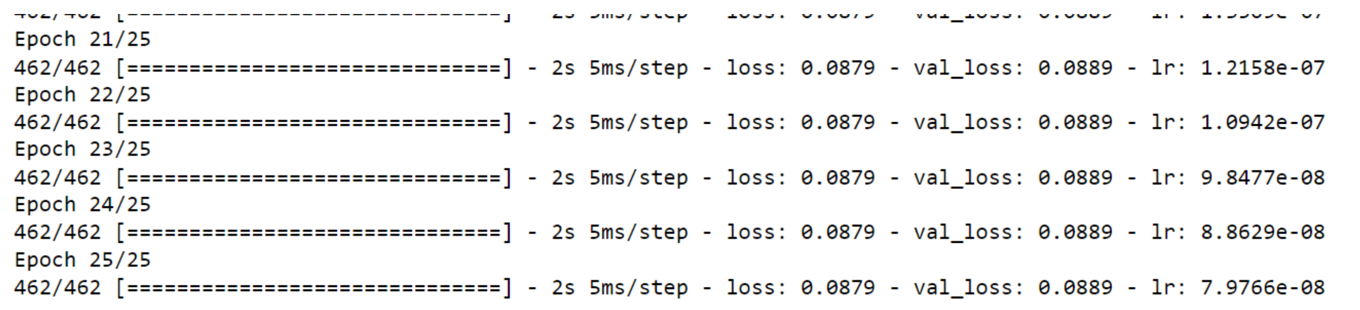
3. Building the E1D1 model: The E1D1 model is an encoder-decoder system used for sequence prediction. The input layer of the model takes the past sequences as input. The encoder LSTM layer processes the input sequences and returns the encoder states. The decoder inputs are created by repeating the last output of the encoder. The decoder LSTM layer takes the decoder inputs and the encoder states as initial states and produces the decoder outputs. Finally, the decoder outputs are passed through a dense layer to generate the predicted future sequences. In summary, the model has a total of 13,832 trainable parameters. It takes sequences of length 4 with 8 features as input, processes them through an encoder LSTM layer, repeats the output vector, and then decodes it using another LSTM layer. Finally, the dense layer is applied to each time step of the decoder outputs to generate the predictions for the future stroke.
Below is the summary of the E1D1 model:



```
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_12 (InputLayer)          [(None, 4, 8)]       0           []

 lstm_32 (LSTM)                 [(None, 32),         5248        ['input_12[0][0]']
                                 (None, 32),
                                 (None, 32)]

 repeat_vector_11 (RepeatVector  (None, 1, 32)       0           ['lstm_32[0][0]']
 )

 lstm_33 (LSTM)                 (None, 1, 32)        8320        ['repeat_vector_11[0][0]',
                                                                  'lstm_32[0][1]',
                                                                  'lstm_32[0][2]']

 time_distributed_20 (TimeDistr  (None, 1, 8)        264         ['lstm_33[0][0]']
 ibuted)

==================================================================================================
Total params: 13,832
Trainable params: 13,832
Non-trainable params: 0
__________________________________________________________________________________________________
```

6. Compiling and training the model: The model is compiled with the Adam optimizer and the Huber loss function. The choice of Huber loss is suggested by the referenced LSTM work mentioned earlier, incorporating both MSE and MAE, the loss function is more robust to outliers, which may be a great choice for real world dirty data. The Optimizer is Adam and we started with a learning rate of 1e-6, then, the model incorporates a reduced learning rate for every epoch, in which the learning rate is 0.9 the previous. A learning rate of 1e-6 can be considered very small for a starting point, however, this is the adjustment in our training process. During our training, we found out that the model suffered seriously from gradient exploding, resulting in the loss containing nan value and the model being inconsistent. We had a hard time dealing with the tuning of the hyper parameters. We start by setting the batch size as 1 and learning rate as 1e-9, then, we gradually increase the parameters while slightly modifying the layers in our model. At first, the unit is set to 100, however, we found a too complicated computations for complicated data results in a very early gradient exploding, so the final units are tuned down to 32. Another approach here is by incorporating the clipnum parameters, which also restrict the model from possibly nan losses with gradient issues. Finally, when the batch size is 32, with a reduced learning rate starting from 1e-6 and clipnum=1, the model can compile and fit the data successfully.

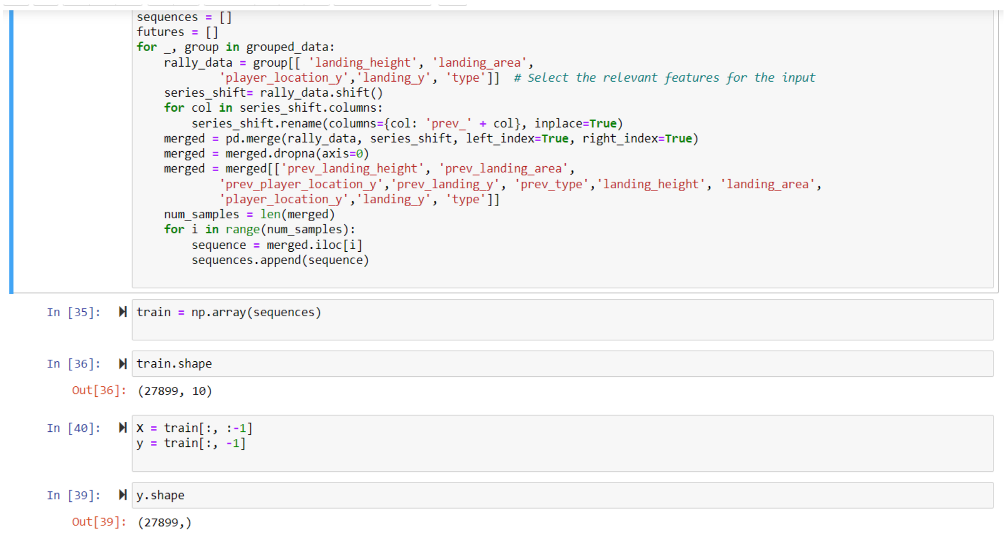
After training, the model reaches approximately 0.0879 to 0.0889 in training and validating loss.



```
Epoch 21/25
462/462 [==============================] - 2s 5ms/step - loss: 0.0879 - val_loss: 0.0889 - lr: 1.2158e-07
Epoch 22/25
462/462 [==============================] - 2s 5ms/step - loss: 0.0879 - val_loss: 0.0889 - lr: 1.0942e-07
Epoch 23/25
462/462 [==============================] - 2s 5ms/step - loss: 0.0879 - val_loss: 0.0889 - lr: 9.8477e-08
Epoch 24/25
462/462 [==============================] - 2s 5ms/step - loss: 0.0879 - val_loss: 0.0889 - lr: 8.8629e-08
Epoch 25/25
462/462 [==============================] - 2s 5ms/step - loss: 0.0879 - val_loss: 0.0889 - lr: 7.9766e-08
```

Stroke_Type_Classification

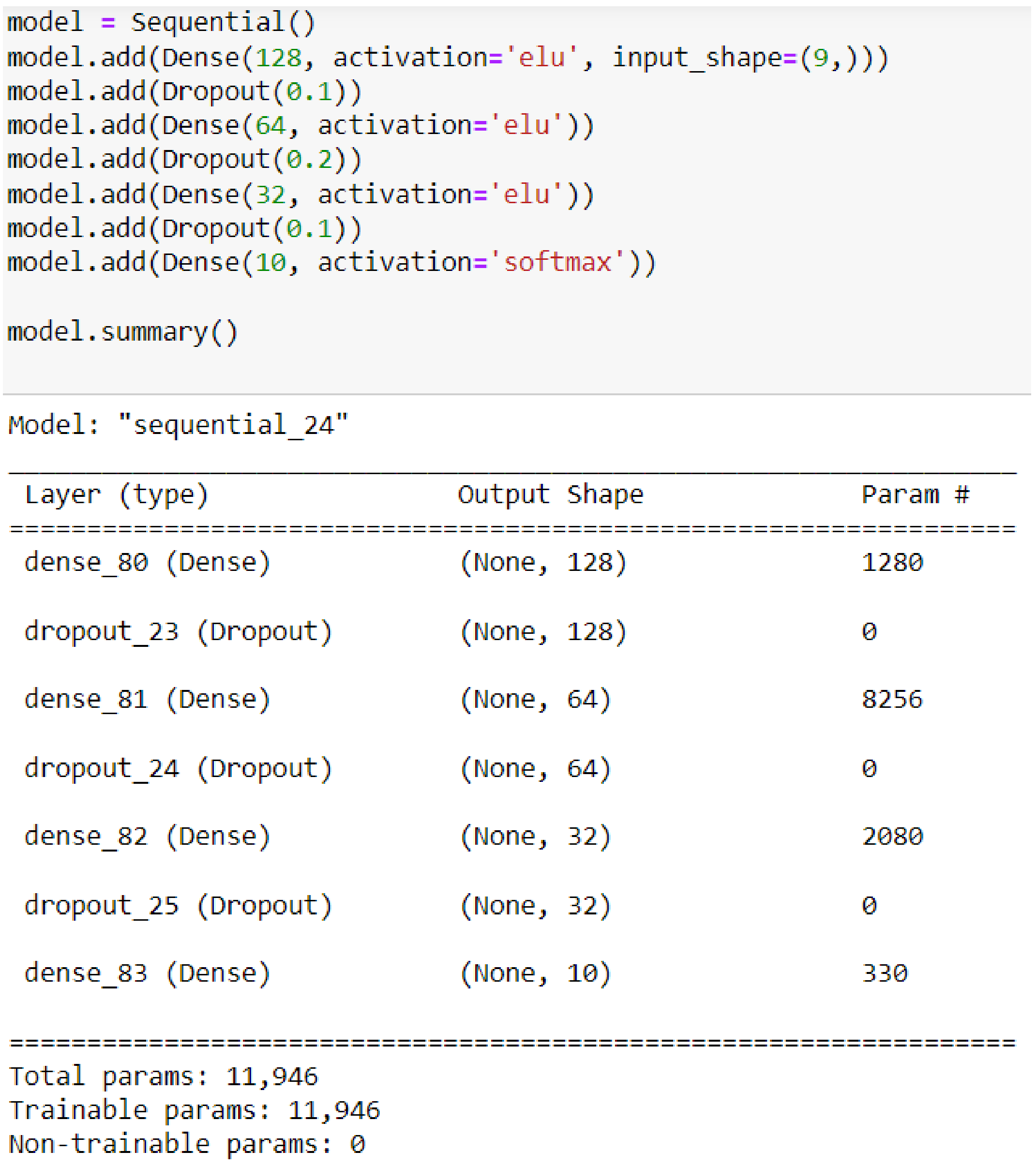
In this part, An easy classification model using a neural network to predict possible types of stroke is the main approach.
The datasets contain features as mentioned above, the selected current stroke feature and its lagged feature(the previous stroke), making the prediction as a direct classification problem containing the feature in both horizontal and vertical aspect, the first row of every stroke is dropped for it does not contain previous stroke.
For the datasets, 27899 train data rows are generated with 10 features relevant to the current and previous 1 sequence. The target label is the 10 types of stroke for the **current** stroke.



```
sequences = []
futures = []
for _, group in grouped_data:
    rally_data = group[[ 'landing_height', 'landing_area',
            'player_location_y','landing_y', 'type']]  # Select the relevant features for the input
    series_shift= rally_data.shift()
    for col in series_shift.columns:
        series_shift.rename(columns={col: 'prev_' + col}, inplace=True)
    merged = pd.merge(rally_data, series_shift, left_index=True, right_index=True)
    merged = merged.dropna(axis=0)
    merged = merged[['prev_landing_height', 'prev_landing_area',
            'prev_player_location_y','prev_landing_y', 'prev_type','landing_height', 'landing_area',
            'player_location_y','landing_y', 'type']]
    num_samples = len(merged)
    for i in range(num_samples):
        sequence = merged.iloc[i]
        sequences.append(sequence)
```

```
In [35]:  train = np.array(sequences)

In [36]:  train.shape
 Out[36]: (27899, 10)

In [40]:  X = train[:, :-1]
          y = train[:, -1]

In [39]:  y.shape
 Out[39]: (27899,)
```

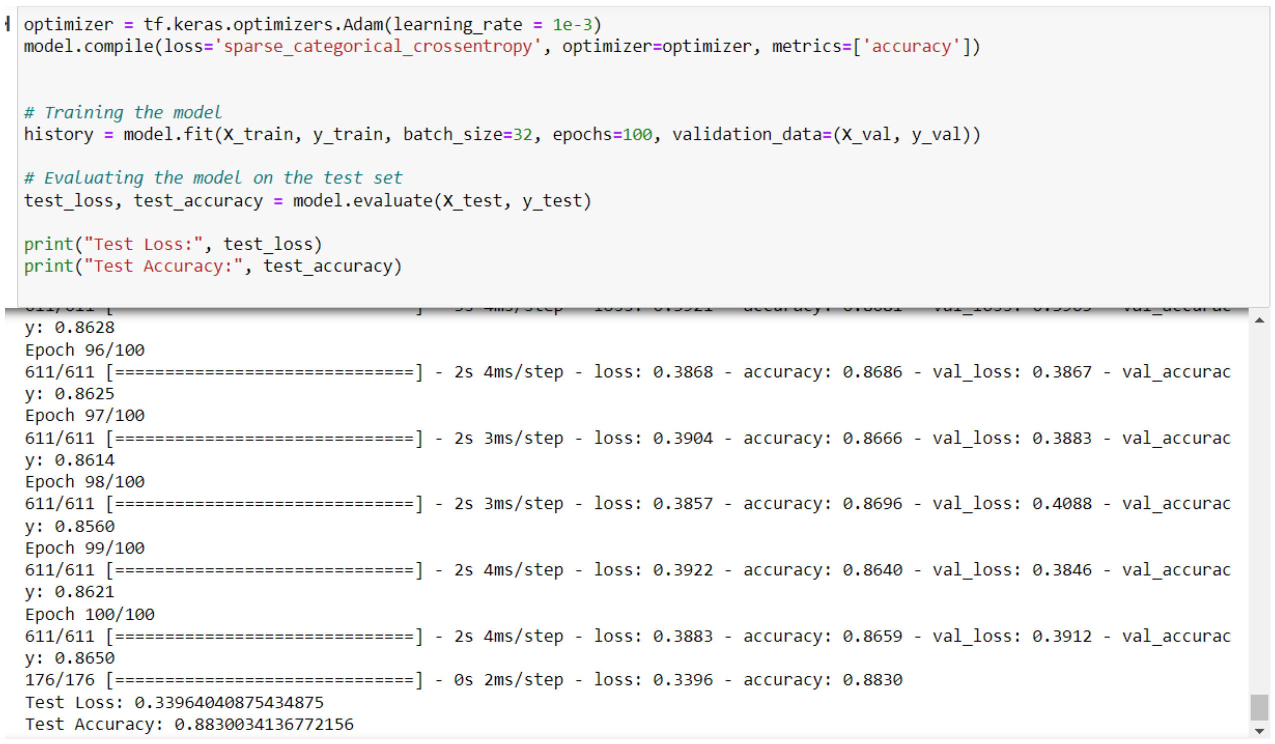
For the model's architecture, it contains 3 dense layers, each containing 128, 64 and 32 respectively. I don't want the network to be too complicated because I think the correlation is already pretty obvious.
After each Dense layer, there is a Dropout layer. At first, we did not add the drop out layer and the validation loss is a bit poor that indicates signs of overfitting, so we decided to add Dropout layers and tuned the number and proportion, finally, three dropout layers are added.
The final activation is Softmax, and dense it to 10 labels representing the 10 types of stroke we want to predict.
For the activation, we use "elu" instead of "relu" because we found relu results in dying due to negative outputs in some times. An the elu performs better than leaky relu, which is another alternative, so we decided to stick to elu.



```python
model = Sequential()
model.add(Dense(128, activation='elu', input_shape=(9,)))
model.add(Dropout(0.1))
model.add(Dense(64, activation='elu'))
model.add(Dropout(0.2))
model.add(Dense(32, activation='elu'))
model.add(Dropout(0.1))
model.add(Dense(10, activation='softmax'))

model.summary()
```

```
Model: "sequential_24"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 dense_80 (Dense)            (None, 128)               1280      
                                                                 
 dropout_23 (Dropout)        (None, 128)               0         
                                                                 
 dense_81 (Dense)            (None, 64)                8256      
                                                                 
 dropout_24 (Dropout)        (None, 64)                0         
                                                                 
 dense_82 (Dense)            (None, 32)                2080      
                                                                 
 dropout_25 (Dropout)        (None, 32)                0         
                                                                 
 dense_83 (Dense)            (None, 10)                330       
                                                                 
=================================================================
Total params: 11,946
Trainable params: 11,946
Non-trainable params: 0
```

We also use Adam as the optimizer here, and we stick to a learning rate of 0.001.
The loss function here is sparse_categorical_crossentropy, for we want to predict categorical data.
After we use the batch size of 32 and run for 100 epochs, the accuracy is around 86% to 89%, and the testing accuracy is a desirable 88.3%.



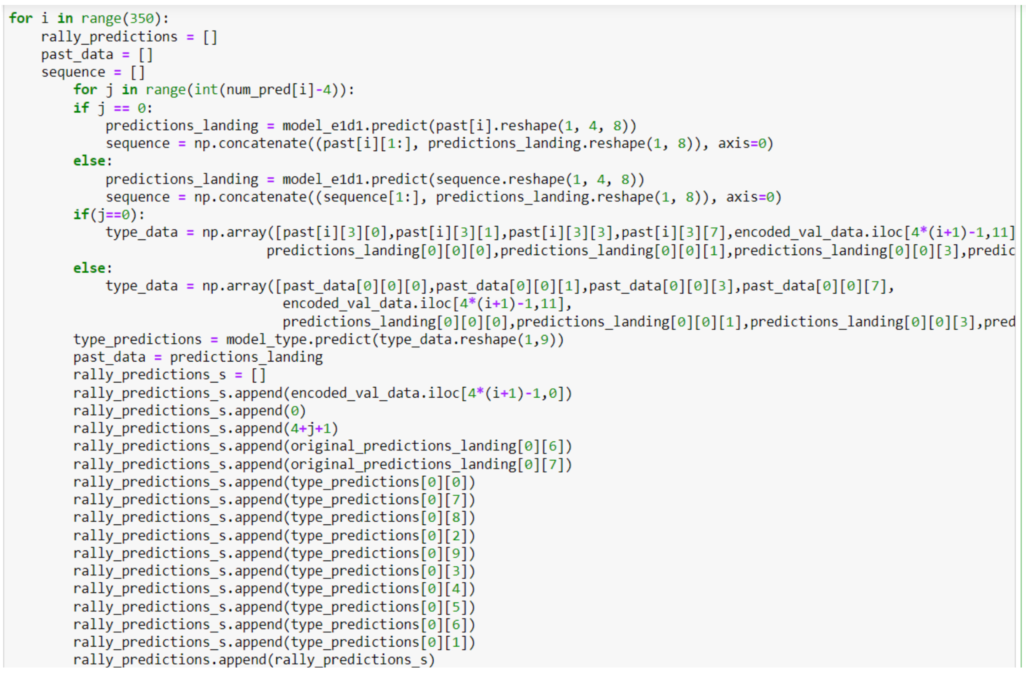
```
optimizer = tf.keras.optimizers.Adam(learning_rate = 1e-3)
model.compile(loss='sparse_categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])


# Training the model
history = model.fit(X_train, y_train, batch_size=32, epochs=100, validation_data=(X_val, y_val))

# Evaluating the model on the test set
test_loss, test_accuracy = model.evaluate(X_test, y_test)

print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
```

```
y: 0.8628
Epoch 96/100
611/611 [==============================] - 2s 4ms/step - loss: 0.3868 - accuracy: 0.8686 - val_loss: 0.3867 - val_accurac
y: 0.8625
Epoch 97/100
611/611 [==============================] - 2s 3ms/step - loss: 0.3904 - accuracy: 0.8666 - val_loss: 0.3883 - val_accurac
y: 0.8614
Epoch 98/100
611/611 [==============================] - 2s 3ms/step - loss: 0.3857 - accuracy: 0.8696 - val_loss: 0.4088 - val_accurac
y: 0.8560
Epoch 99/100
611/611 [==============================] - 2s 4ms/step - loss: 0.3922 - accuracy: 0.8640 - val_loss: 0.3846 - val_accurac
y: 0.8621
Epoch 100/100
611/611 [==============================] - 2s 4ms/step - loss: 0.3883 - accuracy: 0.8659 - val_loss: 0.3912 - val_accurac
y: 0.8650
176/176 [==============================] - 0s 2ms/step - loss: 0.3396 - accuracy: 0.8830
Test Loss: 0.33964040875434875
Test Accuracy: 0.8830034136772156
```

Final_Inference

In the final inference, the generation of the prediction is explained as below.

1. Prepare the datasets: Here, we recreated the same data form as in the model, we scaled the data using MinMaxScaling, encoded the"type" feature, and generated selected past 4 sequences as what we did in the first model.
2. Then, we gone through a huge loop that basically conclude all the generating of the prediction.



```
for i in range(350):
    rally_predictions = []
    past_data = []
    sequence = []
        for j in range(int(num_pred[i]-4)):
        if j == 0:
            predictions_landing = model_e1d1.predict(past[i].reshape(1, 4, 8))
            sequence = np.concatenate((past[i][1:], predictions_landing.reshape(1, 8)), axis=0)
        else:
            predictions_landing = model_e1d1.predict(sequence.reshape(1, 4, 8))
            sequence = np.concatenate((sequence[1:], predictions_landing.reshape(1, 8)), axis=0)
        if(j==0):
            type_data = np.array([past[i][3][0],past[i][3][1],past[i][3][3],past[i][3][7],encoded_val_data.iloc[4*(i+1)-1,11]
                                predictions_landing[0][0][0],predictions_landing[0][0][1],predictions_landing[0][0][3],predic
        else:
            type_data = np.array([past_data[0][0][0],past_data[0][0][1],past_data[0][0][3],past_data[0][0][7],
                                  encoded_val_data.iloc[4*(i+1)-1,11],
                                  predictions_landing[0][0][0],predictions_landing[0][0][1],predictions_landing[0][0][3],pred
        type_predictions = model_type.predict(type_data.reshape(1,9))
        past_data = predictions_landing
        rally_predictions_s = []
        rally_predictions_s.append(encoded_val_data.iloc[4*(i+1)-1,0])
        rally_predictions_s.append(0)
        rally_predictions_s.append(4+j+1)
        rally_predictions_s.append(original_predictions_landing[0][6])
        rally_predictions_s.append(original_predictions_landing[0][7])
        rally_predictions_s.append(type_predictions[0][0])
        rally_predictions_s.append(type_predictions[0][7])
        rally_predictions_s.append(type_predictions[0][8])
        rally_predictions_s.append(type_predictions[0][2])
        rally_predictions_s.append(type_predictions[0][9])
        rally_predictions_s.append(type_predictions[0][3])
        rally_predictions_s.append(type_predictions[0][4])
        rally_predictions_s.append(type_predictions[0][5])
        rally_predictions_s.append(type_predictions[0][6])
        rally_predictions_s.append(type_predictions[0][1])
        rally_predictions.append(rally_predictions_s)
```

First, there are 1400 pairs in total, which means there are 350 past sequence data, 350 rallies for predictions.
For each , start with the given 4 sequence to predict the 5th sequence, an inverse_transform is used here to obtain the landing coordinates, then the process continues for further stroke.
At the same time, after each sequence is predicted, we generate the type predictions probabilities with the stroke type classification model, the process continues until every round of every rally is done, then we write it in the csv.

Conclusion and Future Improvement

https://github.com/chiou1203/Final_Project_28

Unfortunately, due to poor communication and heavy loading, I did not finish this project well and I forgot the due date for phase 2 so I did not upload the csv in time, but since I know that my score may be poor, I also provide the csv in the github link. The report is also written although it still missing many things. For future improvement, We are looking forward to building a all-in-one model, the model contains a encoder-decoder system while each loss function is on a different function, resulting in a continuously generatioin of future model concerning all the features together.