WESTERN UNIVERSITY

Comparative analysis of Deep Learning models for detecting frequencies in Noisy Sinusoidal Wave

Author:
Sahana Chakravarty

Supervisor:
Dr. Vijay PARSA

A thesis submitted in fulfilment of the requirements for the degree of Master of Engineering

in the

Software Engineering
Electrical and Computer Engineering

April 2022

Declaration of Authorship

I, Sahana Chakravarty, declare that this thesis titled, “Application of Deep Learning for detecting frequencies in Noisy Sinusoidal Wave” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a M. Eng. degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

_____

Date:

_____

Acknowledgements

I offer my sincerest gratitude to my supervisor, Dr. Vijay Parsa who gave me the opportunity of doing this research. His guidance accelerated the process and kept me on the right track.

My special thanks to Vahid A Chenarlogh, who helped me gain more depth on the project.

I am grateful to all my professors and the whole Western university for giving me this opportunity.

My sincere and heartfelt gratitude to my parents and my mentor, Shree Soumyarup Saha without whom this journey could not have commenced and fulfilled so gracefully.

WESTERN UNIVERSITY

Abstract

Engineering
Electrical and Computer Engineering

Master of Engineering

Training a Deep Learning model for detecting frequencies in Noisy Sinusoidal Signal

by Sahana Chakravarty

A sinusoidal wave characterizes an audio signal, such as music and speech, in terms of amplitude, its phase component and frequency. In digital form, this is essentially a large array of numbers that represents position at a specific time or its amplitude. The generated sinusoidal signal is then fed with additive white Gaussian noise (AWGN) to produce a noisy sin wave. 10000 samples consisting of 9 frequencies ranging from 250 – 6000 Hz were used as input.

A three-layer Neural Network (LSTM) was created to train, test and validate the model. Among bidirectional LSTM and LSTM, the former model gave us almost better accuracy with minimum loss.

Contents

Introduction

Many applications such as power systems, communications and radar applications had been facing problems in estimating frequency of a noisy audio signal. Although we are acquainted with many theoretical techniques such as Fourier transform, least square methods and phase lock loop but they were unable to determine the frequency correctly.[1]

Deep learning methods in recent years have been able to solve many challenging issues in medical diagnosis, image recognition and classification and speech recognition. In the field of signal processing, researchers have been able to separate out noise from spectrogram and frequency monitoring with real time data.[1]

In this project, a deep-learning algorithm had been used to see how we can train noisy sinusoidal waves that simulates audio signals to get a robust model that would be able to further classify frequencies. Since Neural Network is able to recognize underlying relationships in a large set of data, the resulting output is quite accurate and can detect patterns which human beings are unable to detect. Hence, using a trained Neural Network model would be able to find the frequency with speed and accuracy. [1]

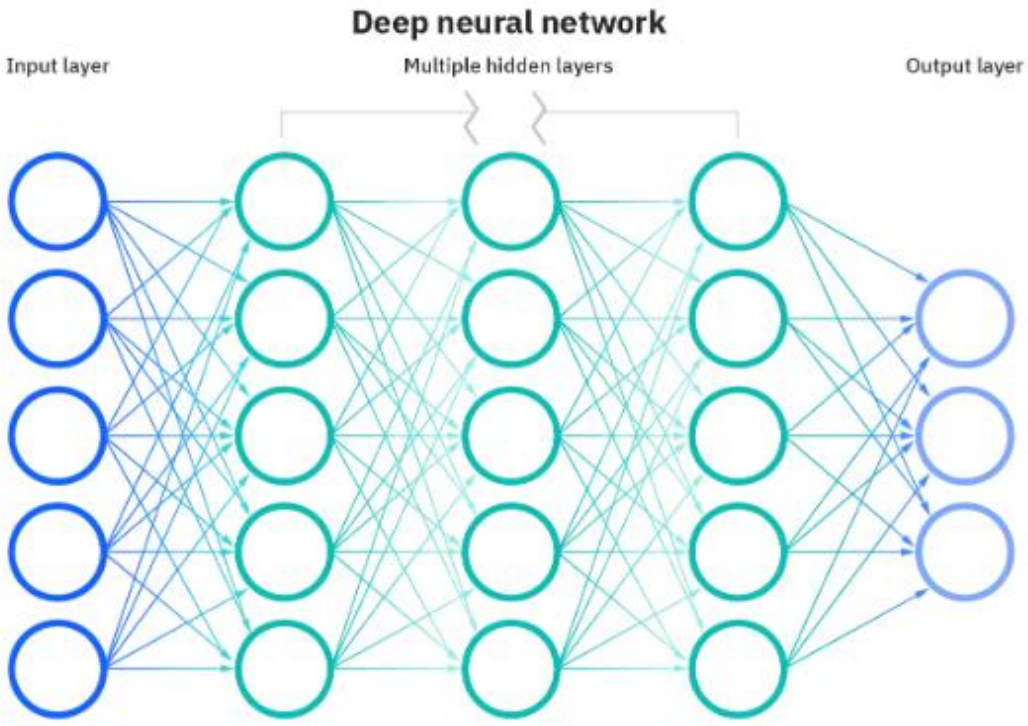


Figure 1: Neural Network outline [4]

Data Preparation

Neural networks rely on training data samples to learn and enhance their accuracy over time. Here, we used training and testing dataset in 80 20 ratios. The training data is used for training the model at each step while the testing dataset is used to test the model.

One could imagine the dataset being split as follows:

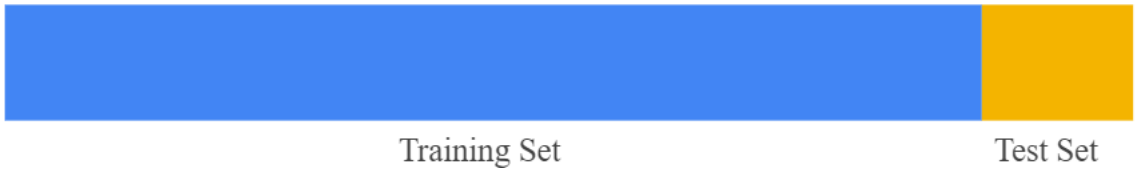


Figure 2: Slicing a single data set into a training set and test set [5]

We need to make sure these below two criterions are met by the test set:

- Is large enough to yield statistically meaningful results.
- Is representative of the data set as a whole. In other words, we should not pick a test set that has different features than the training set.
- Also, we should not train our model with test data.

The best model is found when we get a model that has lowest loss and good accuracy on training dataset and testing dataset.

A total of 90000 samples were prepared for the whole dataset and we used 80% of the waves (72000) for training dataset and 20% as the testing dataset (18000).

The sampling rate of 32 kHz was used in input. The frequency list contained frequencies such as 250, 500, 750, 1000, 1500, 2000, 3000, 4000, 6000 Hz respectively. For each wave, 9000 samples were considered with randomized phase and the value of alpha ranging between .1 to 5.

The whole dataset was constructed of $90{,}000 \times 2048$ array. This meant that the input layer of the neural network had 2048 nodes.

Ideally, while predicting with the validation set and to find the frequency of each wave, the output layer of the neural network would have 1 node that corresponds to the output frequency.

In our case, there are 90000 samples that are divided into 9 classes from 0 to 1 to return a binary matrix representation of the input using the function to_categorical. As Neural networks work well when their output is between 0 and 1, so, here the input data has been normalized by standard deviation and then split into 80 and 20 parts. These pre-processing steps are done to get the best model.

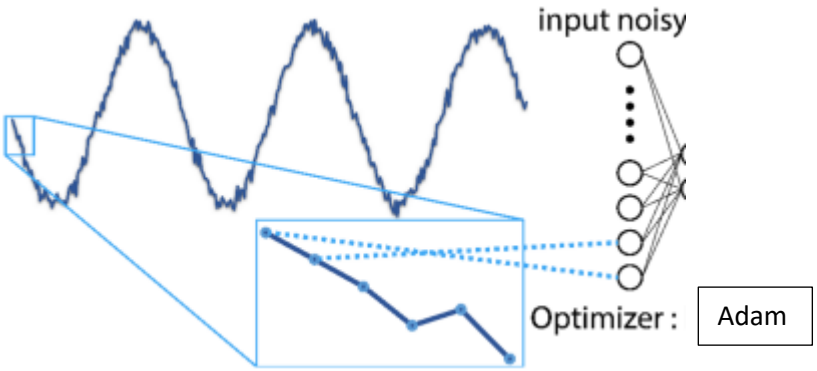


Figure 3: Schematic of the neural network (NN) model [1]

Optimisers play a very crucial role to increasing the accuracy of the model. It calculates learning rate for each parameter and compares against other adaptive learning algorithms. In this project, Adam optimizer has worked best among others and so, it was chosen.

Method

To define the problem, our input signal is represented as follows:

$S(t)=A\sin(2\pi ft + \phi) + \Omega(t)$ **[1]**

where A is amplitude, t is time, ϕ is phase offset, and Ω is zero-mean Gaussian noise with a variance of σ^2. So, the S(t) which is the noisy wave will be our input and the f which is the frequency will be our output. The signal-to-noise ratio (SNR), which shows the quality of the signal, is the ratio of signal power P_S to noise power P_N **[1]**.

$$SNR = \frac{P_S}{P_N} = \left(\frac{A}{\sigma}\right)^2$$

While in decibels it can be represented as:

$$SNR_{dB} = 10\log_{10}(SNR) = 10\log_{10}\left[\left(\frac{A}{\sigma}\right)^2\right]$$

That gives us

$$\sigma^2 = \frac{A^2}{10^{\frac{SNR_{dB}}{10}}}$$

So given SNR_{dB} and A we can obtain the variance that is needed for calculating the noise function **[1]**.

In this case, we have randomized the phase and the SNR thus producing noisy sinusoidal waves as follows:

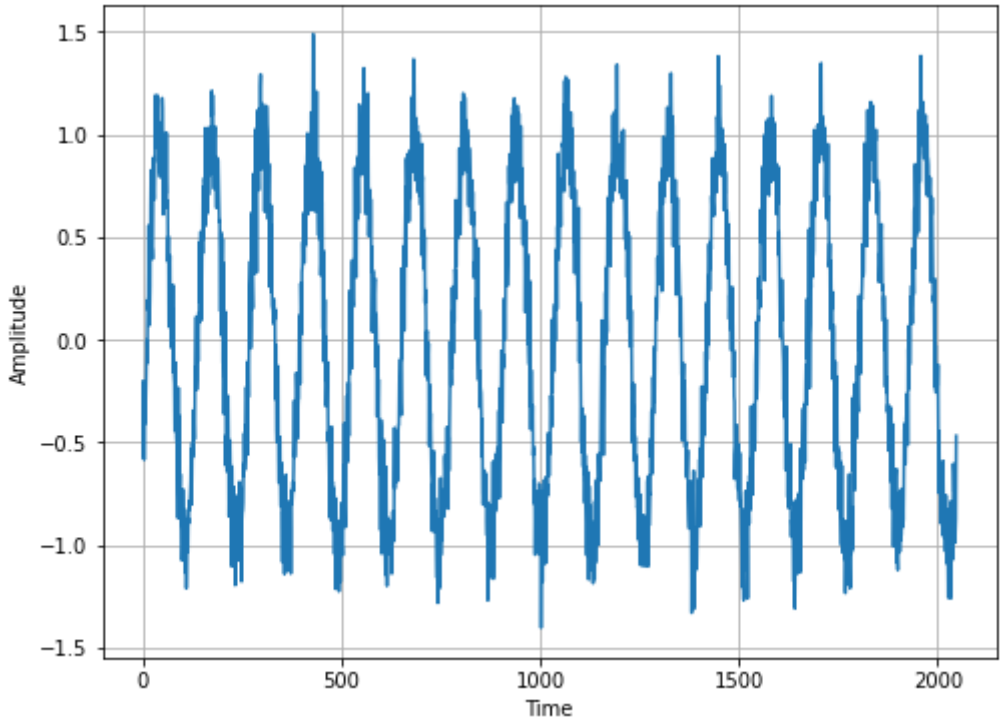


Figure 4: Noisy Sinewave signal generated from data

Network Design

This architecture that was found to be best fit was Bidirectional long-short term memory (Bidirectional LSTM). The second best was found to be LSTM (Long short-term memory) Model.

We are going to briefly discuss about LSTM and Bi-directional LSTM and how this was implemented in solving this problem in this project.

Both LSTM and Bi-directional LSTM are an advanced RNN (Recurrent Neural Network) [6]

RNN (recurrent neural network) is a type of neural network that are used to develop speech recognition and natural language processing models. They remember the sequence of the data being used and use that pattern to predict next steps [6].

A feedback loop system is used that makes it unique from other neural networks. Those loops process data in sequence and shares it to different nodes. Then it predicts the next steps from gathered information. This process is called memory. The loop structure takes the sequence of the input data and converts an independent variable to a dependant variable for its next layer [7].

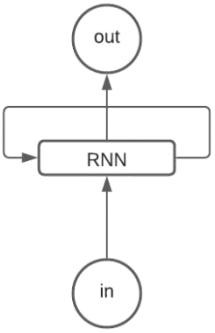


Figure 5: RNN loops [7]

In the above picture we can see how RNN processes the input, but whenever a different input comes, it gathers information from the loop and does the prediction.

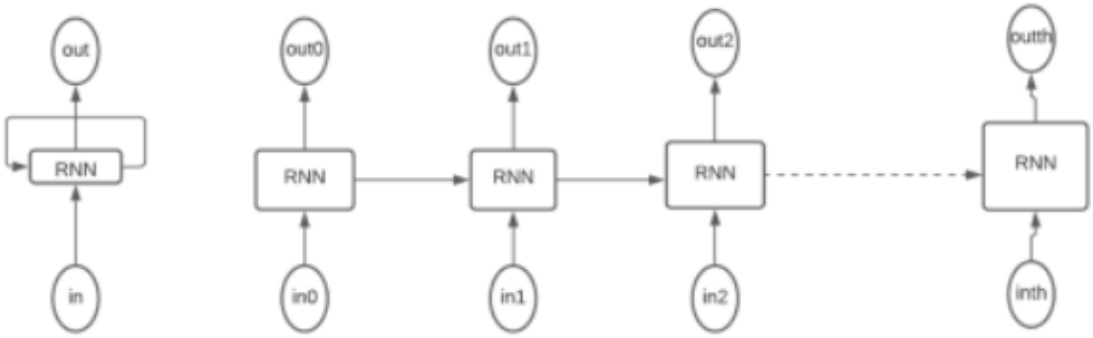


Figure 6: An unrolled recurrent neural network [7]

In the above picture we can see how a series of RNN would work.

However, the RNN architecture suffers from the long-term dependency problem which occurs while connecting previous information to new information. This is why LSTM (Long short-term memory network) was introduced which has a similar structure but different memory or repeating module.

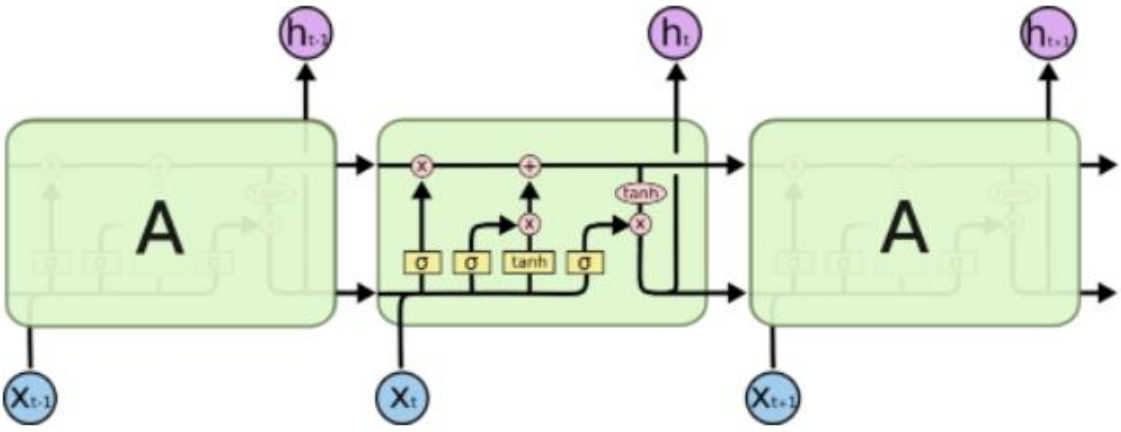
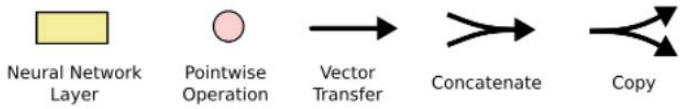


Figure 7: The repeating module in an LSTM contains four interacting layers [7]

The block diagram of the repeating module looks like the above image.

Where ,



Here, each line transmits the entire vector from the output node to input of next node. The pointwise operations are mathematical operations like vectors, the merging line concatenates vectors, and the diverging lines send information copies to different nodes. The horizontal line on top is the data conveyor.

1.1 LSTM

Now we will briefly see how gates work in LSTM. They allow flow of information to the lower part of the structure and thus control which information can be removed or added. This is how a unidirectional LSTM functions where the network is responsible for storing forward information. They are formed of sigmoid neural net layer and a pointwise multiplication operation as shown in below figure.

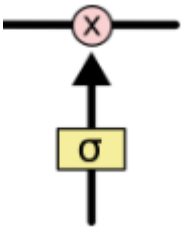


Figure 8: Gate cell [6]

The LSTM consists of three parts, as shown in the image below and each part performs an individual function.

In the first part, it is decided whether the information coming from the previous timestamp would be remembered or is irrelevant and could be forgotten. The second part of the cell attempts to take new information from the input. The last part of the cell passes the updated information from current timestamp to next timestamp [6].

These three cell parts of an LSTM are called gates where the first part is called **Forget gate, the** second part is known as **the Input gate** and the last one is **the Output gate** [6].

Like a simple RNN, an LSTM also has a hidden state where H_{t-1} represents the hidden state of the previous timestamp and H_t is the hidden state of the current timestamp. Along with this, it also has a cell state represented by C_{t-1} and C_t for previous and current timestamp respectively. The hidden state is called short term memory and the cell state is known as long term memory [6].

Below is the image of an LSTM cell representing the functions

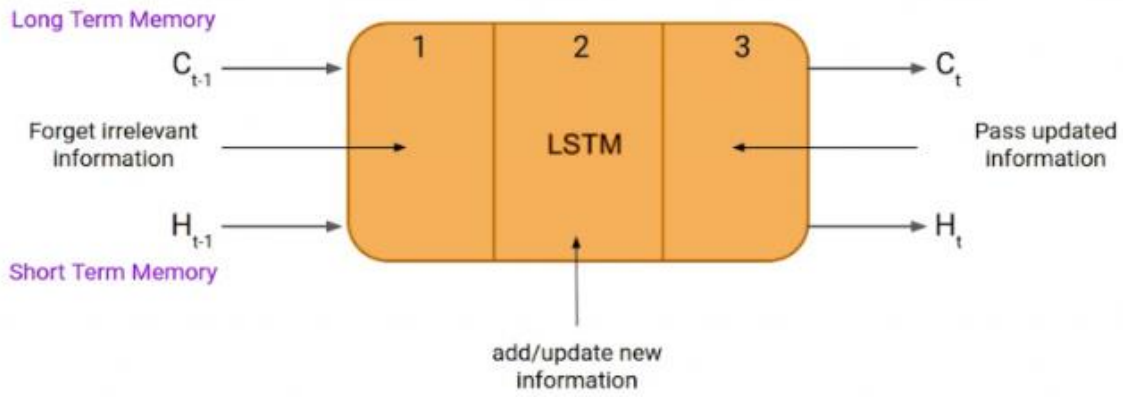


Figure 9: LSTM Cell information transfer [6]

The cell state carries information along with timestamps.

Figure 10: LSTM unit cell [6]

A **forget gate** in an LSTM network can be defined as whether information from previous timestamp is to be kept or forgotten. The equation of a Forget Gate is as follows:

Forget Gate: $f_t = \sigma(x_t * U_f + H_{t-1} * W_f)$

Where,

- X_t: input to the current timestamp.
- U_f: weight associated with the input
- H_{t-1}: The hidden state of the previous timestamp
- W_f: It is the weight matrix associated with hidden state

Later, a sigmoid function is applied over it. This makes f_t an integer between 0 and 1. It is later multiplied with the cell state of the previous timestamp as shown below**[6].**

$C_{t-1} * f_t = 0$ ...if $f_t = 0$ (forget everything)

$C_{t-1} * f_t = C_{t-1}$ ...if $f_t = 1$ (forget nothing)

An **Input Gate** is used to quantify the incoming new information from the input. The equation is as follows: $i_t = \sigma(x_t * U_i + H_{t-1} * W_i)$

Where,

- X_t: Input at the current timestamp t
- U_i: weight matrix of input
- H_{t-1}: A hidden state at the previous timestamp
- W_i: Weight matrix of input associated with hidden state

Again, sigmoid function is applied over it. As a result, the value of I at timestamp t would be between 0 and 1.

New information passed to a cell is a function of a hidden state at the previous timestamp t-1 and input x at timestamp t. For the activation function, tanh, the value of this new information is between -1 and 1. If N_t is negative, then information is subtracted from the cell state while if positive, then information is added to the cell state at current timestamp. The value of N_t does not get added directly to the cell state.

The updated equation is as follows: $C_t = f_t * C_{t-1} + i_t * N_t$ (updating cell state)

C_{t-1} is the cell state at current timestamp while others are the values that have been calculated before.

The equation of Output gate is like two previous gates, as below:

$o_t = \sigma(x_t * U_o + H_{t-1} * W_o)$

The values of the output gate would be between 0 and 1 because of the sigmoid function. To calculate the current hidden state we shall use this O_t and tanh of the updated cell state as follow: $H_t = o_t * \tanh(C_t)$

It is found that the hidden state is a function of Long term memory (C_t) and the current output. For taking the output of the current timestamp, SoftMax activation needs to be applied on the hidden state H_t.

$\text{Output} = \text{Softmax}(H_t)$

The token with the maximum score in the output would be the prediction **[6]**.

1.2 BI- LSTM (Bi-directional long-short term memory)

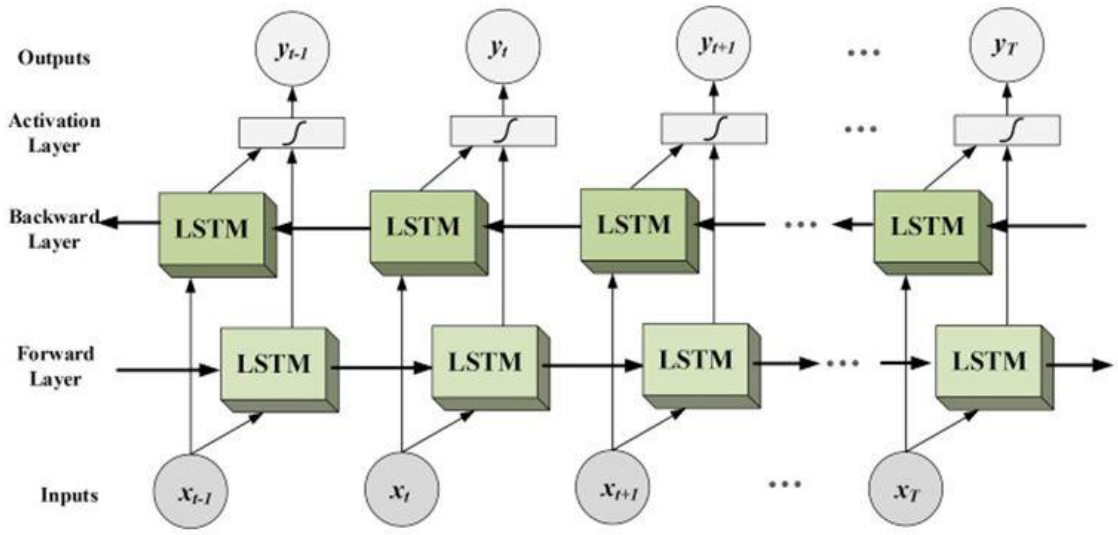


Figure 11: Information flow from backward and forward directions [7]

Bidirectional long-short term memory (BI-LSTM) is a neural network architecture that has sequence information in both directions i.e., backwards (future to past) or forward (past to future).

The key difference with regular LSTM and BI-LSTM is that the input flow is in one direction for the former while it is either backwards or forwards for the latter. In bi-directional LSTM, it is possible to make the input flow in both directions to preserve the past and future information [7].

In the above diagram, the flow of information is from backward and forward layers. BI-LSTM is usually used to determine sequence-to-sequence tasks such as text classification, speech recognition and forecasting models.

1.3 Use of LSTM and Bi-LSTM in Project

Among the RNN models generally used in Neural Network, LSTM is able to comprehend long term dependencies that are kept briefly in its cell states while controlling the flow of information by gates [3]. Our proposed algorithm adopts LSTM and Bi-LSTM. However, the accuracy is better in LSTM. We will discuss how both these architectures are used in the project.

In the Bi-LSTM, the output at current timestamp t is dependent on both previous data in the sequence and upcoming data. When we train two LSTMs within the Bi-LSTM, one on the forward while the other on the backward direction, this would occur. The LSTM in the forward direction adopts the forward dependencies, while the LSTM in the backward direction processes the use of backward dependencies. We get the output by combining the hidden states of both LSTMs.

A three-layer network with 64, 128, 256 neurons have been used in the first, second and third hidden layer. To prevent overfitting, a dropout of 50% is applied to its hidden layers. In other words, half of the total number of neurons are dropped randomly in these three layers. Therefore, the model learns to deduce a general relationship rather than memorize the data. This is done when a single neuron is forced to use rather than rely on neighbouring neurons [2]. Filters of various sizes are added to the convolutional layer such as 16, 32, 64, 128, 256. These filters are used to increase the training dataset for getting better results. After this, the activation function called Rectified Linear Units (ReLU) is applied. At the end a SoftMax layer is used for classification.

All codes were written in Python with the help of TensorFlow and Keras packages. Calculations were performed on a computer with a 4-core 3.50-GHz processor, 32 GB of RAM, and an NVIDIA-SMI GTX 750Ti GPU with 2 GB GDDR5 RAM. The processing of the data and training the final model took 12 h on computer.

Results

The hyper-parameters ware tuned on trial-and-error basis. Comparative analysis was done on both LSTM and Bi-LSTM models where the parameters used were kept as the same numerical value while training the models. The number of epochs that the models were trained was 50 and the batch size was 100. It was also observed that a larger batch size and data set provided better results and so, around 90000 samples were taken. Increasing the data set increased the accuracy from 16.5% to around 92% in both the models.

These below results demonstrate the benefits of memory buried in bidirectional LSTMs. It was also found that the loss in the model was quite consistent. Below are a few graphs of LSTM model and Bi-LSTM Model.

In the below figure [Fig 12], we can see how the accuracy have steadily increased for both test and train dataset in Bi-LSTM Model. Here the accuracy increased steadily up to around 93%.

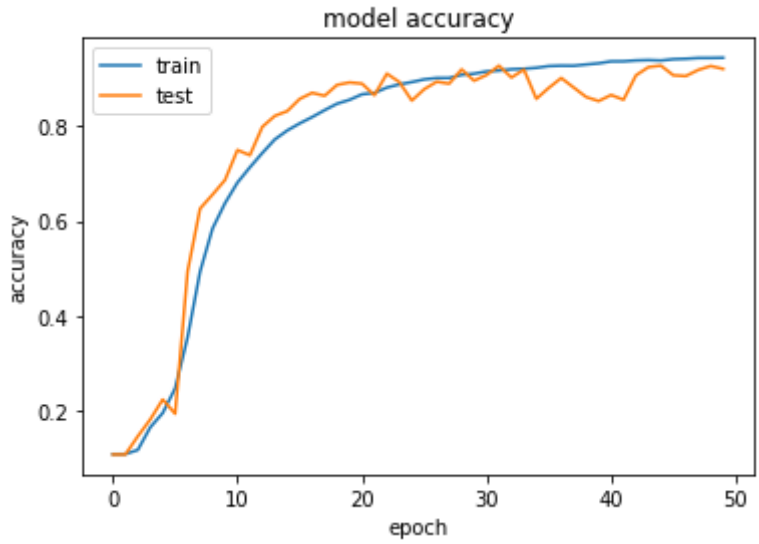


Fig 12: Model accuracy in Bi-LSTM Model

The below plot [Fig 13] shows the model loss for Bi-LSTM which shows limited variation and the loss has decreased gradually when the model was trained for more epochs.

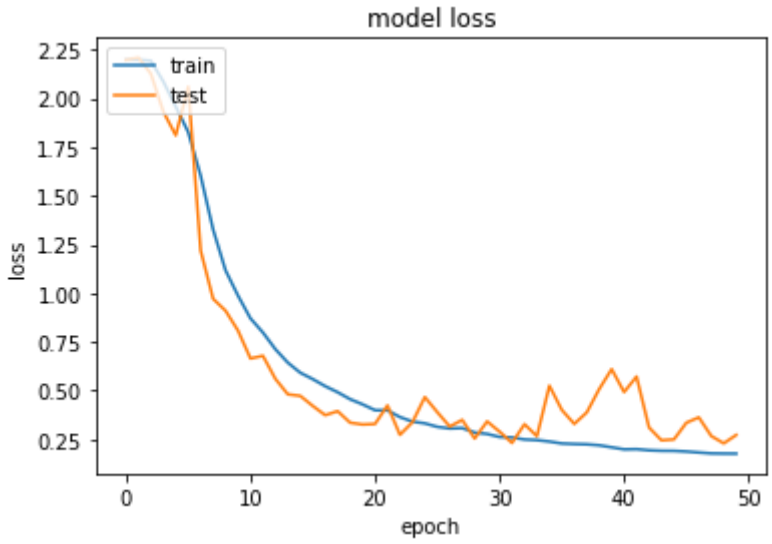


Fig 13: Model loss in Bi-LSTM Model

In the below figure [Fig 14], we can see the model accuracy for LSTM which was around a maximum value of 92- 94%.

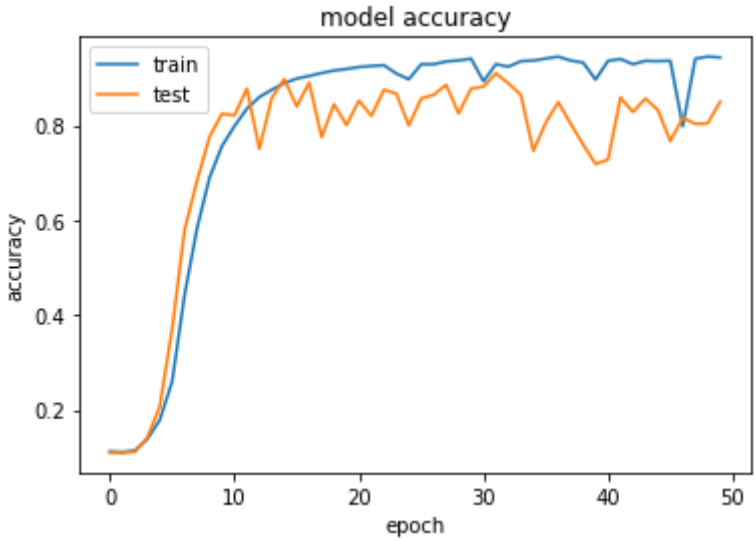


Fig 14: Model accuracy in LSTM Model

In the below plot [Fig 15], we can see the model loss has decreased overall but has suffered higher fluctuations.

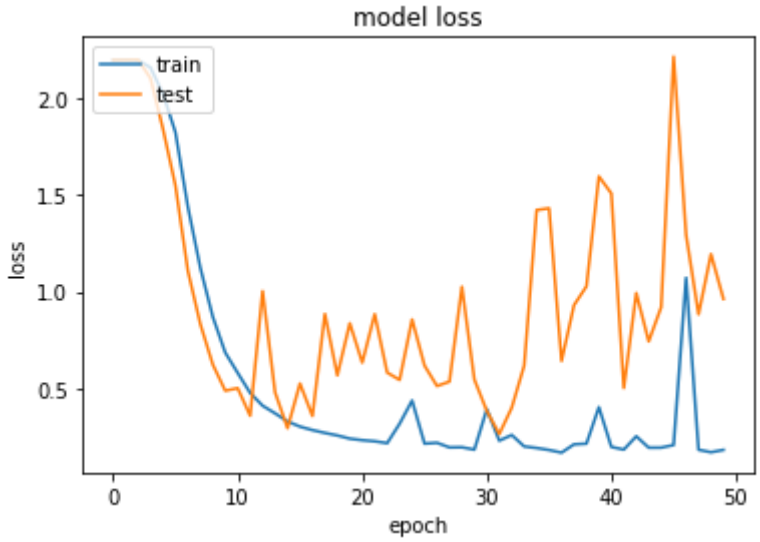


Fig 15: Model loss in LSTM Model

Comparing the two models, it was found that Bi -LSTM performed better than LSTM. This is because although the accuracy varied in similar range of 90-94%, the loss was higher in LSTM than Bi-LSTM.

For data visualization, we have used seaborn package. Confusion matrix and classification report was created to plot and test the model.

Various metrics such as precision, recall and f-beta score were measured to get the classification report.

Precision is the calculated based on the ratio: tp / (tp + fp) where tp is the number of true positives and fp the number of false positives. It is a classifier that characterizes a sample not to be labelled as positive when it is negative.

Recall is the ratio tp / (tp + fn) where tp is the number of true positives and fn the number of false negatives. Here, the classifier finds all the positive samples.

F-beta score interprets the weighted harmonic mean of the precision and recall, where an F-beta score has the best value at 1 and worst score at 0.

The F-beta score weights recall more than precision by a factor of beta. beta == 1.0 means recall, and precision are equally important.

The support is the number of occurrences of each class in y_true.

	precision	recall	f1-score	support
(250)	1.00	0.85	0.92	2001
(500)	0.99	0.95	0.97	2014
(750)	0.99	0.95	0.97	2078
(1000)	0.97	0.97	0.97	2057
(1500)	0.98	0.96	0.97	1970
(2000)	0.97	0.95	0.96	1971
(3000)	0.91	0.92	0.91	1976
(4000)	0.66	0.99	0.79	1982
(6000)	0.96	0.74	0.84	1951

Fig 16: Classification report for Bi-LSTM Model

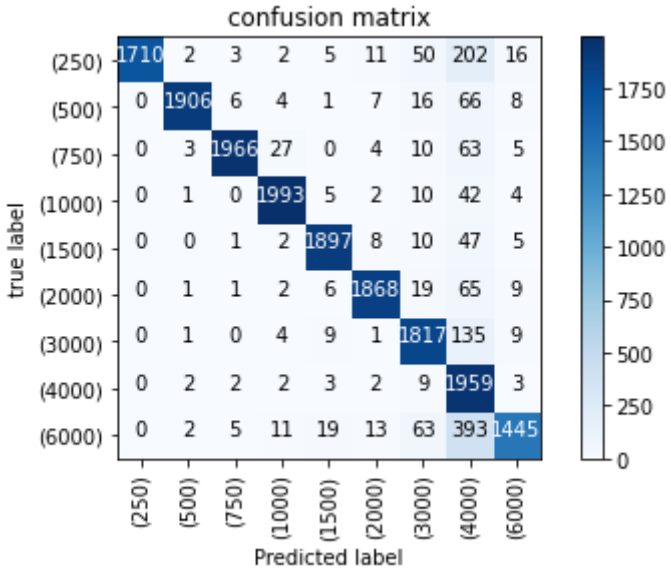


Fig 17: Confusion Matrix for Bi-LSTM Model

	precision	recall	f1-score	support
(250)	1.00	0.74	0.85	2001
(500)	1.00	0.72	0.84	2014
(750)	1.00	0.82	0.90	2078
(1000)	1.00	0.90	0.95	2057
(1500)	1.00	0.76	0.86	1970
(2000)	1.00	0.89	0.94	1971
(3000)	1.00	0.89	0.94	1976
(4000)	0.99	0.94	0.96	1982
(6000)	0.42	1.00	0.59	1951

Fig 18: Classification report for LSTM Model

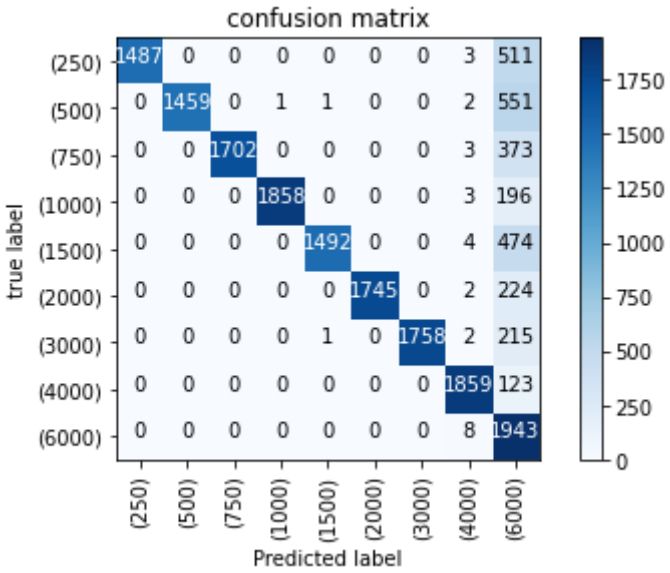


Fig 19: Confusion Matrix for LSTM Model

Comparing the results for both Bi-LSTM and LSTM, it is found that the precision for true positives is much higher in Bi-LSTM than those found in LSTM. This further confirms that Bi-LSTM outperformed LSTM in finding the frequencies in the noisy data.

Conclusion

Deep Learning was used to train a model that had noisy sinusoidal signal as input data. 90000 samples were used to train, test, and validate the model.

Data pre-processing and model architecture was discussed where it was explained how using deep neural networks such as LSTM and Bi-LSTM is trained to get around 92 - 94% accuracy. It was found through the metrics calculation and confusion matrix plotting that Bi-LSTM outperformed LSTM as it had more true positives. Even though the accuracy of both the models ranged between 92-94%, the model loss was lesser in Bi-LSTM.

Finally, we concluded that Bi-LSTM produced better results than LSTM.

Reference

[1] Sajedian, I., Rho, J. Accurate and instant frequency estimation from noisy sinusoidal waves by deep learning. *Nano Convergence* **6,** 27 (2019). https://doi.org/10.1186/s40580-019-0197-y

[2] V. A. Chenarlogh, H. B. Jond and J. Platoš, "A Robust Deep Model for Human Action Recognition in Restricted Video Sequences," 2020 43rd International Conference on Telecommunications and Signal Processing (TSP), 2020, pp. 541-544, doi: 10.1109/TSP49548.2020.9163464.

[3] Colah (August 27, 2015) Colah's Blog; Understanding LSTM Networks https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[4] IBM Cloud Education (17 August 2020) Neural Networks https://www.ibm.com/cloud/learn/neural-networks

[5] Machine Learning Crash Course(2020-02-10) Training and Test Sets: Splitting Data https://developers.google.com/machine-learning/crash-course/training-and-test-sets/splitting-data

[6] Saxena,S (March 16, 2021) Introduction to Long Short Term Memory (LSTM) https://www.analyticsvidhya.com/blog/2021/03/introduction-to-long-short-term-memory-lstm/

[7] Verma,Y (July 17, 2021) Complete Guide To Bidirectional LSTM https://analyticsindiamag.com/complete-guide-to-bidirectional-lstm-with-python-codes/

Appendix

```
from sklearn.utils import shuffle
import pandas as pd
import seaborn as sns
from math import sqrt
import math
import tensorflow as tf
from sklearn.metrics import mean_squared_error, mean_absolute_error
from sklearn.preprocessing import StandardScaler,MinMaxScaler
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from tensorflow.keras.optimizers import Adam
from tensorflow import keras
from tensorflow.keras.layers import Bidirectional
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Conv2D, Flatten, Dense,Input, Dense,
Dropout
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import LSTM, MaxPooling1D, Conv1D, Flatten,
LeakyReLU, Activation, SimpleRNN
from tensorflow.keras.callbacks import EarlyStopping,
LearningRateScheduler
from tensorflow.keras.preprocessing.sequence import TimeseriesGenerator
import datetime
from pandas import Series
from numpy.random import randn
import os
import random as rn
import numpy as np
import time
import matplotlib.pyplot as plt
import random
from tensorflow.keras import backend as t
from tensorflow.keras.layers import Embedding, Masking
from tensorflow.keras.utils import to_categorical

#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%
sine_samples = []
data = []
fs = 32000 #sampling rate
Ts = 1/fs #time duration
#t=0.1
freqList = [250, 500, 750 , 1000, 1500, 2000, 3000, 4000, 6000]
#n = np.linspace(0,1,2048) #no of samples
n = np.arange(0, 2048, 1) #no of samples
print("number of samples is = ", n)
```

```
for freq in freqList:
   list_phi=[]
   for _ in range(10000):
       phi = np.random.rand()*(2*np.pi)
       list_phi.append(phi)
   for phases in range(len(list_phi)):
       alpha = np.random.uniform(0.1,5)
       #print (alpha)
       wave = np.sin(2* np.pi * freq * n / fs + phases)
       noise = alpha * np.random.normal(0, 1, size = (2048,))
       signal = wave + noise
       sine_samples.append(signal)

train_data = np.array(sine_samples)
num_samples = len(train_data)
print(num_samples)
print(train_data.shape)



#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%
label1=np.ones((num_samples),dtype = int)
label1[0:9999]= 0
label1[10000:19999] = 1
label1[20000:29999] = 2
label1[30000:39999] = 3
label1[40000:49999] = 4
label1[50000:59999] = 5
label1[60000:69999] = 6
label1[70000:79999] = 7
label1[80000:89999] = 8

num_classes = 9
Y=to_categorical(label1, num_classes)



def Normalize(data):
    mean_data=np.mean(data)
    std_data=np.std(data)
    norm_data=(data-mean_data)/std_data
    return norm_data

#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%
x_train,x_test,y_train,y_test=train_test_split(train_data, Y,
test_size=0.2,random_state=2)
x_train=Normalize(x_train)
x_test=Normalize(x_test)
```

```
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%
n_units=[64, 128, 256]
n_epochs=50
batch_size = 100
data_shape = (2048, 1)
filters = [16, 32, 64, 128, 256]
model = Sequential()
#model.add(LSTM(n_units, input_shape = data_shape))
#model.add(Masking( mask_value=0, batch_input_shape=(batch_size,
data_shape[0], data_shape[1])))
model.add(Conv1D(filters[0], kernel_size=3, padding='same', strides = 1,
                 batch_input_shape=(batch_size, data_shape[0],
data_shape[1])))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Conv1D(filters[1], kernel_size=3, strides = 1, padding='same'))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Conv1D(filters[2], kernel_size=3, strides = 1, padding='same'))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Conv1D(filters[3], kernel_size=3, strides = 1, padding='same'))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Conv1D(filters[4], kernel_size=3, strides = 1, padding='same'))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Bidirectional(LSTM(n_units[0], stateful=True, return_sequences=
True)))
model.add(Dropout(0.2))
model.add(Activation('relu'))

model.add(Bidirectional(LSTM(n_units[1], stateful=True, return_sequences=
True)))
model.add(Dropout(0.2))
model.add(Activation('relu'))

model.add(Bidirectional(LSTM(n_units[2], stateful=True, return_sequences=
False)))
model.add(Dropout(0.2))
model.add(Activation('relu'))
model.add(Dense(num_classes, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='rmsprop',
metrics=['accuracy'])
```

```
history = model.fit(x_train, y_train,
validation_data=(x_test,y_test),  batch_size= batch_size,
epochs=n_epochs, shuffle=False)



#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%
n_units=[64, 128, 256]
n_epochs=50
batch_size = 100
data_shape = (2048, 1)
filters = [16, 32, 64, 128, 256]
model = Sequential()
#model.add(LSTM(n_units, input_shape = data_shape))
#model.add(Masking( mask_value=0, batch_input_shape=(batch_size,
data_shape[0], data_shape[1])))
model.add(Conv1D(filters[0], kernel_size=3, padding='same', strides = 1,
                 batch_input_shape=(batch_size, data_shape[0],
data_shape[1])))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Conv1D(filters[1], kernel_size=3, strides = 1, padding='same'))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Conv1D(filters[2], kernel_size=3, strides = 1, padding='same'))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Conv1D(filters[3], kernel_size=3, strides = 1, padding='same'))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Conv1D(filters[4], kernel_size=3, strides = 1, padding='same'))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(LSTM(n_units[0], return_sequences=True))
model.add(Activation('relu'))

model.add(LSTM(n_units[1], return_sequences=True))
model.add(Activation('relu'))

model.add(LSTM(n_units[2]))
model.add(Activation('relu'))
```

```
model.add(Dropout(0.2))

model.add(Dense(num_classes, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='rmsprop',
metrics=['accuracy'])
history = model.fit(x_train, y_train,
validation_data=(x_test,y_test),  batch_size= batch_size,
epochs=n_epochs, shuffle=False)



loss_per_epoch = history.history['loss']
plt.plot(range(len(loss_per_epoch)), loss_per_epoch)

# list all data in history
print(history.history.keys())
# summarize history for accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()


import itertools
from sklearn.metrics import classification_report,confusion_matrix

def plot_confusion_matrix(cm,classes,normalize=False,title='Confusion
matrix',cmap=plt.cm.Blues):
    plt.imshow(cm,interpolation='nearest',cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes,rotation=90)
    plt.yticks(tick_marks, classes)
    if normalize:
        cm=cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print("Confusion matrix without normalization")
```

```
    print(cm)
    thresh=cm.max() / 2.
    for i,j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j,i,cm[i,j],horizontalalignment="center",
        color="white" if cm[i,j] > thresh else "black")
    plt.tight_layout()
    plt.ylabel('true label')
    plt.xlabel('Predicted label')


Y_pred =model.predict(x_test,batch_size=batch_size,verbose=0)
print(Y_pred)
y_pred = np.argmax(Y_pred, axis=1)
#y_pred = model.predict_classes(x_test, batch_size=batch_size, verbose=0)
target_names = ['(250)','(500)','(750)', '(1000)','(1500)','(2000)',
'(3000)', '(4000)', '(6000)']
print(classification_report(np.argmax(y_test, axis=1), y_pred,
target_names=target_names))
cnf_matrix=(confusion_matrix(np.argmax(y_test,axis=1), y_pred))
plot_confusion_matrix(cnf_matrix, classes=target_names, title='confusion
matrix')
```