In [1]:
```
from google.colab import drive
drive.mount('content')
```

```
Mounted at content
```

In [2]:
```
cd /content/content/MyDrive
```

```
/content/content/MyDrive
```

In [3]:
```
ls
```

```
'Colab Notebooks'/   recordingss.zip                       weight_cifer/
 lstm/               spoken_digit/
 recordings/         spoken_digit-20210126T005131Z-001.zip
```

Spoken Digit Recognition

In this notebook, You will do Spoken Digit Recognition.

Input - speech signal, output - digit number

It contains

1. Reading the dataset. and Preprocess the data set. Detailed instrctions are given below. You have to write the code in the same cell which contains the instrction.
2. Training the LSTM with RAW data
3. Converting to spectrogram and Training the LSTM network
4. Creating the augmented data and doing step 2 and 3 again.

instructions:

1. Don't change any Grader Functions. Don't manipulate any Grader functions. If you manipulate any, it will be considered as plagiarised.

2. Please read the instructions on the code cells and markdown cells. We will explain what to write.

3. please return outputs in the same format what we asked. Eg. Don't return List of we are asking for a numpy array.

4. Please read the external links that we are given so that you will learn the concept behind the code that you are writing.

5. We are giving instructions at each section if necessary, please follow them.

Every Grader function has to return True.

In [4]:
```
import numpy as np
from tensorflow.keras.layers import Input, LSTM, Dense
from sklearn.metrics import roc_auc_score
from sklearn.metrics import f1_score
from tensorflow.keras.models import Model
import tensorflow as tf
import pandas as pd
import librosa
import os
##if you need any imports you can do that here.
```

In [5]:
```
all_files = list()
data_files = os.listdir('recordings')
for i,sub_file in enumerate(data_files):
  if (sub_file.endswith("wav")):
    sub_file_path = 'recordings' + '/' + sub_file
    sub_file_path = str(sub_file_path)
    all_files.append(sub_file_path)
```

Grader function 1

In [6]:
```
def grader_files():
    temp = len(all_files)==2000
    temp1 = all([x[-3:]=="wav" for x in all_files])
    temp = temp and temp1
    return temp
grader_files()
```

Out[6]: True

Create a dataframe(name=df_audio) with two columns(path, label).
You can get the label from the first letter of name.

Eg: 0_jackson_0 --> 0

0_jackson_43 --> 0

In [7]:
```
#Create a dataframe(name=df_audio) with two columns(path, label).
#You can get the label from the first letter of name.
#Eg: 0_jackson_0 --> 0
#0_jackson_43 --> 0
label = [int(x[11]) for x in all_files]
label[0:5]

# intialise data of lists.
data = {'path':all_files,
        'label':label}

# Create DataFrame
df_audio = pd.DataFrame(data)
df_audio.head(5)
```

Out[7]:

	path	label
0	recordings/7_nicolas_39.wav	7
1	recordings/9_nicolas_32.wav	9
2	recordings/8_theo_9.wav	8
3	recordings/7_theo_20.wav	7
4	recordings/0_nicolas_8.wav	0

In [8]:
```
y = df_audio['label'].values
```

In [9]:
```
#info
df_audio.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2000 entries, 0 to 1999
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   path    2000 non-null   object
 1   label   2000 non-null   int64
dtypes: int64(1), object(1)
memory usage: 31.4+ KB
```

Grader function 2

In [10]:
```
def grader_df():
    #flag_shape = df_audio.shape==(2000,2)
    flag_columns = all(df_audio.columns==['path', 'label'])
    list_values = list(df_audio.label.value_counts())
    flag_label = len(list_values)==10
    flag_label2 = all([i==200 for i in list_values])
    final_flag = flag_columns and flag_columns and flag_label and flag_label2
    return final_flag
grader_df()
```

Out[10]: True

In [11]:
```
from sklearn.utils import shuffle
df_audio = shuffle(df_audio, random_state=33)#don't change the random state
```

Train and Validation split

In [12]:
```
#split the data into train and validation and save in X_train, X_test, y_train, y_test
#use stratify sampling
#use random state of 45
#use test size of 30%
# train test split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(df_audio['path'],df_audio['label'] , test_size=0.3, stratify=df_audio['label'])
```

Grader function 3

In [13]:
```
def grader_split():
    flag_len = (len(X_train)==1400) and (len(X_test)==600) and (len(y_train)==1400) and (len(y_test)==600)
    values_ytrain = list(y_train.value_counts())
    flag_ytrain = (len(values_ytrain)==10) and (all([i==140 for i in values_ytrain]))
    values_ytest = list(y_test.value_counts())
    flag_ytest = (len(values_ytest)==10) and (all([i==60 for i in values_ytest]))
    final_flag = flag_len and flag_ytrain and flag_ytest
    return final_flag
grader_split()
```

Out[13]: True

Preprocessing

All files are in the "WAV" format. We will read those raw data files using the librosa

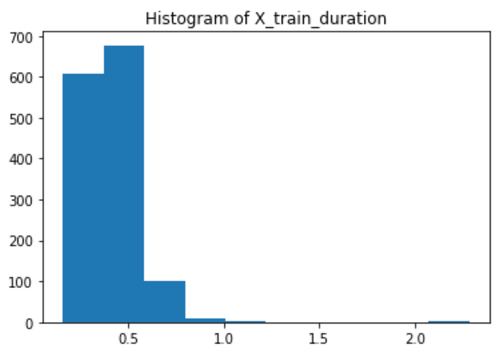
In [14]:
```
sample_rate = 22050
def load_wav(x, get_duration=True):
    '''This return the array values of audio with sampling rate of 22050 and Duration'''
    #loading the wav file with sampling rate of 22050
    samples, sample_rate = librosa.load(x, sr=22050)
    if get_duration:
        duration = librosa.get_duration(samples, sample_rate)
        return [samples, duration]
    else:
        return samples
```

In [15]:
```
X_train_process = [load_wav(row, get_duration=True) for row in X_train.tolist()]
X_test_process = [load_wav(row, get_duration=True) for row in X_test.tolist()]
```

In [ ]:
```
#use load_wav function that was written above to get every wave.
#save it in X_train_processed and X_test_processed
# X_train_processed/X_test_processed should be dataframes with two columns(raw_data, duration) with same index of X_train/y_train
```

In [16]:
```
from matplotlib import pyplot
#plot the histogram of the duration for trian

X_train_duration  = [i[1] for i in X_train_process]
# plot scores
pyplot.hist(X_train_duration)
pyplot.title("Histogram of X_train_duration")
pyplot.show()
```



In [ ]:
```
#print 0 to 100 percentile values with step size of 10 for train data duration.
```

In [ ]:
```
import numpy as np
p = [0,10,20,30,40,50,60,70,80,90,100]
range =  np.percentile(X_train_duration, p)

for i , j  in enumerate(range):
  print(f'{p[i]} th percentile is {range[i]}')
```

```
0 th percentile is 0.1435374149659864
10 th percentile is 0.25988208616780045
20 th percentile is 0.30080725623582766
30 th percentile is 0.33424489795918366
40 th percentile is 0.36007256235827667
50 th percentile is 0.3915873015873016
60 th percentile is 0.418639455782313
70 th percentile is 0.44988662131519275
80 th percentile is 0.48596825396825394
90 th percentile is 0.5549160997732426
100 th percentile is 2.282766439909297
```

In [ ]:
```
##print 90 to 100 percentile values with step size of 1.
```

In [ ]:
```
import numpy as np
p = [90,91,92,93,94,95,96,97,98,99,100]
range =  np.percentile(X_train_duration, p)

for i , j  in enumerate(range):
  print(f'{p[i]} th percentile is {range[i]}')
```

```
90 th percentile is 0.5549160997732426
91 th percentile is 0.5659854875283448
92 th percentile is 0.5779083900226759
93 th percentile is 0.5933292517006803
94 th percentile is 0.609092970521542
95 th percentile is 0.6231496598639454
96 th percentile is 0.6420553287981859
97 th percentile is 0.6635741496598639
98 th percentile is 0.6956090702947844
99 th percentile is 0.7831392290249433
100 th percentile is 2.282766439909297
```

In [18]:
```
X_trian_raw_data = [i[0] for i in X_train_process]
X_train_duration = [i[1] for i in X_train_process]

dict = {'raw_data':X_trian_raw_data , 'duration':X_train_duration}

X_train_processed = pd.DataFrame(dict)
```

In [19]:
```
X_test_raw_data = [i[0] for i in X_test_process]
X_test_duration = [i[1] for i in X_test_process]

dict = {'raw_data':X_test_raw_data , 'duration':X_test_duration}

X_test_processed = pd.DataFrame(dict)
```

Grader function 4

In [20]:
```
def grader_processed():
    flag_columns = (all(X_train_processed.columns==['raw_data', 'duration'])) and (all(X_test_processed.columns==['raw_data', 'duration']))
    flag_shape = (X_train_processed.shape ==(1400, 2)) and (X_test_processed.shape==(600,2))
    return flag_columns and flag_shape
grader_processed()
```

Out[20]: True

Based on our analysis 99 percentile values are less than 0.8sec so we will limit maximum length of X_train_processed and X_test_processed to 0.8 sec. It is similar to pad_sequence for a text dataset.

While loading the audio files, we are using sampling rate of 22050 so one sec will give array of length 22050. so, our maximum length is 0.8*22050 = 17640

Pad with Zero if length of sequence is less than 17640 else Truncate the number.

Also create a masking vector for train and test.

masking vector value = 1 if it is real value, 0 if it is pad value. Masking vector data type must be bool.

In [21]:
```
X_train_sample = [i[0] for i in X_train_processed]
X_test_sample = [i[0] for i in X_test_processed]
```

In [22]:
```
max_length  = 17640
```

In [23]:
```
## as discussed above, Pad with Zero if length of sequence is less than 17640 else Truncate the number.
## save in the X_train_pad_seq, X_test_pad_seq
## also Create masking vector X_train_mask, X_test_mask

## all the X_train_pad_seq, X_test_pad_seq, X_train_mask, X_test_mask will be numpy arrays mask vector dtype must be bool.
```

In [24]:
```
X_train_processed.head()
```

Out[24]:

	raw_data	duration
0	[0.004544994, -0.0043530054, -0.012745843, -0....	0.422404
1	[0.00025552875, 2.5027432e-05, 0.00012410665, ...	0.391882
2	[-0.00746698, -0.0061093145, -0.0028450494, 0....	0.374785
3	[-0.013037005, -0.009333977, 0.0036948763, 0.0...	0.469887
4	[-0.0003973734, -0.00043307224, -0.00045968636...	0.318005

In [25]:
```
X_train_sample = X_train_processed['raw_data'].values
X_test_sample = X_test_processed['raw_data'].values
```

In [26]:
```
X_train_sample.shape,X_test_sample.shape
```

Out[26]: ((1400,), (600,))

In [27]:
```
X_train_pad_seq = []
for li in X_train_sample:
  if len(li) < max_length:
    li = li.tolist()
    a = [0]*(max_length - len(li))
    li.extend(a)
    X_train_pad_seq.append(li)

  else:
    li = li.tolist()
    X_train_pad_seq.append(li[0:max_length])

X_train_pad_seq = np.array(X_train_pad_seq)


X_test_pad_seq = []
for li in X_test_sample:
```

```
    if len(li) < max_length:
      li = li.tolist()
      a = [0]*(max_length - len(li))
      li.extend(a)
      X_test_pad_seq.append(li)

    else:
      li = li.tolist()
      X_test_pad_seq.append(li[0:max_length])

X_test_pad_seq = np.array(X_test_pad_seq)
```

In [28]:
```
X_train_mask = np.array([(i > 0).tolist()for i in X_train_pad_seq])
X_test_mask = np.array([(i > 0).tolist()for i in X_test_pad_seq])
```

In [29]:
```
X_train_mask.dtype , X_test_mask.dtype
```

Out[29]: (dtype('bool'), dtype('bool'))

Grader function 5

In [30]:
```
def grader_padoutput():
    flag_padshape = (X_train_pad_seq.shape==(1400, 17640)) and (X_test_pad_seq.shape==(600, 17640)) and (y_train.shape==(1400,))
    flag_maskshape = (X_train_mask.shape==(1400, 17640)) and (X_test_mask.shape==(600, 17640)) and (y_test.shape==(600,))
    flag_dtype = (X_train_mask.dtype==bool) and (X_test_mask.dtype==bool)
    return flag_padshape and flag_maskshape and flag_dtype
    return flag_padshape
grader_padoutput()
```

Out[30]: True

In [31]:
```
X_train_mask = X_train_mask.astype('float')
X_test_mask = X_test_mask.astype('float')
```

1. Giving Raw data directly.

```
Now we have

Train data: X_train_pad_seq, X_train_mask and y_train
Test data: X_test_pad_seq, X_test_mask and y_test

We will create a LSTM model which takes this input.

Task:

1. Create an LSTM network which takes "X_train_pad_seq" as input, "X_train_mask" as mask input. You can use any number
of LSTM cells. Please read LSTM documentation(https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTM) in
tensorflow to know more about mask and also https://www.tensorflow.org/guide/keras/masking_and_padding
2. Get the final output of the LSTM and give it to Dense layer of any size and then give it to Dense layer of size
10(because we have 10 outputs) and then compile with the sparse categorical cross entropy( because we are not
converting it to one hot vectors).
3. Use tensorboard to plot the graphs of loss and metric(use micro F1 score as metric) and histograms of gradients.
4. make sure that it won't overfit.
5. You are free to include any regularization
```

In [32]:
```
X_train_pad_seq.shape[1]
```

Out[32]: 17640

In [33]:
```
Y_train = tf.keras.utils.to_categorical(y_train, 10)
Y_test = tf.keras.utils.to_categorical(y_test, 10)
```

MODEL :- 1

In [ ]:
```
## as discussed above, please write the LSTM

lstm = LSTM(units = 25,activation="tanh",kernel_initializer=tf.keras.initializers.he_uniform(seed=0))

input_layer = Input(shape=(X_train_pad_seq.shape[1],1),dtype=float )
input_mask = Input(shape=(X_train_mask.shape[1],1),dtype=bool)
LSTM_layer  = lstm(inputs=input_layer,mask=input_mask)
dense = Dense(50,activation="relu",kernel_initializer=tf.keras.initializers.he_uniform(seed=0))(LSTM_layer)
output_1 = Dense(10,activation='softmax',kernel_initializer=tf.keras.initializers.GlorotUniform(seed=0))(dense)

model = Model(inputs = [input_layer,input_mask],outputs = output_1)
model.compile(optimizer='adam',loss = tf.keras.losses.sparse_categorical_crossentropy,metrics='accuracy')
model.summary()
```

```
Model: "model"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            [(None, 17640, 1)]   0
__________________________________________________________________________________________________
```

```
input_2 (InputLayer)            [(None, 17640, 1)]   0
__________________________________________________________________________________________________
lstm (LSTM)                     (None, 25)           2700        input_1[0][0]
                                                                 input_2[0][0]
__________________________________________________________________________________________________
dense (Dense)                   (None, 50)           1300        lstm[0][0]
__________________________________________________________________________________________________
dense_1 (Dense)                 (None, 10)           510         dense[0][0]
==================================================================================================
Total params: 4,510
Trainable params: 4,510
Non-trainable params: 0
__________________________________________________________________________________________________
```

In [34]:

```
class f1_score_and_auc_Callback(tf.keras.callbacks.Callback):

    def  on_train_begin(self,logs={}):
      self.f1_micro=[]
      self.auc_score=[]

    def on_epoch_end(self, epoch, logs=None):
      y_pred=self.model.predict([X_test_pad_seq,X_test_mask])
      y_pred = np.argmax(y_pred, axis = 1)

      y_true=y_test
      score=f1_score(y_true, y_pred, average='micro')

      self.f1_micro.append(score)
      print(" F1 micro :",score)

metrics=f1_score_and_auc_Callback()
```

In [ ]:

```
model.fit(x=[X_train_pad_seq,X_train_mask],y=y_train,validation_data=([X_test_pad_seq,X_test_mask],y_test),
            epochs=5,batch_size=10,steps_per_epoch=len(X_train_mask)//10 , callbacks=metrics)
```

```
Epoch 1/5
140/140 [==============================] - 63s 227ms/step - loss: 2.3092 - accuracy: 0.1023 - val_loss: 2.3015 - val_accuracy: 0.0917
 F1 micro : 0.10499999999999998
Epoch 2/5
140/140 [==============================] - 30s 217ms/step - loss: 2.2991 - accuracy: 0.0888 - val_loss: 2.2991 - val_accuracy: 0.1400
 F1 micro : 0.115
Epoch 3/5
140/140 [==============================] - 31s 218ms/step - loss: 2.2981 - accuracy: 0.1119 - val_loss: 2.2984 - val_accuracy: 0.1100
 F1 micro : 0.095
Epoch 4/5
140/140 [==============================] - 30s 217ms/step - loss: 2.2939 - accuracy: 0.1097 - val_loss: 2.2948 - val_accuracy: 0.1317
 F1 micro : 0.10333333333333333
Epoch 5/5
140/140 [==============================] - 30s 218ms/step - loss: 2.2918 - accuracy: 0.1166 - val_loss: 2.2915 - val_accuracy: 0.1250
 F1 micro : 0.09333333333333334
```

Out[ ]: `<tensorflow.python.keras.callbacks.History at 0x7fa4f57bf7d0>`

2. Converting into spectrogram and giving spectrogram data as input

```
We can use librosa to convert raw data into spectrogram. A spectrogram shows the features in a two-dimensional
representation with the
intensity of a frequency at a point in time i.e we are converting Time domain to frequency domain. you can read more
about this in https://pnsn.org/spectrograms/what-is-a-spectrogram
```

In [35]:

```
def convert_to_spectrogram(raw_data):
    '''converting to spectrogram'''
    spectrum = librosa.feature.melspectrogram(y=raw_data, sr=sample_rate, n_mels=64)
    logmel_spectrum = librosa.power_to_db(S=spectrum, ref=np.max)
    return logmel_spectrum
```

In [36]:

```
X_train_spectrogram = [convert_to_spectrogram(row) for row in X_train_pad_seq]
X_train_spectrogram = np.array(X_train_spectrogram)

X_test_spectrogram = [convert_to_spectrogram(row) for row in X_test_pad_seq]
X_test_spectrogram = np.array(X_test_spectrogram)
```

In [ ]:

```
##use convert_to_spectrogram and convert every raw sequence in X_train_pad_seq and X_test_pad-seq.
## save those all in the X_train_spectrogram and X_test_spectrogram ( These two arrays must be numpy arrays)
```

Grader function 6

In [37]:

```
def grader_spectrogram():
    flag_shape = (X_train_spectrogram.shape==(1400,64, 35)) and (X_test_spectrogram.shape == (600, 64, 35))
    return flag_shape
grader_spectrogram()
```

Out[37]: True

In [ ]:

MODEL :- 2

In [45]:
```
class f1_score_and_auc_Callback(tf.keras.callbacks.Callback):

    def  on_train_begin(self,logs={}):
      self.f1_micro=[]
      self.auc_score=[]

    def on_epoch_end(self, epoch, logs=None):
      y_pred=self.model.predict(X_test_spectrogram)
      y_pred = np.argmax(y_pred, axis = 1)

      y_true=y_test
      score=f1_score(y_true, y_pred, average='micro')


      self.f1_micro.append(score)
      print(" F1 micro :",score)


metrics1=f1_score_and_auc_Callback()
```

In [38]:
```
## as discussed above, please write the LSTM

lstm = LSTM(units = 100,activation="tanh",kernel_initializer=tf.keras.initializers.he_uniform(seed=0),return_sequences=True)

input_layer = Input(shape=(X_test_spectrogram[0].shape),dtype=float)
LSTM_layer  = lstm(inputs=input_layer)
glo_avg = tf.keras.layers.GlobalAveragePooling1D(data_format='channels_first')(LSTM_layer)
dense = Dense(50,activation="relu",kernel_initializer=tf.keras.initializers.he_uniform(seed=0))(glo_avg)
output_1 = Dense(10,activation='softmax',kernel_initializer=tf.keras.initializers.GlorotUniform(seed=0))(dense)

model = Model(inputs = [input_layer],outputs = output_1)
model.compile(optimizer='adam',loss = tf.keras.losses.sparse_categorical_crossentropy,metrics='accuracy')
model.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 64, 35)]          0
_________________________________________________________________
lstm (LSTM)                  (None, 64, 100)           54400
_________________________________________________________________
global_average_pooling1d (Gl (None, 64)                0
_________________________________________________________________
dense (Dense)                (None, 50)                3250
_________________________________________________________________
dense_1 (Dense)              (None, 10)                510
=================================================================
Total params: 58,160
Trainable params: 58,160
Non-trainable params: 0
_________________________________________________________________
```

In [42]:
```
X_train_spectrogram.shape , X_test_spectrogram.shape ,y_train.shape,y_test.shape
```

Out[42]:
```
((1400, 64, 35), (600, 64, 35), (1400,), (600,))
```

In [46]:
```
model.fit(x=X_train_spectrogram,y=y_train,validation_data=(X_test_spectrogram,y_test),
            epochs=60,batch_size=10,steps_per_epoch=len(X_train_spectrogram)//10, callbacks=metrics1)
```

```
Epoch 1/60
140/140 [==============================] - 1s 6ms/step - loss: 2.0538 - accuracy: 0.2607 - val_loss: 1.8639 - val_accuracy: 0.3667
 F1 micro : 0.36666666666666664
Epoch 2/60
140/140 [==============================] - 1s 6ms/step - loss: 1.6948 - accuracy: 0.4371 - val_loss: 1.5437 - val_accuracy: 0.4650
 F1 micro : 0.465
Epoch 3/60
140/140 [==============================] - 1s 6ms/step - loss: 1.4344 - accuracy: 0.5150 - val_loss: 1.4146 - val_accuracy: 0.5200
 F1 micro : 0.52
Epoch 4/60
140/140 [==============================] - 1s 6ms/step - loss: 1.2568 - accuracy: 0.6050 - val_loss: 1.1972 - val_accuracy: 0.6400
 F1 micro : 0.64
Epoch 5/60
140/140 [==============================] - 1s 6ms/step - loss: 1.1125 - accuracy: 0.6621 - val_loss: 1.0655 - val_accuracy: 0.6500
 F1 micro : 0.65
Epoch 6/60
140/140 [==============================] - 1s 6ms/step - loss: 1.0267 - accuracy: 0.6850 - val_loss: 0.9794 - val_accuracy: 0.7017
 F1 micro : 0.7016666666666667
Epoch 7/60
140/140 [==============================] - 1s 6ms/step - loss: 0.9289 - accuracy: 0.7179 - val_loss: 0.9059 - val_accuracy: 0.6983
 F1 micro : 0.6983333333333334
Epoch 8/60
140/140 [==============================] - 1s 6ms/step - loss: 0.8673 - accuracy: 0.7186 - val_loss: 0.8597 - val_accuracy: 0.7250
 F1 micro : 0.7250000000000001
Epoch 9/60
140/140 [==============================] - 1s 6ms/step - loss: 0.8162 - accuracy: 0.7457 - val_loss: 0.8058 - val_accuracy: 0.7267
 F1 micro : 0.7266666666666666
Epoch 10/60
140/140 [==============================] - 1s 6ms/step - loss: 0.7548 - accuracy: 0.7586 - val_loss: 0.7446 - val_accuracy: 0.7583
 F1 micro : 0.7583333333333333
Epoch 11/60
140/140 [==============================] - 1s 6ms/step - loss: 0.7242 - accuracy: 0.7679 - val_loss: 0.6986 - val_accuracy: 0.7717
 F1 micro : 0.7716666666666666
Epoch 12/60
```

```
140/140 [==============================] - 1s 6ms/step - loss: 0.6736 - accuracy: 0.7900 - val_loss: 0.6847 - val_accuracy: 0.7733
 F1 micro : 0.7733333333333333
Epoch 13/60
140/140 [==============================] - 1s 6ms/step - loss: 0.6345 - accuracy: 0.7929 - val_loss: 0.6253 - val_accuracy: 0.7867
 F1 micro : 0.7866666666666666
Epoch 14/60
140/140 [==============================] - 1s 6ms/step - loss: 0.6037 - accuracy: 0.8036 - val_loss: 0.6087 - val_accuracy: 0.7933
 F1 micro : 0.7933333333333333
Epoch 15/60
140/140 [==============================] - 1s 6ms/step - loss: 0.5647 - accuracy: 0.8129 - val_loss: 0.5805 - val_accuracy: 0.7967
 F1 micro : 0.7966666666666665
Epoch 16/60
140/140 [==============================] - 1s 6ms/step - loss: 0.5516 - accuracy: 0.8293 - val_loss: 0.5608 - val_accuracy: 0.8367
 F1 micro : 0.8366666666666667
Epoch 17/60
140/140 [==============================] - 1s 6ms/step - loss: 0.5386 - accuracy: 0.8243 - val_loss: 0.5734 - val_accuracy: 0.8000
 F1 micro : 0.8000000000000002
Epoch 18/60
140/140 [==============================] - 1s 6ms/step - loss: 0.5400 - accuracy: 0.8250 - val_loss: 0.5592 - val_accuracy: 0.8117
 F1 micro : 0.8116666666666666
Epoch 19/60
140/140 [==============================] - 1s 6ms/step - loss: 0.5071 - accuracy: 0.8436 - val_loss: 0.5199 - val_accuracy: 0.8317
 F1 micro : 0.8316666666666667
Epoch 20/60
140/140 [==============================] - 1s 6ms/step - loss: 0.4888 - accuracy: 0.8471 - val_loss: 0.4973 - val_accuracy: 0.8283
 F1 micro : 0.8283333333333334
Epoch 21/60
140/140 [==============================] - 1s 6ms/step - loss: 0.4693 - accuracy: 0.8443 - val_loss: 0.4851 - val_accuracy: 0.8300
 F1 micro : 0.83
Epoch 22/60
140/140 [==============================] - 1s 6ms/step - loss: 0.4542 - accuracy: 0.8564 - val_loss: 0.4855 - val_accuracy: 0.8517
 F1 micro : 0.8516666666666667
Epoch 23/60
140/140 [==============================] - 1s 6ms/step - loss: 0.4418 - accuracy: 0.8650 - val_loss: 0.4752 - val_accuracy: 0.8350
 F1 micro : 0.835
Epoch 24/60
140/140 [==============================] - 1s 6ms/step - loss: 0.4301 - accuracy: 0.8543 - val_loss: 0.4482 - val_accuracy: 0.8467
 F1 micro : 0.8466666666666667
Epoch 25/60
140/140 [==============================] - 1s 6ms/step - loss: 0.4242 - accuracy: 0.8671 - val_loss: 0.4477 - val_accuracy: 0.8467
 F1 micro : 0.8466666666666667
Epoch 26/60
140/140 [==============================] - 1s 6ms/step - loss: 0.4204 - accuracy: 0.8707 - val_loss: 0.4644 - val_accuracy: 0.8583
 F1 micro : 0.8583333333333333
Epoch 27/60
140/140 [==============================] - 1s 6ms/step - loss: 0.4177 - accuracy: 0.8686 - val_loss: 0.4378 - val_accuracy: 0.8650
 F1 micro : 0.865
Epoch 28/60
140/140 [==============================] - 1s 6ms/step - loss: 0.3931 - accuracy: 0.8786 - val_loss: 0.4369 - val_accuracy: 0.8633
 F1 micro : 0.8633333333333333
Epoch 29/60
140/140 [==============================] - 1s 6ms/step - loss: 0.3883 - accuracy: 0.8793 - val_loss: 0.4156 - val_accuracy: 0.8733
 F1 micro : 0.8733333333333333
Epoch 30/60
140/140 [==============================] - 1s 6ms/step - loss: 0.3859 - accuracy: 0.8771 - val_loss: 0.3922 - val_accuracy: 0.8733
 F1 micro : 0.8733333333333333
Epoch 31/60
140/140 [==============================] - 1s 6ms/step - loss: 0.3774 - accuracy: 0.8807 - val_loss: 0.4123 - val_accuracy: 0.8783
 F1 micro : 0.8783333333333333
Epoch 32/60
140/140 [==============================] - 1s 6ms/step - loss: 0.3770 - accuracy: 0.8736 - val_loss: 0.3629 - val_accuracy: 0.8900
 F1 micro : 0.89
Epoch 33/60
140/140 [==============================] - 1s 6ms/step - loss: 0.3379 - accuracy: 0.9000 - val_loss: 0.3725 - val_accuracy: 0.8900
 F1 micro : 0.89
Epoch 34/60
140/140 [==============================] - 1s 6ms/step - loss: 0.3364 - accuracy: 0.8971 - val_loss: 0.3777 - val_accuracy: 0.8817
 F1 micro : 0.8816666666666667
Epoch 35/60
140/140 [==============================] - 1s 6ms/step - loss: 0.3315 - accuracy: 0.8979 - val_loss: 0.3475 - val_accuracy: 0.8983
 F1 micro : 0.8983333333333333
Epoch 36/60
140/140 [==============================] - 1s 6ms/step - loss: 0.3286 - accuracy: 0.8979 - val_loss: 0.3576 - val_accuracy: 0.8883
 F1 micro : 0.8883333333333333
Epoch 37/60
140/140 [==============================] - 1s 6ms/step - loss: 0.3082 - accuracy: 0.9107 - val_loss: 0.3873 - val_accuracy: 0.8583
 F1 micro : 0.8583333333333333
Epoch 38/60
140/140 [==============================] - 1s 6ms/step - loss: 0.3180 - accuracy: 0.8993 - val_loss: 0.3869 - val_accuracy: 0.8933
 F1 micro : 0.8933333333333333
Epoch 39/60
140/140 [==============================] - 1s 6ms/step - loss: 0.3207 - accuracy: 0.9007 - val_loss: 0.3316 - val_accuracy: 0.8983
 F1 micro : 0.8983333333333333
Epoch 40/60
140/140 [==============================] - 1s 6ms/step - loss: 0.3148 - accuracy: 0.9050 - val_loss: 0.3363 - val_accuracy: 0.9000
 F1 micro : 0.9
Epoch 41/60
140/140 [==============================] - 1s 6ms/step - loss: 0.2906 - accuracy: 0.9150 - val_loss: 0.3336 - val_accuracy: 0.8917
 F1 micro : 0.8916666666666667
Epoch 42/60
140/140 [==============================] - 1s 6ms/step - loss: 0.2780 - accuracy: 0.9171 - val_loss: 0.2993 - val_accuracy: 0.9100
 F1 micro : 0.91
Epoch 43/60
140/140 [==============================] - 1s 6ms/step - loss: 0.2847 - accuracy: 0.9057 - val_loss: 0.3606 - val_accuracy: 0.8817
 F1 micro : 0.8816666666666667
Epoch 44/60
140/140 [==============================] - 1s 6ms/step - loss: 0.2912 - accuracy: 0.9071 - val_loss: 0.3240 - val_accuracy: 0.8983
 F1 micro : 0.8983333333333333
```

```
Epoch 45/60
140/140 [==============================] - 1s 6ms/step - loss: 0.2671 - accuracy: 0.9214 - val_loss: 0.2853 - val_accuracy: 0.9133
 F1 micro : 0.9133333333333333
Epoch 46/60
140/140 [==============================] - 1s 6ms/step - loss: 0.2599 - accuracy: 0.9164 - val_loss: 0.2975 - val_accuracy: 0.9000
 F1 micro : 0.9
Epoch 47/60
140/140 [==============================] - 1s 6ms/step - loss: 0.2591 - accuracy: 0.9214 - val_loss: 0.2813 - val_accuracy: 0.9167
 F1 micro : 0.9166666666666666
Epoch 48/60
140/140 [==============================] - 1s 6ms/step - loss: 0.2534 - accuracy: 0.9271 - val_loss: 0.2804 - val_accuracy: 0.9117
 F1 micro : 0.9116666666666666
Epoch 49/60
140/140 [==============================] - 1s 6ms/step - loss: 0.2514 - accuracy: 0.9286 - val_loss: 0.2836 - val_accuracy: 0.9100
 F1 micro : 0.91
Epoch 50/60
140/140 [==============================] - 1s 6ms/step - loss: 0.2437 - accuracy: 0.9279 - val_loss: 0.2961 - val_accuracy: 0.8967
 F1 micro : 0.8966666666666666
Epoch 51/60
140/140 [==============================] - 1s 6ms/step - loss: 0.2474 - accuracy: 0.9286 - val_loss: 0.3063 - val_accuracy: 0.9083
 F1 micro : 0.9083333333333333
Epoch 52/60
140/140 [==============================] - 1s 6ms/step - loss: 0.2448 - accuracy: 0.9307 - val_loss: 0.2897 - val_accuracy: 0.9167
 F1 micro : 0.9166666666666666
Epoch 53/60
140/140 [==============================] - 1s 6ms/step - loss: 0.2465 - accuracy: 0.9186 - val_loss: 0.3150 - val_accuracy: 0.8883
 F1 micro : 0.8883333333333333
Epoch 54/60
140/140 [==============================] - 1s 6ms/step - loss: 0.2229 - accuracy: 0.9307 - val_loss: 0.2740 - val_accuracy: 0.9183
 F1 micro : 0.9183333333333333
Epoch 55/60
140/140 [==============================] - 1s 6ms/step - loss: 0.2376 - accuracy: 0.9300 - val_loss: 0.2672 - val_accuracy: 0.9200
 F1 micro : 0.92
Epoch 56/60
140/140 [==============================] - 1s 6ms/step - loss: 0.2298 - accuracy: 0.9336 - val_loss: 0.2918 - val_accuracy: 0.9133
 F1 micro : 0.9133333333333333
Epoch 57/60
140/140 [==============================] - 1s 6ms/step - loss: 0.2443 - accuracy: 0.9286 - val_loss: 0.3041 - val_accuracy: 0.9133
 F1 micro : 0.9133333333333333
Epoch 58/60
140/140 [==============================] - 1s 6ms/step - loss: 0.2385 - accuracy: 0.9357 - val_loss: 0.2733 - val_accuracy: 0.9050
 F1 micro : 0.905
Epoch 59/60
140/140 [==============================] - 1s 6ms/step - loss: 0.2299 - accuracy: 0.9250 - val_loss: 0.2711 - val_accuracy: 0.9183
 F1 micro : 0.9183333333333333
Epoch 60/60
140/140 [==============================] - 1s 6ms/step - loss: 0.2178 - accuracy: 0.9371 - val_loss: 0.2602 - val_accuracy: 0.9133
 F1 micro : 0.9133333333333333
```

Out[46]:
```
<tensorflow.python.keras.callbacks.History at 0x7f88f0185150>
```

Now we have

Train data: X_train_spectrogram and y_train
Test data: X_test_spectrogram and y_test

We will create a LSTM model which takes this input.

Task:

1. Create an LSTM network which takes "X_train_spectrogram" as input and has to return output at every time step.
2. Average the output of every time step and give this to the Dense layer of any size.
(ex: Output from LSTM will be (#., time_steps, features) average the output of every time step i.e, you should get (#.,time_steps)
and then pass to dense layer)
3. give the above output to Dense layer of size 10(output layer) and train the network with sparse categorical cross entropy.
4. Use tensorboard to plot the graphs of loss and metric(use micro F1 score as metric) and histograms of gradients.
5. make sure that it won't overfit.
6. You are free to include any regularization

3. data augmentation

Till now we have done with 2000 samples only. It is very less data. We are giving the process of generating augmented data below.

There are two types of augmentation:
1. time stretching - Time stretching either increases or decreases the length of the file. For time stretching we move the file 30% faster or slower
2. pitch shifting - pitch shifting moves the frequencies higher or lower. For pitch shifting we shift up or down one half-step.

In [47]:
```python
## generating augmented data.
def generate_augmented_data(file_path):
    augmented_data = []
    samples = load_wav(file_path,get_duration=False)
    for time_value in [0.7, 1, 1.3]:
        for pitch_value in [-1, 0, 1]:
            time_stretch_data = librosa.effects.time_stretch(samples, rate=time_value)
            final_data = librosa.effects.pitch_shift(time_stretch_data, sr=sample_rate, n_steps=pitch_value)
```

```
            augmented_data.append(final_data)
    return augmented_data
```

In [48]:

```
x_train_data_aug = []

for path in X_train:
  aug_temp = generate_augmented_data(path)
  x_train_data_aug.extend(aug_temp)

x_train_data_aug = np.array(x_train_data_aug)

y_train_data_aug = []

for i in y_train:
  c = [i]*9
  y_train_data_aug.extend(c)

y_train_data_aug = np.array(y_train_data_aug)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:7: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequenc
es (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant to do this, you
must specify 'dtype=object' when creating the ndarray
  import sys
```

In [49]:

```
X_train_pad_seq1 = []
for li in x_train_data_aug:
  if len(li) < max_length:
    li = li.tolist()
    a = [0]*(max_length - len(li))
    li.extend(a)
    X_train_pad_seq1.append(li)

  else:
    li = li.tolist()
    X_train_pad_seq1.append(li[0:max_length])

X_train_pad_seq1 = np.array(X_train_pad_seq1)
```

In [50]:

```
X_train_mask1 = np.array([(i > 0).tolist()for i in X_train_pad_seq1])
```

MODEL :- 3

In [51]:

```
## as discussed above, please write the LSTM

lstm = LSTM(units = 100,activation="tanh",kernel_initializer=tf.keras.initializers.he_uniform(seed=0))

input_layer = Input(shape=(X_train_pad_seq.shape[1],1),dtype=float)
input_mask = Input(shape=(X_train_mask.shape[1],1),dtype=bool)
LSTM_layer  = lstm(inputs=input_layer,mask=input_mask)
dense = Dense(50,activation="relu",kernel_initializer=tf.keras.initializers.he_uniform(seed=0))(LSTM_layer)
output_1 = Dense(10,activation='softmax',kernel_initializer=tf.keras.initializers.GlorotUniform(seed=0))(dense)

model = Model(inputs = [input_layer,input_mask],outputs = output_1)
model.compile(optimizer='adam',loss = tf.keras.losses.sparse_categorical_crossentropy,metrics='accuracy')
model.summary()
```

```
Model: "model_2"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_3 (InputLayer)            [(None, 17640, 1)]   0
__________________________________________________________________________________________________
input_4 (InputLayer)            [(None, 17640, 1)]   0
__________________________________________________________________________________________________
lstm_2 (LSTM)                   (None, 100)          40800       input_3[0][0]
                                                                 input_4[0][0]
__________________________________________________________________________________________________
dense_4 (Dense)                 (None, 50)           5050        lstm_2[0][0]
__________________________________________________________________________________________________
dense_5 (Dense)                 (None, 10)           510         dense_4[0][0]
==================================================================================================
Total params: 46,360
Trainable params: 46,360
Non-trainable params: 0
__________________________________________________________________________________________________
```

In [52]:

```
X_train_pad_seq1.shape, X_train_mask1.shape, y_train_data_aug.shape
```

Out[52]:

```
((12600, 17640), (12600, 17640), (12600,))
```

In [53]:

```
X_test_pad_seq.shape,X_test_mask.shape,y_test.shape
```

Out[53]:

```
((600, 17640), (600, 17640), (600,))
```

In [54]:

```
model.fit(x=[X_train_pad_seq1,X_train_mask1],y=y_train_data_aug,validation_data=([X_test_pad_seq,X_test_mask],y_test),
            epochs=2,batch_size=10,steps_per_epoch=len(X_train_mask)//10 , callbacks=metrics)
```

```
Epoch 1/2
140/140 [==============================] - 44s 298ms/step - loss: 2.3046 - accuracy: 0.1150 - val_loss: 2.3017 - val_accuracy: 0.1033
```

```
 F1 micro : 0.10000000000000002
Epoch 2/2
140/140 [==============================] - 41s 291ms/step - loss: 2.3012 - accuracy: 0.1155 - val_loss: 2.2994 - val_accuracy: 0.1250
 F1 micro : 0.10000000000000002
```

Out[54]: `<tensorflow.python.keras.callbacks.History at 0x7f89097e5ed0>`

As discussed above, for one data point, we will get 9 augmented data points.

Split data into train and test (80-20 split)

We have 2000 data points(1600 train points, 400 test points)

Do augmentation only on train data, after augmentation we will get 14400 train points.

do the above steps i.e training with raw data and spectrogram data with augmentation.

MODEL :- 4

In [55]:
```
X_train_spectrogram1 = [convert_to_spectrogram(row) for row in X_train_pad_seq1]
X_train_spectrogram1 = np.array(X_train_spectrogram1)

X_test_spectrogram = [convert_to_spectrogram(row) for row in X_test_pad_seq]
X_test_spectrogram = np.array(X_test_spectrogram)
```

In [56]:
```
## as discussed above, please write the LSTM

lstm = LSTM(units = 100,activation="tanh",kernel_initializer=tf.keras.initializers.he_uniform(seed=0),return_sequences=True)

input_layer = Input(shape=(X_test_spectrogram[0].shape),dtype=float)
LSTM_layer  = lstm(inputs=input_layer)
glo_avg = tf.keras.layers.GlobalAveragePooling1D(data_format='channels_first')(LSTM_layer)
dense = Dense(50,activation="relu",kernel_initializer=tf.keras.initializers.he_uniform(seed=0))(glo_avg)
output_1 = Dense(10,activation='softmax',kernel_initializer=tf.keras.initializers.GlorotUniform(seed=0))(dense)

model = Model(inputs = [input_layer],outputs = output_1)
model.compile(optimizer='adam',loss = tf.keras.losses.sparse_categorical_crossentropy,metrics='accuracy')
model.summary()
```

```
Model: "model_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_5 (InputLayer)         [(None, 64, 35)]          0         
_________________________________________________________________
lstm_3 (LSTM)                (None, 64, 100)           54400     
_________________________________________________________________
global_average_pooling1d_2 ( (None, 64)                0         
_________________________________________________________________
dense_6 (Dense)              (None, 50)                3250      
_________________________________________________________________
dense_7 (Dense)              (None, 10)                510       
=================================================================
Total params: 58,160
Trainable params: 58,160
Non-trainable params: 0
_________________________________________________________________
```

In [57]:
```
X_train_spectrogram1.shape , X_test_spectrogram.shape ,y_train_data_aug.shape,y_test.shape
```

Out[57]: `((12600, 64, 35), (600, 64, 35), (12600,), (600,))`

In [59]:
```
model.fit(x=X_train_spectrogram1,y=y_train_data_aug,validation_data=(X_test_spectrogram,y_test),
            epochs=60,batch_size=10,steps_per_epoch=len(X_train_spectrogram1)//10, callbacks=metrics1)
```

```
Epoch 1/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.8588 - accuracy: 0.7082 - val_loss: 0.6784 - val_accuracy: 0.7700
 F1 micro : 0.7699999999999999
Epoch 2/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.6708 - accuracy: 0.7670 - val_loss: 0.5384 - val_accuracy: 0.8433
 F1 micro : 0.8433333333333335
Epoch 3/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.5804 - accuracy: 0.7985 - val_loss: 0.5305 - val_accuracy: 0.8183
 F1 micro : 0.8183333333333332
Epoch 4/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.5177 - accuracy: 0.8254 - val_loss: 0.4310 - val_accuracy: 0.8733
 F1 micro : 0.8733333333333333
Epoch 5/60
1260/1260 [==============================] - 6s 4ms/step - loss: 0.4789 - accuracy: 0.8363 - val_loss: 0.4122 - val_accuracy: 0.8667
 F1 micro : 0.8666666666666667
Epoch 6/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.4582 - accuracy: 0.8421 - val_loss: 0.3508 - val_accuracy: 0.9067
 F1 micro : 0.9066666666666666
Epoch 7/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.4234 - accuracy: 0.8554 - val_loss: 0.3487 - val_accuracy: 0.8983
 F1 micro : 0.8983333333333333
Epoch 8/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.4096 - accuracy: 0.8614 - val_loss: 0.3308 - val_accuracy: 0.9000
 F1 micro : 0.9
Epoch 9/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3868 - accuracy: 0.8648 - val_loss: 0.3121 - val_accuracy: 0.8983
 F1 micro : 0.8983333333333333
```

```
Epoch 10/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3816 - accuracy: 0.8696 - val_loss: 0.2856 - val_accuracy: 0.9200
 F1 micro : 0.92
Epoch 11/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3710 - accuracy: 0.8729 - val_loss: 0.2847 - val_accuracy: 0.9167
 F1 micro : 0.9166666666666666
Epoch 12/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3742 - accuracy: 0.8723 - val_loss: 0.3288 - val_accuracy: 0.8883
 F1 micro : 0.8883333333333333
Epoch 13/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3735 - accuracy: 0.8704 - val_loss: 0.3156 - val_accuracy: 0.9167
 F1 micro : 0.9166666666666666
Epoch 14/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3634 - accuracy: 0.8752 - val_loss: 0.2834 - val_accuracy: 0.9200
 F1 micro : 0.92
Epoch 15/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3566 - accuracy: 0.8784 - val_loss: 0.3203 - val_accuracy: 0.8933
 F1 micro : 0.8933333333333333
Epoch 16/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3524 - accuracy: 0.8794 - val_loss: 0.3073 - val_accuracy: 0.9050
 F1 micro : 0.905
Epoch 17/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3625 - accuracy: 0.8770 - val_loss: 0.2614 - val_accuracy: 0.9267
 F1 micro : 0.9266666666666666
Epoch 18/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3465 - accuracy: 0.8799 - val_loss: 0.2728 - val_accuracy: 0.9217
 F1 micro : 0.9216666666666666
Epoch 19/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3411 - accuracy: 0.8832 - val_loss: 0.2924 - val_accuracy: 0.9000
 F1 micro : 0.9
Epoch 20/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3458 - accuracy: 0.8840 - val_loss: 0.3417 - val_accuracy: 0.8833
 F1 micro : 0.8833333333333333
Epoch 21/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3417 - accuracy: 0.8807 - val_loss: 0.3261 - val_accuracy: 0.8850
 F1 micro : 0.885
Epoch 22/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3372 - accuracy: 0.8880 - val_loss: 0.2792 - val_accuracy: 0.9100
 F1 micro : 0.91
Epoch 23/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3488 - accuracy: 0.8816 - val_loss: 0.2817 - val_accuracy: 0.9133
 F1 micro : 0.9133333333333333
Epoch 24/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3355 - accuracy: 0.8818 - val_loss: 0.3114 - val_accuracy: 0.8933
 F1 micro : 0.8933333333333333
Epoch 25/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3308 - accuracy: 0.8885 - val_loss: 0.2801 - val_accuracy: 0.9017
 F1 micro : 0.9016666666666667
Epoch 26/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3323 - accuracy: 0.8853 - val_loss: 0.2433 - val_accuracy: 0.9233
 F1 micro : 0.9233333333333333
Epoch 27/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3227 - accuracy: 0.8880 - val_loss: 0.2594 - val_accuracy: 0.9250
 F1 micro : 0.925
Epoch 28/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3321 - accuracy: 0.8860 - val_loss: 0.2433 - val_accuracy: 0.9250
 F1 micro : 0.925
Epoch 29/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3150 - accuracy: 0.8901 - val_loss: 0.2713 - val_accuracy: 0.9117
 F1 micro : 0.9116666666666666
Epoch 30/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3080 - accuracy: 0.8915 - val_loss: 0.2531 - val_accuracy: 0.9200
 F1 micro : 0.92
Epoch 31/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3283 - accuracy: 0.8872 - val_loss: 0.2568 - val_accuracy: 0.9183
 F1 micro : 0.9183333333333333
Epoch 32/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3217 - accuracy: 0.8850 - val_loss: 0.2574 - val_accuracy: 0.9217
 F1 micro : 0.9216666666666666
Epoch 33/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3013 - accuracy: 0.8945 - val_loss: 0.2621 - val_accuracy: 0.9100
 F1 micro : 0.91
Epoch 34/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3002 - accuracy: 0.8937 - val_loss: 0.2990 - val_accuracy: 0.9000
 F1 micro : 0.9
Epoch 35/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3237 - accuracy: 0.8867 - val_loss: 0.3120 - val_accuracy: 0.9017
 F1 micro : 0.9016666666666667
Epoch 36/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3382 - accuracy: 0.8806 - val_loss: 0.2423 - val_accuracy: 0.9200
 F1 micro : 0.92
Epoch 37/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3277 - accuracy: 0.8874 - val_loss: 0.2859 - val_accuracy: 0.9017
 F1 micro : 0.9016666666666667
Epoch 38/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3227 - accuracy: 0.8868 - val_loss: 0.3133 - val_accuracy: 0.8983
 F1 micro : 0.8983333333333333
Epoch 39/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3225 - accuracy: 0.8893 - val_loss: 0.2612 - val_accuracy: 0.9150
 F1 micro : 0.915
Epoch 40/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3208 - accuracy: 0.8894 - val_loss: 0.2501 - val_accuracy: 0.9200
 F1 micro : 0.92
Epoch 41/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3137 - accuracy: 0.8908 - val_loss: 0.2897 - val_accuracy: 0.8917
 F1 micro : 0.8916666666666667
Epoch 42/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3168 - accuracy: 0.8898 - val_loss: 0.2742 - val_accuracy: 0.9017
```

```
 F1 micro : 0.9016666666666667
Epoch 43/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3085 - accuracy: 0.8944 - val_loss: 0.2851 - val_accuracy: 0.8983
 F1 micro : 0.8983333333333333
Epoch 44/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3218 - accuracy: 0.8852 - val_loss: 0.2854 - val_accuracy: 0.9050
 F1 micro : 0.905
Epoch 45/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3122 - accuracy: 0.8933 - val_loss: 0.2476 - val_accuracy: 0.9300
 F1 micro : 0.93
Epoch 46/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3165 - accuracy: 0.8887 - val_loss: 0.2942 - val_accuracy: 0.9017
 F1 micro : 0.9016666666666667
Epoch 47/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3293 - accuracy: 0.8815 - val_loss: 0.3269 - val_accuracy: 0.8950
 F1 micro : 0.895
Epoch 48/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3031 - accuracy: 0.8945 - val_loss: 0.2534 - val_accuracy: 0.9133
 F1 micro : 0.9133333333333333
Epoch 49/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3108 - accuracy: 0.8884 - val_loss: 0.3208 - val_accuracy: 0.8933
 F1 micro : 0.8933333333333333
Epoch 50/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3004 - accuracy: 0.8960 - val_loss: 0.2774 - val_accuracy: 0.9083
 F1 micro : 0.9083333333333333
Epoch 51/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3290 - accuracy: 0.8819 - val_loss: 0.2377 - val_accuracy: 0.9233
 F1 micro : 0.9233333333333333
Epoch 52/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3024 - accuracy: 0.8936 - val_loss: 0.2176 - val_accuracy: 0.9333
 F1 micro : 0.9333333333333333
Epoch 53/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.3064 - accuracy: 0.8891 - val_loss: 0.2896 - val_accuracy: 0.8917
 F1 micro : 0.8916666666666667
Epoch 54/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.2920 - accuracy: 0.8966 - val_loss: 0.2735 - val_accuracy: 0.8950
 F1 micro : 0.895
Epoch 55/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.2871 - accuracy: 0.8989 - val_loss: 0.2077 - val_accuracy: 0.9283
 F1 micro : 0.9283333333333333
Epoch 56/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.2881 - accuracy: 0.8986 - val_loss: 0.2770 - val_accuracy: 0.9083
 F1 micro : 0.9083333333333333
Epoch 57/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.2837 - accuracy: 0.9013 - val_loss: 0.2557 - val_accuracy: 0.9117
 F1 micro : 0.9116666666666666
Epoch 58/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.2888 - accuracy: 0.8971 - val_loss: 0.2583 - val_accuracy: 0.9117
 F1 micro : 0.9116666666666666
Epoch 59/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.2925 - accuracy: 0.8969 - val_loss: 0.2464 - val_accuracy: 0.9017
 F1 micro : 0.9016666666666667
Epoch 60/60
1260/1260 [==============================] - 6s 5ms/step - loss: 0.2817 - accuracy: 0.8991 - val_loss: 0.2276 - val_accuracy: 0.9283
 F1 micro : 0.9283333333333333
```

Out[59]:
```
<tensorflow.python.keras.callbacks.History at 0x7f88b4e99b90>
```

In [ ]: