INDIAN INSTITUTE OF TECHNOLOGY

MANDI

HIMACHAL PRADESH- 175005

CS-308

LARGE APPLICATION PRACTICUM

APPLICATION OF LSTM FOR CAPTURING SEQUENTIAL FEATURES

GROUP 4

1. ARYANSH SINGLA (B20085)
2. MAYANK BANSAL (B20156)
3. PALLAV VARSHNEY (B20160)
4. SHUBHAM SHUKLA (B20168)
5. SARTHAK JHA (B20317)
6. VASTAV BANSAL (B20325)

UNDER THE GUIDANCE OF MR. AADHAR GUPTA (MENTOR)

CONTENTS

ACKNOWLEDGEMENT

We have put a lot of effort into this project but none of them would bore fruit if it were not for the kind support and help of many individuals. We would like to extend our sincere thanks to all of them.
We would like to express our special gratitude to Dr. Varun Dutt, the course instructor who gave us the golden opportunity to work on this wonderful project filled with learnings of building an LSTM model for prediction in search & retrieval unity game.

We also want to express our heartfelt thanks to Mr Aadhar Gupta, our mentor for the project for his insight, guidance and elder brother support throughout the project. Also, a special thanks to all those who have contributed in some way or the other in the creation of this platform. We shall always be grateful to all of them.

ABSTRACT

LSTM or Long short-term memory is an artificial neural network used in the fields of artificial intelligence and deep learning. Unlike standard feedforward neural networks, LSTM has feedback connections. Such a recurrent neural network (RNN) can process not only single data points (such as images), but also entire sequences of data. Using LSTM, time series forecasting models can predict future values based on previous, sequential data. This provides greater accuracy for demand forecasters which results in better decision making for the business.

In this project, we have built a tool for a search and retrieval-based unity game, where we have applied LSTM on last n vectors to capture sequential information on current situations. The data is available to us in the form of features containing information about images at a given timestamp. The project uses Python 3 to build the models and predict information for new data.

We have used Keras which is an open-source software library which helps build artificial neural networks and used Google Colab for working. We applied various data preprocessing and normalization techniques to improve the accuracy of our predictions. In the end, we performed various modifications to the model and evaluation of the predictions to get the best results on the predictions.

Our project in the end is able to predict the situation at the current instance with good accuracy.

I. INTRODUCTION

We are training a robot for a game developed on unity where basically we have to predict its next move on the basis of the last n situations which are already given to us. For this, we have built a model on **Long short-term memory** (LSTM) neural networks.

Short Descriptions of technology stack:

Neural networks are a series of algorithms that endeavour to recognize underlying relationships in a set of data through a process that mimics the way the human brain operates. In this sense, neural networks refer to systems of neurons, either organic or artificial in nature. .

Let's get introduced to a neural network type viz. Recurrent Neural Network. Recurrent Neural Network (RNN) allows you to model memory units to persist data and model short-term dependencies. The detection of data correlations and patterns is also employed in time-series forecasting. By exhibiting behaviour comparable to that of the human brain, it also aids in producing predicting predictions for sequential data.

Now in our case, the data is basically sequential as we are taking a few last moments, moves or situations and that's why we are using RNN in our model. But to make sure that your RNN model performs well, RNN needs a lot of data which we are currently not having with us but still, we have enough data. So here we have moved to a more powerful type of RNN viz. Long Short Term Memory (LSTM) Network.

Now, Long Short Term Memory (LSTM) Network is an advanced RNN, a sequential network, that allows information to persist. It is capable of handling the vanishing gradient problem faced by RNNs. And it can also perform better in a short amount of data.
So we are going to use LSTM in our project.

More about LSTM

LSTM has a chain structure that contains four neural networks and different memory blocks called cells.
Information is retained by the cells and the memory manipulations are done by the gates.

1. **Forget Gate:** The forget gate purifies the information that is no longer relevant in the cell state. The gate receives two inputs, x_t (input at the current time) and h_{t-1} (prior cell output), which are multiplied with weight matrices before bias is added. The output of the activation function, which receives the outcome, is binary. If a cell state's output is 0, the piece of information is lost, however, if it is 1, the information is saved for use in the future.
2. **Input Gate:** The input gate updates the cell state with important details. To start, the inputs h_{t-1} and x_t are used to regulate the information using the sigmoid function and filter the values that need to be remembered similarly to the forget gate. Then, a vector containing every possible value between h_{t-1} and x_t is produced using the tanh function, which produces an output ranging from -1 to +1. In order to extract valuable information, the vector's values and the controlled values are finally multiplied.
3. **Output Gate:** The output gate's job is to take useful information out of the current cell state and deliver it as output. The tanh function is first used in

the cell to create a vector. The data is then filtered by the values to be remembered using the inputs h_{t-1} and x_t, and the information is then controlled using the sigmoid function. The vector's values and the controlled values are finally multiplied and supplied as input and output to the following cell, respectively.

II. Methodology

As in the project we have to build a model which has to predict the next move of the robot that we were training. There are three classes which basically indicate the movement i.e. 0, 1 and 2.

We have basically built our model using the LSTM neural network. For this we have used inbuilt utilities from Keras in TensorFlow.

Libraries used:

```
[ ]  import numpy as np
     import tensorflow as tf
     from tensorflow.keras import losses as lo_ss
     from tensorflow.keras.models import Sequential
     from tensorflow.keras.layers import LSTM
     from tensorflow.keras.layers import Dense, Dropout
     import pandas as pd
     from matplotlib import pyplot as plt
     from sklearn.preprocessing import StandardScaler
     import seaborn as sns
     from sklearn import model_selection

     from sklearn.datasets import make_classification
     from sklearn.utils import class_weight
     from sklearn.metrics import classification_report , roc_auc_score
```

Now we are given CSV data having a total of 155 columns having 1 output column, 150 features and 4 irrelevant columns (already known). So we have preprocessed data, standardized it and stored features in one and output in another variable.

```
# Making a new dataframe which contains only relevant features
df = df1[:,5:]
df = np.array(df)

# Standardizing data
from sklearn.preprocessing import StandardScaler
slr=StandardScaler()
slr.fit(df)
df = slr.transform(df)

# final dataframe
df=pd.DataFrame(df)
df.head()

# Storing output corresponding to the new dataframe and according to the timestamp
y=np.array(df1[:,0])
y=y[timestamp:]

y, y.shape
```

Now to train LSTM, the features we have are the last ‘t’ timestamp’s data at one time. So we have manipulated data in such a way that ‘t’ consecutive features (rows) are combined in one and then we have put those features in an array giving a 3-dimensional array of shape (no_of_rows, timestamp, 150).

```
# Making 3d array where each row contains last t rows information of the data
x = np.array(df)
# print(x.shape,y.shape)

x_3d = []
for i in range(4035-t):
  tmp = []
  for j in range(t):
    tmp.append(x[j+i])
  x_3d.append(tmp)
x_3d = np.array(x_3d)
x_3d.shape
```

```
(4030, 5, 150)
```

After that we split the data into train-test sets with an appropriate ratio, to train our data on the train set and then test it over the test set and find the accuracy of the prediction made.

```
# making test and train input and output data for the lstm layer
total_size = x_3d.shape[0]

# assigning train-test ratio to split data in train-test
train_test_ratio=0.8
train_size = int(total_size * train_test_ratio)
test_size = total_size - train_size
```

Now there are two ways we have assigned the data to train and test.

a) Sequentially assignment of data: Here basically we have assigned first train_size data to train data and left the last data to test data.

```
[ ] # Sequentially assignment of data : Taking the first data as train data and last one as test data
    x_train = x_3d[ : train_size]; x_test = x_3d[train_size : ]
    y_train = y[ : train_size]; y_test = y[train_size : ]
```

b) Random assignment of data: On the basis of a random number that we have generated between 0 to 100. We have assigned corresponding data to train (if the randomly generated number < 80) else we will assign it as test data.

```
# these array will contain test and train data
x_train, y_train, x_test, y_test = [], [], [], []

# Random assignment of data : filling the test and train data randomly
for i in range(x_3d.shape[0]):
  feature = []
  corr_output = []

  no = np.random.randint(100)

  if(no < 80):
    feature = x_3d[i]
    corr_output = y[i]

    x_train.append(feature)
    y_train.append(corr_output)

  else:
    feature = x_3d[i]
    corr_output = y[i]

    x_test.append(feature)
    y_test.append(corr_output)
```

Problem : We were getting the class prediction for class 0 to be biased as our model was predicting zero most of the time. We can see it in the confusion matrix as given below.

```
[ ] print(confusion_matrix(y_test_ans,y_pre_ans))

    [[395   1  57]
     [223   0  44]
     [250   0  39]]
```

Here we got introduced to the new concept of **class imbalance.**

Class imbalance: This is a scenario where the number of observations belonging to one class is significantly lower than those belonging to the other classes. In this situation, the predictive model developed using conventional algorithms could be biased and inaccurate.

Rare Event: The prediction for the class is less than 5%.

Problems

a) The features of the minority class are treated as noise and are often ignored.

b) High probability of misclassification of the minority class.

c) Total accuracy is not an appropriate measure to evaluate model performance.

Approaches to handle the imbalance data:

1. Normal changing the size of the class data: Either increasing the frequency of the minority class or decreasing the frequency of the majority class.
2. Random Under-Sampling: Randomly eliminating majority class examples. This is done until the majority and minority class instances are balanced out.

3. Random Over-Sampling: Increases the number of instances in the minority class by randomly replicating them in order to present a higher representation of the minority class in the sample.
4. Biasing the weights: Here we can provide the factors to bias the weights for the classes.
5. Cluster-based Oversampling: K-means clustering algorithm is independently applied to minority and majority class instances. Subsequently, each cluster is oversampled such that all clusters of the same class have an equal number of instances and all classes have the same size.
6. Synthetic Minority Over-sampling: A subset of data is taken from the minority class as an example and then new synthetic similar instances are created. These synthetic instances are then added to the original dataset.
7. XG-boost: More efficient implementation of Gradient Boosting Algorithm.

The other way could be to use many classifiers and then aggregate the predictions at the end. But we are already doing them by using neural networks.
In our model, we have implemented random Oversampling, Undersampling and biasing of the weights as our solution for the class imbalance problems.

Oversampling by repeating the data:

```
# Over-sampling with repeating just after
x_train_1, y_train_1 = [], []
c1, c2, c3 = 0, 0, 0

for i in range(main_x_train.shape[0]):
  if(y_train[i] == 0):
    x_train_1.append(main_x_train[i])
    y_train_1.append(0)
    c1 += 1
  elif(y_train[i] == 1):
    x_train_1.append(main_x_train[i]); x_train_1.append(main_x_train[i])
    y_train_1.append(1); y_train_1.append(1)
    c2 += 2
  elif(y_train[i] == 2 ):
    x_train_1.append(main_x_train[i]); x_train_1.append(main_x_train[i])
    y_train_1.append(2); y_train_1.append(2)
    c3 += 2

x_train = x_train_1; y_train = y_train_1
x_train = np.array(x_train); y_train = np.array(y_train)
```

Random Oversampling:

Handling Class imbalance

```
# Random Over-sampling
x_train_1, y_train_1 = [], []
c1, c2, c3 = 0, 0, 0

for i in range(main_x_train.shape[0]):
  if(y_train[i] == 0):
    x_train_1.append(main_x_train[i])
    y_train_1.append(0)
    c1+=1
  elif(y_train[i] == 1):
    x_train_1.append(main_x_train[i])
    y_train_1.append(1)
    c2+=1
  elif(y_train[i] == 2 ):
    x_train_1.append(main_x_train[i])
    y_train_1.append(2)
    c3=c3+1

for i in range(main_x_train.shape[0]):
  if(y_train[i] == 1):
    no = np.random.randint(100)
    if(no >= 30):
      x_train_1.append(main_x_train[i])
      y_train_1.append(1)
      c2 += 1
  elif(y_train[i] == 2 ):
    no = np.random.randint(100)
    if(no >= 27):
      x_train_1.append(main_x_train[i])
      y_train_1.append(2)
      c3 += 1
```

Random Undersampling:

```
# Random under-sampling

print("main_x_train : ", main_x_train.shape)

x_train_1, y_train_1 = [], []
c1, c2, c3 = 0, 0, 0

for i in range(main_x_train.shape[0]):
  if(main_y_train[i] == 0):
    to_take = np.random.randint(100)
    if(to_take > 50):
      x_train_1.append(main_x_train[i])
      y_train_1.append(0)
      c1 += 1
  elif(main_y_train[i] == 1):
    x_train_1.append(main_x_train[i]) #; x_train_1.append(main_x_train[i])
    y_train_1.append(1) #; y_train_1.append(1)
    c2 += 1
  elif(main_y_train[i] == 2 ):
    x_train_1.append(main_x_train[i]) #; x_train_1.append(main_x_train[i])
    y_train_1.append(2) #; y_train_1.append(2)
    c3 += 1

x_train = x_train_1; y_train = y_train_1
x_train = np.array(x_train); y_train = np.array(y_train)
```

Biasing the weights:

```
# Different weights for the classes to be assigned

sklearn_weights = class_weight.compute_class_weight(class_weight = 'balanced',classes = np.unique(y_train),y = y_train)
sklearn_weights = dict(zip(np.unique(y_train),sklearn_weights))

# manually assigning weights
# sklearn_weights={0:1738,1:2348,2:2362}

sklearn_weights
```

```
{0: 1.0218089990817263, 1: 1.0197021764032073, 2: 0.9609240069084629}
```

F-score: In the project, we have used the traditional balanced F-score or F_1-score which is calculated as the harmonic mean of precision and recall.

$$F1\ Score = 2 \times \frac{recall \times precision}{recall + precision}$$

Since F_1-score ignores the true negatives, it tends to be misleading in the case of unbalanced classes, and hence making all the classes balanced was important to get the correct evaluation of the F_1 score as we have already done it before so we were also getting quite good F_1 score.

Model making: We have worked on the Sequential model of LSTM networks. Here we have tried many different models where we have tried to change the number of neurons on layers, different combinations of layers (LSTM and Dense) and activation functions.

Finally, we have made the following model:

```
# Model 2
model = Sequential()
model.add(LSTM(200, activation='relu', return_sequences=True, input_shape=(t, 150,)))
model.add(LSTM(400, activation='relu', return_sequences=True))
model.add(LSTM(800, activation='relu', return_sequences=True))
model.add(LSTM(150, activation='relu'))
model.add(Dense(100, activation='relu'))
model.add(Dense(64, activation='relu'))

model.add(Dense(10, activation='relu'))
model.add(Dense(3,activation='sigmoid'))
optimi=keras.optimizers.Adam()

# added f1-score here
model.compile(optimizer=optimi, loss='sparse_categorical_crossentropy',metrics=['accuracy',get_f1])

# fitting model for different epochs
# without class weights
# history = model.fit(x_train, y_train, epochs=30, validation_split=0.2, verbose=1)

# # with class weights
history = model.fit(x_train, y_train, epochs=20, validation_split=0.2, verbose=1, class_weight=sklearn_weights)
test_output = model.predict(x_test, verbose=1)
```

On the basis of the above model, we have got our predictions.

```
from sklearn.metrics import confusion_matrix, accuracy_score

# printing confusion matrix and accuracy score
print("confusion matrix : ", confusion_matrix(y_test,test_output_3))
test_output = pd.DataFrame(test_output)
print("accuracy score : ", accuracy_score(y_test,test_output_3))
```

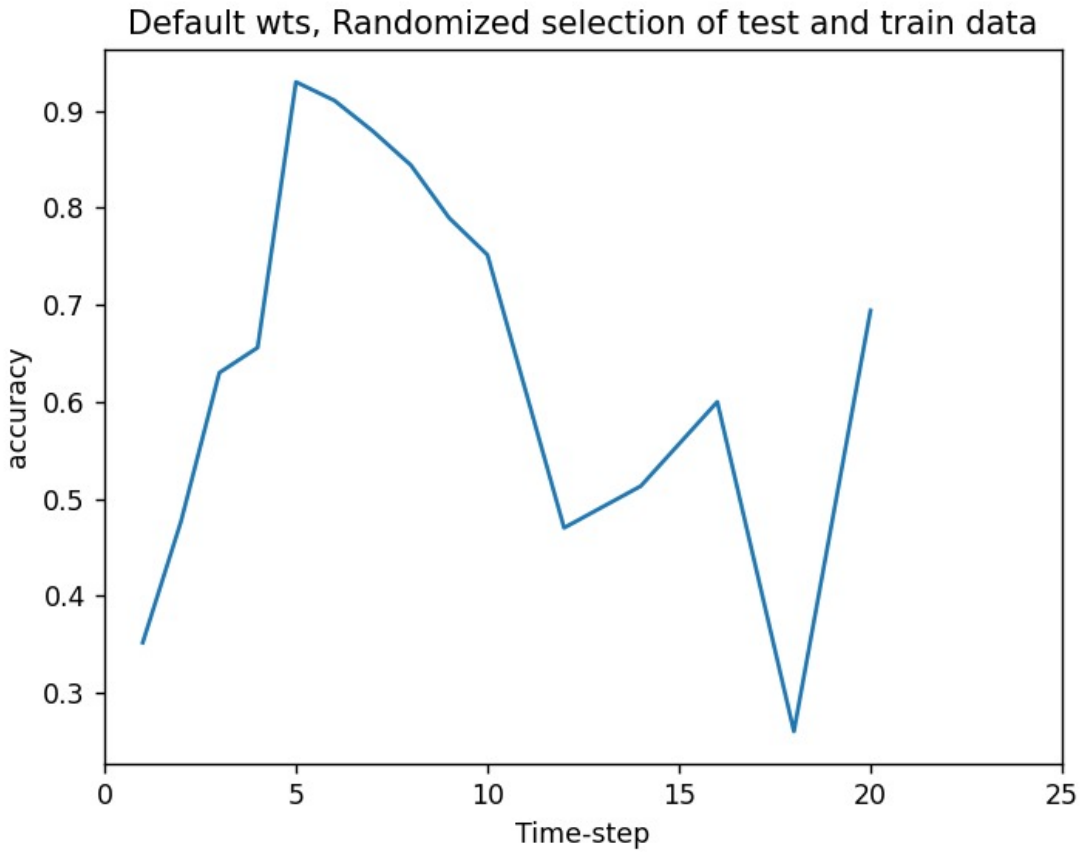
```
confusion matrix :  [[289  44  38]
 [ 63 126   4]
 [ 79  14 158]]
accuracy score :  0.7030674846625767
```

III. RESULTS / OUTPUTS

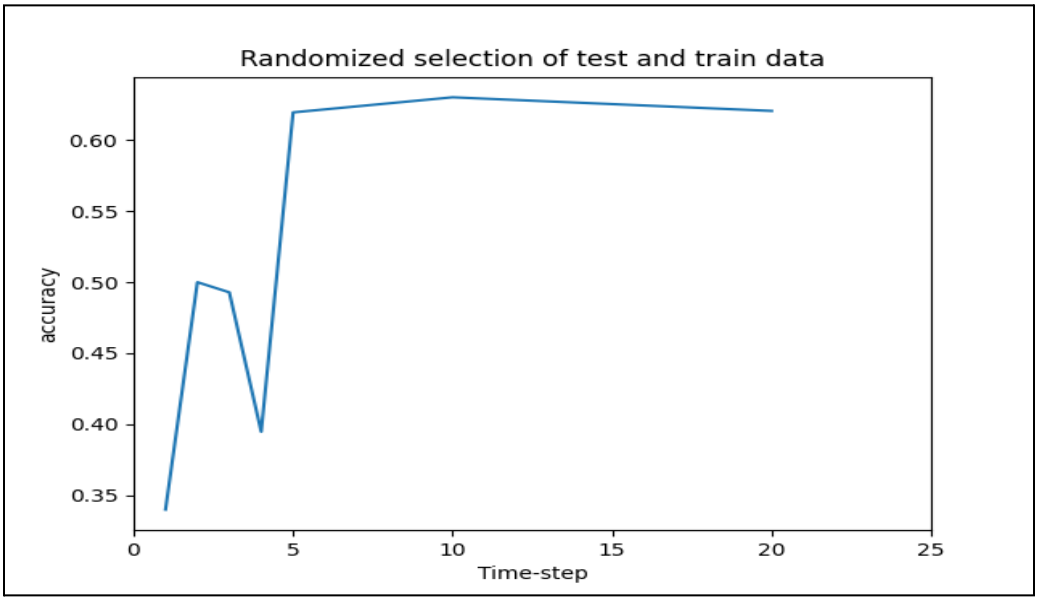
Outputs for different types of models we have tried: In the project, we have tried models in different cases like random oversampling, undersampling, biased weights, and balancing the frequency of all the classes.

The best accuracy for all of them is about equal as can be seen by the output graphs.

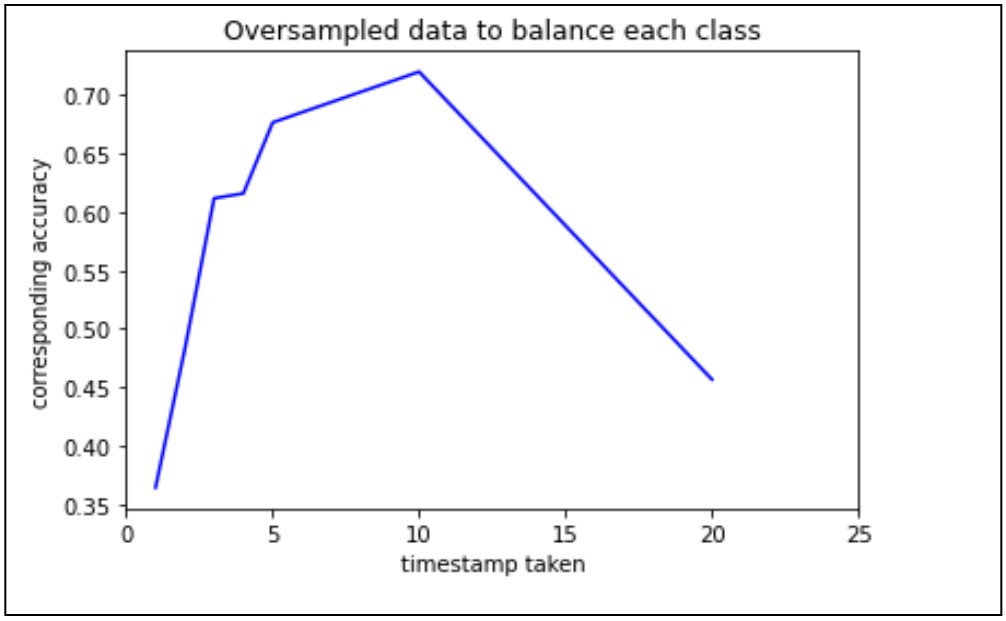
Output 1: Default weights of the classes and random selection of the data



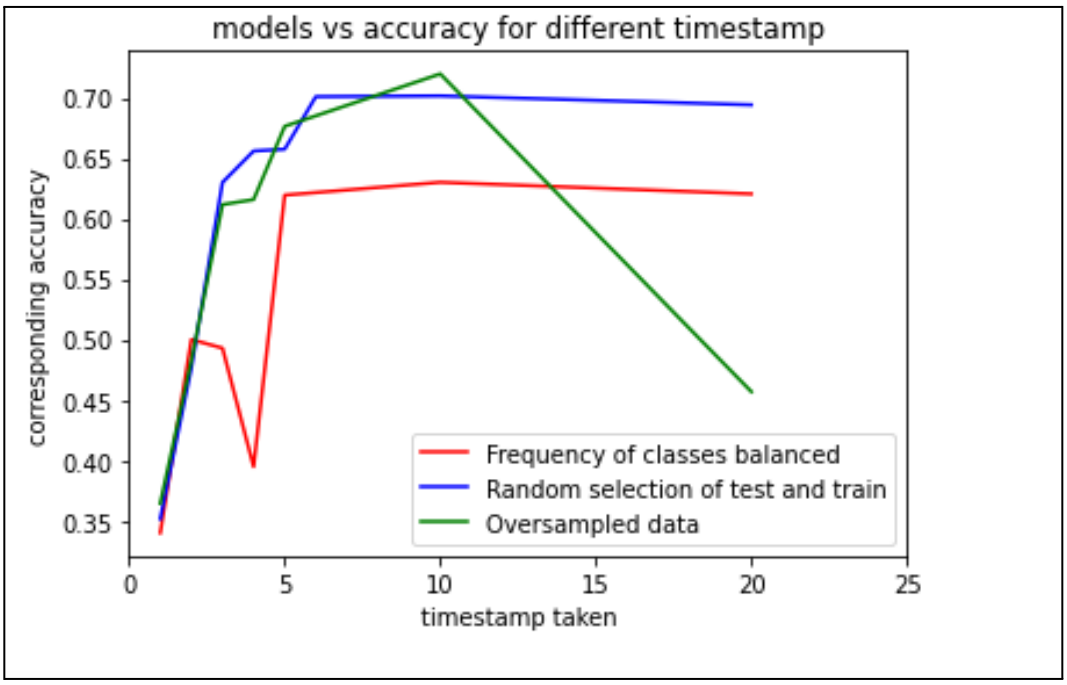
Output 2: Oversampling with Random selection of test and train data having an unbalanced frequency.



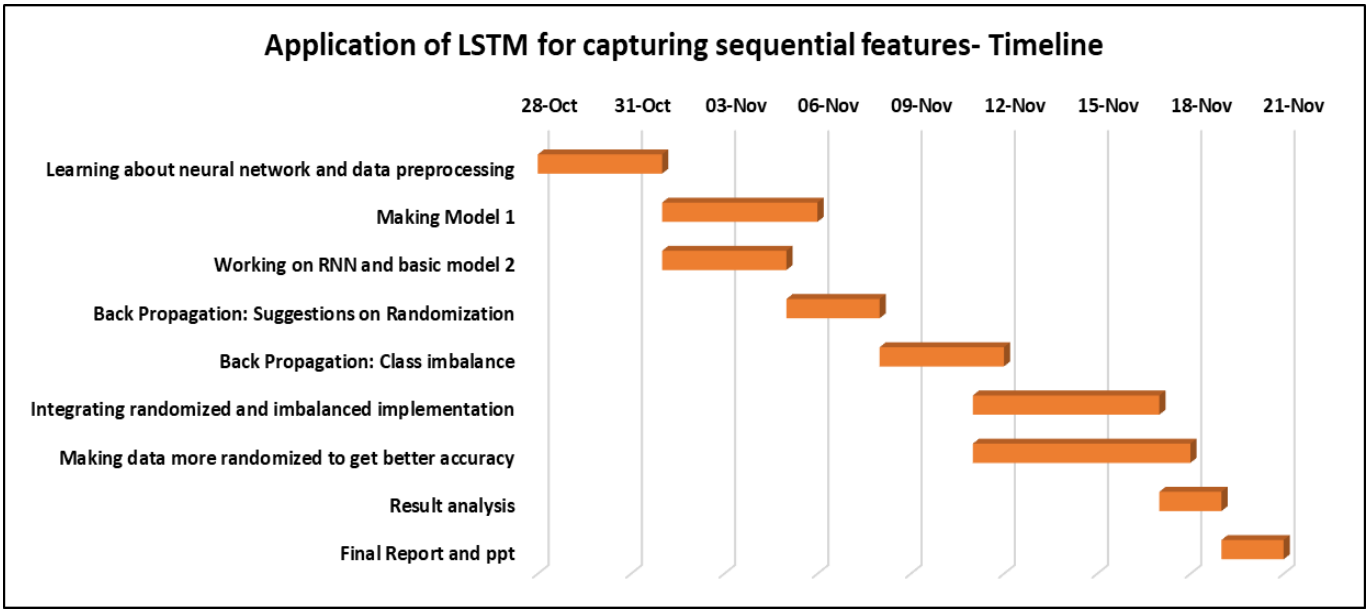
Output 3: Oversampling with Random selection of test and train data having a balanced frequency.



Finally combining all:



IV. GANTT CHART



V. FUTURE ENHANCEMENTS

Well, the main requirement is achieving a higher accuracy even, so we could even have tried more architectures as well.Usage of reinforcement learning could also help in attaining more accuracy as reinforcement learning is based on earning rewards the model itself uses the test data to further enhance the weights based on the the rewards available for the bot.

Since this model is more or less based on trial and error method more architectures could be tried and used to enhance the accuracy further.

But the most significant of all huge increases in data could also enable the bot to learn about different situations about how the bot will react in various other possible situations that couldn't be covered in the current training data.

CONCLUSION

In conclusion, the project ‘Application of LSTM for capturing sequential features’ to train an intelligent robot helped us learn and get familiar with some basic models and modern-day techniques of training robots.

In the project, we worked on training a robot to help it predict its next move between three viz. straight , left or right. From having a 0 accuracy in the beginning we reached to having an accuracy of more than 93%. We also learnt ways to deal with problems like class imbalance, applied weights, techniques like oversampling and undersampling and observed its results on the accuracy of prediction.

Besides we learnt about various loss functions and optimizers and which should be used based upon the situation. We processed the data and applied various techniques and models to reach a final accuracy of 93% and a f1 score of around 0.95. The models developed as a part of the project can be further refined using reinforcement learning techniques and can predict the next ‘correct move’ up to a great extent. Whole in whole we had a great time learning new techniques, and how to modify the data after learning and observing through the confusion matrix.