University of Tehran
Electrical and Computer Engineering Department
Neural Networks and Deep Learning
Mini-Project 2

Name	Danial Saeedi
Student No	810198571
Date	Saturday - 2022 08 January

Table of contents

1. Question 1: Music Generation

Part 1

Note: A note is your basic building block of music. A note is one key on a piano. It's pretty hard to define by itself.

Chord: A chord is 3 or more notes played together at the same time. For example, a C Major chord consists of 3 notes, C E and G. It can be any group of 3 different notes, some just wouldn't sound good.

Loading Dataset

In [6]:
```
# Frédéric Chopin
path = "/content/drive/MyDrive/MP2_Dataset/Classical_Music_MIDI/"
#Getting midi files
midi_list = []
for i in os.listdir(path):
  if i.endswith(".mid"):
    tr = path+i
    midi = converter.parse(tr)
    midi_list.append(midi)
```

Extracting Notes

In [8]:
```
corpus = extract_notes(midi_list)
```

In [9]:
```
print("Total notes:", len(corpus))
```

```
Total notes: 63429
```

Part 2: Data Exploration & Pre-processing

Create a count dictionary

In [10]:
```
count = Counter(corpus)
print("Unique notes:", len(count))
```

```
Unique notes: 317
```

Exploring the notes dictionary

In [11]:
```
def Average(lst):
    return sum(lst) / len(lst)
```

In [12]:
```
notes = list(count.keys())
frequency = list(count.values())
```

In [13]:
```
print("Avg frequency for a note:", Average(frequency))
print("Most frequent note appeared:", max(frequency), "times")
print("Least frequent note appeared:", min(frequency), "time")
```

```
Avg frequency for a note: 200.09148264984228
Most frequent note appeared: 1869 times
Least frequent note appeared: 1 time
```

Plotting Distribution of Notes

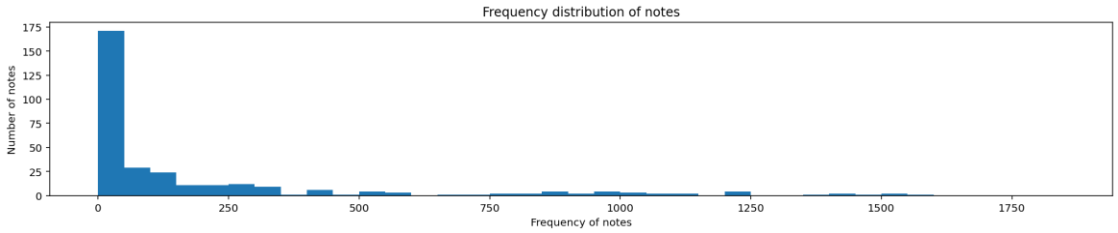

Figure 1 Frequency Distribution of notes

Finding Rare Notes

Let's find the list of rare notes (less than 80 recurrence) and remove them.

In [15]:
```
rare_note_occurrence = 80
rare_note = []
for index, (key, value) in enumerate(count.items()):
  if value < rare_note_occurrence:
    rare_note.append(key)

# Number of occurrence of notes less than 80
print(len(rare_note))
```

```
193
```

Creating Training/Test Dataset

We'll create a list of sorted unique symbols:

In [17]:
```
symbols = sorted(list(set(corpus)))

symbols[90:100]
```

Out[17]:
```
['5.8.0',
 '5.8.10',
 '5.9',
 '5.9.0',
 '5.9.11',
 '6',
 '6.10',
 '6.10.0',
 '6.10.1',
 '6.11']
```

The length of corpus and number of unique symbols are stored at length_corpus and lenght_symbols respectively:

In [18]:
```
length_corpus = len(corpus)
length_symbols = len(symbols)
```

And then we'll build a dictionary of symbols and their indices:

In [19]:
```
dictionary = dict((c, i) for i, c in enumerate(symbols))
reverse_dictionary = dict((i, c) for i, c in enumerate(symbols))
```

In [20]:
```
dictionary
```

```
 'F1': 201,
 'F2': 202,
 'F3': 203,
 'F4': 204,
 'F5': 205,
 'F6': 206,
 'F7': 207,
 'G#1': 208,
 'G#2': 209,
 'G#3': 210,
 'G#4': 211,
 'G#5': 212,
 'G#6': 213,
 'G1': 214,
 'G2': 215,
 'G3': 216,
 'G4': 217,
 'G5': 218,
 'G6': 219}
```

Train/Test Split

Here we will split the corpus in equal length of strings and output target:(Length of each sequence = 40 based on project description)

In [21]:
```
sequence_length = 40
features = []
targets = []
for i in range(0, length_corpus - sequence_length, 1):
    feature = corpus[i:i + sequence_length]
    target = corpus[i + sequence_length]
    features.append([dictionary[j] for j in feature])
    targets.append(dictionary[target])
```

In [22]:
```
length_datapoints = len(targets)
print("Total # sequences in the corpus:", length_datapoints)
```

```
Total # sequences in the corpus: 60318
```

Normalize data

In [23]:
```
shape = (length_datapoints, sequence_length, 1)
X = np.reshape(features, shape)
X = X/float(length_symbols)
```

One-hot encoding for target

In [24]:
```
y = tensorflow.keras.utils.to_categorical(targets)
```

In [25]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Part 3: Defining the model

In [26]:
```
import tensorflow as tf
model = Sequential()

in_shape = (X.shape[1], X.shape[2])

model.add(LSTM(1024, input_shape=in_shape, return_sequences=True))
model.add(LSTM(512))
model.add(Dense(512))

#Output
model.add(Dense(y.shape[1], activation='softmax'))

optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
model.compile(loss='categorical_crossentropy', optimizer=optimizer)
```

In [27]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 40, 1024)          4202496

 lstm_1 (LSTM)               (None, 512)               3147776

 dense (Dense)               (None, 512)               262656

 dense_1 (Dense)             (None, 220)               112860

=================================================================
Total params: 7,725,788
Trainable params: 7,725,788
Non-trainable params: 0
_________________________________________________________________
```

Part 4: Training

```
Epoch 14/30
189/189 [==============================] - 17s 88ms/step - loss: 3.4207 - val_loss: 3.5880
Epoch 15/30
189/189 [==============================] - 17s 88ms/step - loss: 3.1637 - val_loss: 3.4587
Epoch 16/30
189/189 [==============================] - 17s 89ms/step - loss: 2.8802 - val_loss: 3.3241
Epoch 17/30
189/189 [==============================] - 17s 88ms/step - loss: 2.5808 - val_loss: 3.2465
Epoch 18/30
189/189 [==============================] - 17s 88ms/step - loss: 2.2824 - val_loss: 3.1789
Epoch 19/30
189/189 [==============================] - 17s 88ms/step - loss: 2.0015 - val_loss: 3.1411
Epoch 20/30
189/189 [==============================] - 17s 88ms/step - loss: 1.7397 - val_loss: 3.1621
Epoch 21/30
189/189 [==============================] - 17s 88ms/step - loss: 1.4861 - val_loss: 3.1602
Epoch 22/30
189/189 [==============================] - 17s 88ms/step - loss: 1.2478 - val_loss: 3.2677
Epoch 23/30
189/189 [==============================] - 17s 88ms/step - loss: 1.0608 - val_loss: 3.3454
Epoch 24/30
189/189 [==============================] - 17s 88ms/step - loss: 0.8786 - val_loss: 3.4677
Epoch 25/30
189/189 [==============================] - 17s 88ms/step - loss: 0.7313 - val_loss: 3.5588
Epoch 26/30
189/189 [==============================] - 17s 88ms/step - loss: 0.6007 - val_loss: 3.7400
Epoch 27/30
189/189 [==============================] - 17s 88ms/step - loss: 0.5001 - val_loss: 3.9036
Epoch 28/30
189/189 [==============================] - 17s 88ms/step - loss: 0.4218 - val_loss: 4.0830
Epoch 29/30
189/189 [==============================] - 17s 88ms/step - loss: 0.3405 - val_loss: 4.2181
Epoch 30/30
189/189 [==============================] - 17s 88ms/step - loss: 0.2800 - val_loss: 4.4383
CPU times: user 5min 38s, sys: 8.51 s, total: 5min 46s
Wall time: 8min 28s
```

Figure 2 Running Time and Loss per epoch

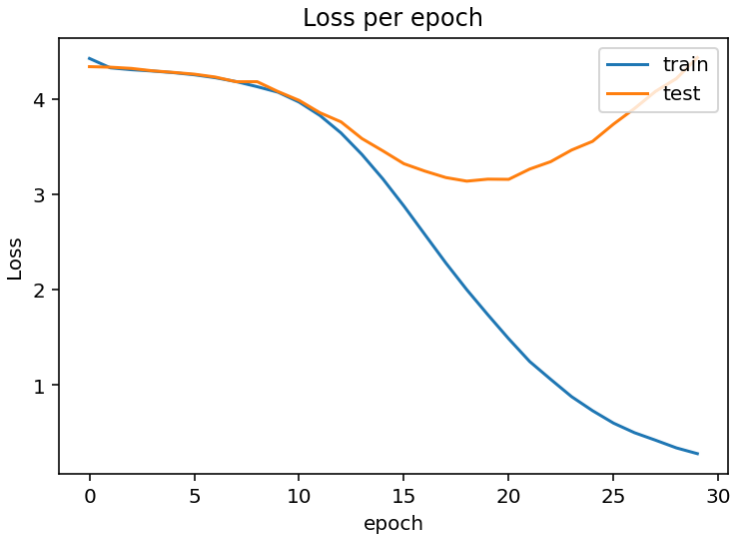

Figure 3 Loss per epoch

Generating multiple melodies

The generated melodies are stored at **generated** folder.

Training with Mozart songs and adding Dropout Layer

```
In [32]: # Mozart
path = "/content/drive/MyDrive/MP2_Dataset/mozart/"
#Getting midi files
midi_list = []
for i in os.listdir(path):
  if i.endswith(".mid"):
    tr = path+i
    midi = converter.parse(tr)
    midi_list.append(midi)
```

Extracting Notes

```
In [33]: corpus = extract_notes(midi_list)
```

```
In [34]: print("Total notes:", len(corpus))

Total notes: 59618
```

Data Exploration & Pre-processing for Mozart Dataset

Create a count dictionary

```
In [35]: count = Counter(corpus)
print("Unique notes:", len(count))

Unique notes: 197
```

Exploring the notes dictionary

In [36]:
```
notes = list(count.keys())
frequency = list(count.values())
```

In [37]:
```
print("Avg frequency for a note:", Average(frequency))
print("Most frequent note appeared:", max(frequency), "times")
print("Least frequent note appeared:", min(frequency), "time")
```

```
Avg frequency for a note: 302.6294416243655
Most frequent note appeared: 3116 times
Least frequent note appeared: 1 time
```

Plotting the distribution of notes

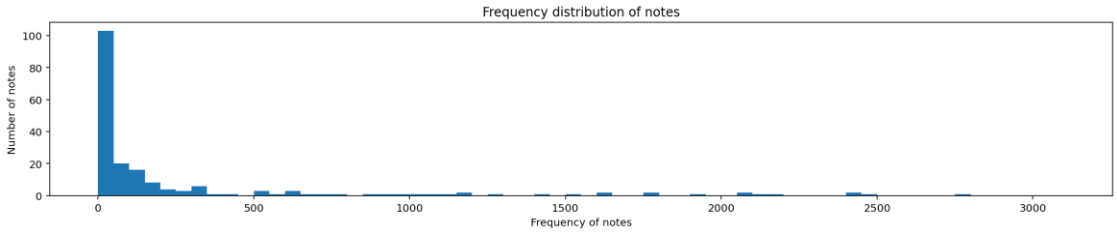

Figure 4 Frequency distribution notes in Mozart

Finding Rare Notes

In [39]:
```
rare_note_occurrence = 80
rare_note = []
for index, (key, value) in enumerate(count.items()):
  if value < rare_note_occurrence:
    rare_note.append(key)

# Number of occurrence of notes less than 80
print(len(rare_note))
```

```
121
```

Creating Training/Test Dataset

This part is similar to Chopin.

Define model

In [50]:
```
model = Sequential()

in_shape = (X.shape[1], X.shape[2])

model.add(LSTM(1024, input_shape=in_shape, return_sequences=True))
model.add(Dropout(0.5))
model.add(LSTM(512))
model.add(Dropout(0.5))
model.add(Dense(512))
model.add(Dropout(0.5))

#Output
model.add(Dense(y.shape[1], activation='softmax'))

optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
model.compile(loss='categorical_crossentropy', optimizer=optimizer)
```

Plotting training/validation loss per epoch

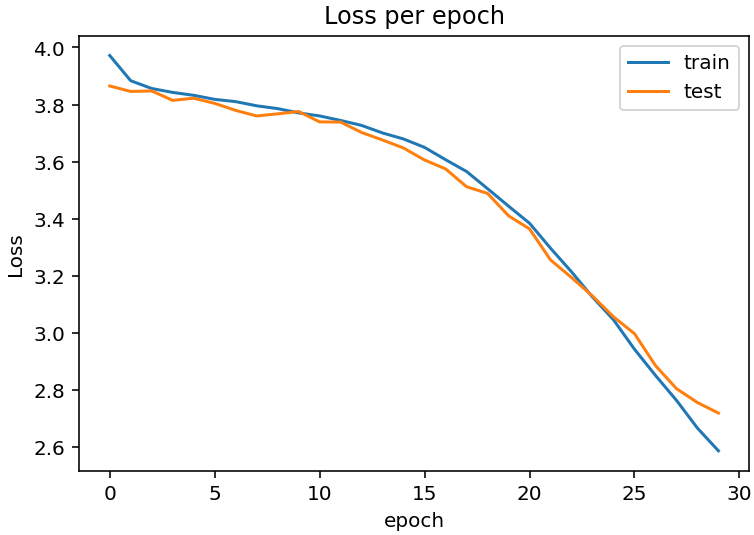

Figure 5 Loss per epoch with Dropout Layer

The model's performance with Dropout Layers was significantly better

2. Question 2: Pitch/Note Detection

Part 1: Converting MIDI files to WAV

Converting MIDI files to WAV has been done with Convert_MIDI_to_WAV.py.

```
!unzip /content/drive/MyDrive/Output.zip
```

Plotting

Y and X axes are **Amplitude** and **Time** respectively.

Difference: Overall, 1_0.wav has higher amplitude than 1_12.wav which means 1_0 has higher pitch.

Similarity: The rise and fall of sound wave in both 1_0.wav and 1_2.wav are **simultaneous**.

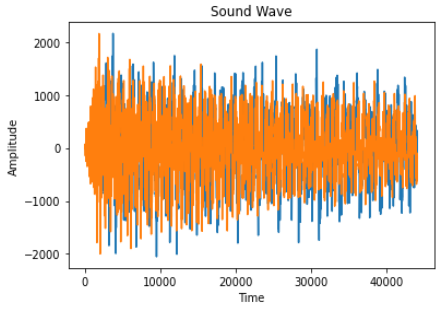

Figure 6 Plotting 1_0.wav and 1_12.wav

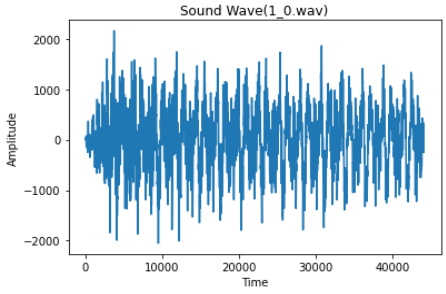

Figure 7 Plotting 1_0.wav

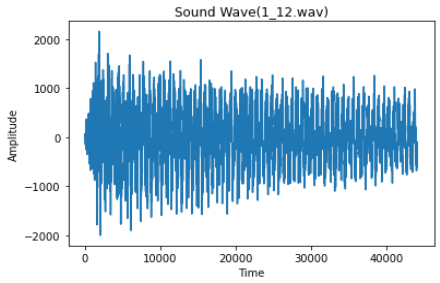

Figure 8 Plotting 1_12.wav

Part 2: Train/Validation/Test Split

There are 108 notes for each instrument. We have to split the dataset such that every instrument is represented in training set. For every instrument we select 80%, 20% and 10% of notes randomly for training Train, Validation and Test set respectively. This way it is guaranteed that the dataset is balanced for training.

Note that if we do it randomly on all files, training set could be **Imbalanced**.

1. Training Size: 9952 (70%)
2. Validation Size: 2489 (20%)
3. Test Size: 1383 (10%)

Out[182]:

	instrument	path	data_type	note_0	note_1	note_2	note_3	note_4	note_5	note_6	note_7	note_8	note_9	note_10	note_11	note_1
0	0	converted/0_0.wav	train	1	0	0	0	0	0	0	0	0	0	0	0	
1	0	converted/0_1.wav	train	0	1	0	0	0	0	0	0	0	0	0	0	
2	0	converted/0_2.wav	train	0	0	1	0	0	0	0	0	0	0	0	0	
3	0	converted/0_3.wav	train	0	0	0	1	0	0	0	0	0	0	0	0	
4	0	converted/0_4.wav	train	0	0	0	0	1	0	0	0	0	0	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
13819	127	converted/127_103.wav	val	0	0	0	0	0	0	0	0	0	0	0	0	
13820	127	converted/127_104.wav	train	0	0	0	0	0	0	0	0	0	0	0	0	
13821	127	converted/127_105.wav	train	0	0	0	0	0	0	0	0	0	0	0	0	
13822	127	converted/127_106.wav	test	0	0	0	0	0	0	0	0	0	0	0	0	
13823	127	converted/127_107.wav	val	0	0	0	0	0	0	0	0	0	0	0	0	

13824 rows × 112 columns

Figure 9 Data Frame After Train/Val/Test Split

Part 3: Implementing Model

Part A

Training time for RNN was the highest among these models with the lowest accuracy.

But GRU **training time** was significantly **better** with higher **accuracy**.

Model	Training Time	Validation Loss
RNN	70 min	4.6953
LSTM	27 min	4.50
GRU	12 min	4.44

RNN

In [ ]:
```
# define model
model = Sequential()

# Simple RNN layer
model.add(SimpleRNN(32,input_shape=(input_250ms_shape, 1)))

# Final Prediction
model.add(Dense(notes,activation='softmax'))

model.compile(optimizer='adam', loss='categorical_crossentropy')
```

In [ ]:
```
model.summary()
```

```
Model: "sequential_8"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 simple_rnn_8 (SimpleRNN)    (None, 32)                1088

 dense_9 (Dense)             (None, 108)               3564

=================================================================
Total params: 4,652
Trainable params: 4,652
Non-trainable params: 0
_________________________________________________________________
```

```
Epoch 1/10
78/78 [==============================] - 420s 5s/step - loss: 4.6827 - val_loss: 4.6930
Epoch 2/10
78/78 [==============================] - 419s 5s/step - loss: 4.6861 - val_loss: 4.6850
Epoch 3/10
78/78 [==============================] - 420s 5s/step - loss: 4.6862 - val_loss: 4.6855
Epoch 4/10
78/78 [==============================] - 419s 5s/step - loss: 4.6820 - val_loss: 4.6868
Epoch 5/10
78/78 [==============================] - 418s 5s/step - loss: 4.6828 - val_loss: 4.6943
Epoch 6/10
78/78 [==============================] - 419s 5s/step - loss: 4.6886 - val_loss: 4.7020
Epoch 7/10
78/78 [==============================] - 420s 5s/step - loss: 4.6928 - val_loss: 4.6919
Epoch 8/10
78/78 [==============================] - 421s 5s/step - loss: 4.6886 - val_loss: 4.6936
Epoch 9/10
78/78 [==============================] - 418s 5s/step - loss: 4.6879 - val_loss: 4.6961
Epoch 10/10
78/78 [==============================] - 417s 5s/step - loss: 4.6869 - val_loss: 4.6953
CPU times: user 1h 57min, sys: 12min 9s, total: 2h 9min 9s
Wall time: 1h 10min 23s
```

Figure 10 Run time and loss per epoch

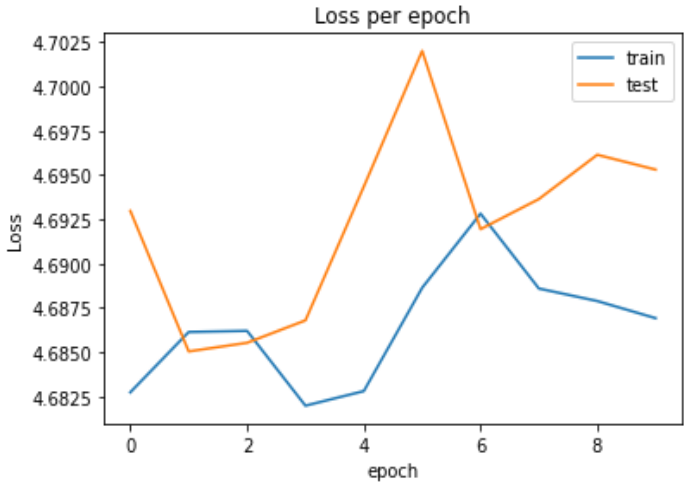

Figure 11 Loss per epoch for RNN

In [ ]:
```
# define model
model = Sequential()

# LSTM layer
model.add(LSTM(128,input_shape=(input_250ms_shape, 1)))

# Final Prediction
model.add(Dense(notes,activation='softmax'))

model.compile(optimizer='adam', loss='categorical_crossentropy')
```

In [ ]:
```
model.summary()
```

```
Model: "sequential_11"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_2 (LSTM)               (None, 128)               66560

 dense_12 (Dense)            (None, 108)               13932

=================================================================
Total params: 80,492
Trainable params: 80,492
Non-trainable params: 0
_________________________________________________________________
```

```
Epoch 1/20
311/311 [==============================] - 86s 266ms/step - loss: 4.6838 - val_loss: 4.6840
Epoch 2/20
311/311 [==============================] - 82s 264ms/step - loss: 4.6819 - val_loss: 4.6848
Epoch 3/20
311/311 [==============================] - 79s 254ms/step - loss: 4.6838 - val_loss: 4.6857
Epoch 4/20
311/311 [==============================] - 78s 250ms/step - loss: 4.7161 - val_loss: 4.7449
Epoch 5/20
311/311 [==============================] - 79s 255ms/step - loss: 4.7275 - val_loss: 4.7198
Epoch 6/20
311/311 [==============================] - 83s 266ms/step - loss: 4.7140 - val_loss: 4.7104
Epoch 7/20
311/311 [==============================] - 79s 255ms/step - loss: 4.7051 - val_loss: 4.7153
Epoch 8/20
311/311 [==============================] - 79s 254ms/step - loss: 4.7061 - val_loss: 4.7010
Epoch 9/20
311/311 [==============================] - 79s 255ms/step - loss: 4.7037 - val_loss: 4.7018
Epoch 10/20
311/311 [==============================] - 83s 266ms/step - loss: 4.7007 - val_loss: 4.7021
Epoch 11/20
311/311 [==============================] - 79s 255ms/step - loss: 4.7006 - val_loss: 4.6967
Epoch 12/20
311/311 [==============================] - 83s 265ms/step - loss: 4.6994 - val_loss: 4.7004
Epoch 13/20
311/311 [==============================] - 82s 265ms/step - loss: 4.6984 - val_loss: 4.6994
Epoch 14/20
311/311 [==============================] - 79s 255ms/step - loss: 4.6982 - val_loss: 4.7011
Epoch 15/20
311/311 [==============================] - 79s 255ms/step - loss: 4.6975 - val_loss: 4.6975
Epoch 16/20
311/311 [==============================] - 79s 255ms/step - loss: 4.6950 - val_loss: 4.7020
Epoch 17/20
311/311 [==============================] - 83s 266ms/step - loss: 4.6953 - val_loss: 4.7006
Epoch 18/20
311/311 [==============================] - 79s 255ms/step - loss: 4.6939 - val_loss: 4.6983
Epoch 19/20
311/311 [==============================] - 80s 256ms/step - loss: 4.6938 - val_loss: 4.6982
Epoch 20/20
311/311 [==============================] - 80s 256ms/step - loss: 4.6906 - val_loss: 4.6952
CPU times: user 20min 48s, sys: 5min 42s, total: 26min 30s
Wall time: 26min 50s
```

Figure 12 Run time and loss per epoch

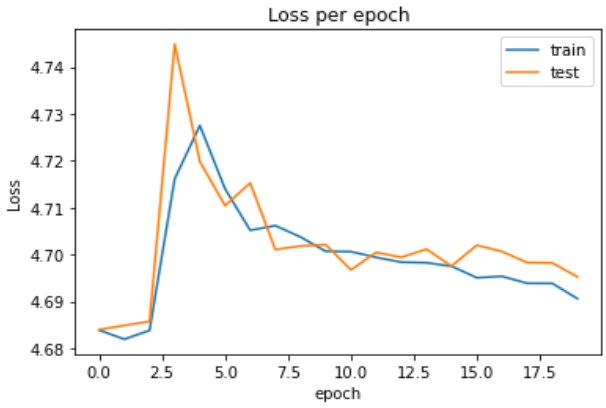

Figure 13 Loss per epoch for LSTM model

In [ ]:
```
# define model
model = Sequential()

# Simple RNN layer
model.add(GRU(128,input_shape=(input_250ms_shape, 1)))

# Final Prediction
model.add(Dense(notes,activation='softmax'))

model.compile(optimizer='adam', loss='categorical_crossentropy')
```

In [ ]:
```
model.summary()
```

```
Model: "sequential_12"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 gru (GRU)                   (None, 128)               50304

 dense_13 (Dense)            (None, 108)               13932

=================================================================
Total params: 64,236
Trainable params: 64,236
Non-trainable params: 0
_________________________________________________________________
```

```
Epoch 1/10
311/311 [==============================] - 73s 229ms/step - loss: 4.6833 - val_loss: 4.6832
Epoch 2/10
311/311 [==============================] - 75s 240ms/step - loss: 4.6821 - val_loss: 4.6845
Epoch 3/10
311/311 [==============================] - 71s 227ms/step - loss: 4.6737 - val_loss: 4.6643
Epoch 4/10
311/311 [==============================] - 70s 227ms/step - loss: 4.6622 - val_loss: 4.6564
Epoch 5/10
311/311 [==============================] - 71s 227ms/step - loss: 4.5812 - val_loss: 4.5444
Epoch 6/10
311/311 [==============================] - 71s 227ms/step - loss: 4.5121 - val_loss: 4.5082
Epoch 7/10
311/311 [==============================] - 71s 228ms/step - loss: 4.4841 - val_loss: 4.4865
Epoch 8/10
311/311 [==============================] - 71s 227ms/step - loss: 4.4657 - val_loss: 4.4699
Epoch 9/10
311/311 [==============================] - 71s 227ms/step - loss: 4.4515 - val_loss: 4.4586
Epoch 10/10
311/311 [==============================] - 71s 227ms/step - loss: 4.4396 - val_loss: 4.4482
CPU times: user 9min 38s, sys: 2min 13s, total: 11min 51s
Wall time: 11min 52s
```

Figure 14 Run time and loss per epoch

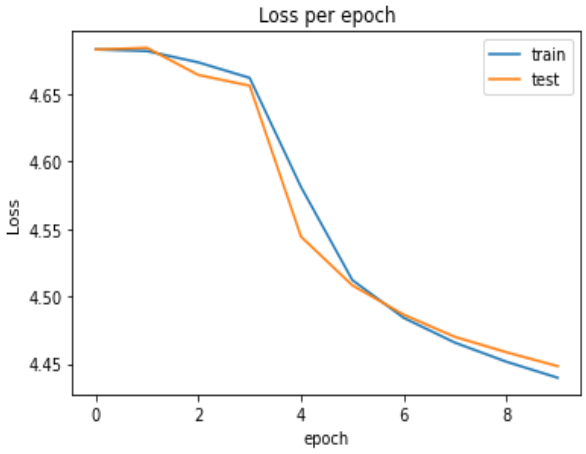

Figure 15 Loss per epoch for GRU

Part B

Adding Dropout layer to LSTM reduced training time and validation loss. But adding Dropout layer to RNN and GRU made the overall performance worse.

Model	Training Time	Validation Loss
RNN	3 hours!	4.6975
LSTM	13 min	4.47
GRU	13 min	4.69

Adding Dropout to RNN

In [ ]:
```
# define model
model = Sequential()

# Simple RNN layer
model.add(SimpleRNN(32,input_shape=(input_250ms_shape, 1)))
# Dropout Layer
model.add(Dropout(0.5))
# Final Prediction
model.add(Dense(notes,activation='softmax'))

model.compile(optimizer='adam', loss='categorical_crossentropy')
model.summary()
```

```
Model: "sequential_13"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 simple_rnn_9 (SimpleRNN)    (None, 32)                1088

 dropout (Dropout)           (None, 32)                0

 dense_14 (Dense)            (None, 108)               3564

=================================================================
Total params: 4,652
Trainable params: 4,652
Non-trainable params: 0
_________________________________________________________________
```

```
Epoch 1/10
311/311 [==============================] - 1156s 4s/step - loss: 4.6880 - val_loss: 4.6867
Epoch 2/10
311/311 [==============================] - 1148s 4s/step - loss: 4.7106 - val_loss: 4.6961
Epoch 3/10
311/311 [==============================] - 1147s 4s/step - loss: 4.7138 - val_loss: 4.6917
Epoch 4/10
311/311 [==============================] - 1132s 4s/step - loss: 4.7540 - val_loss: 4.6963
Epoch 5/10
311/311 [==============================] - 1121s 4s/step - loss: 4.7568 - val_loss: 4.7012
Epoch 6/10
311/311 [==============================] - 1115s 4s/step - loss: 4.7463 - val_loss: 4.7006
Epoch 7/10
311/311 [==============================] - 1107s 4s/step - loss: 4.7377 - val_loss: 4.6962
Epoch 8/10
311/311 [==============================] - 1108s 4s/step - loss: 4.7298 - val_loss: 4.7015
Epoch 9/10
311/311 [==============================] - 1107s 4s/step - loss: 4.7318 - val_loss: 4.6943
Epoch 10/10
311/311 [==============================] - 1107s 4s/step - loss: 4.7235 - val_loss: 4.6975
CPU times: user 4h 54min 15s, sys: 27min 20s, total: 5h 21min 35s
Wall time: 3h 8min 23s
```

Figure 16 Run time and loss per epoch

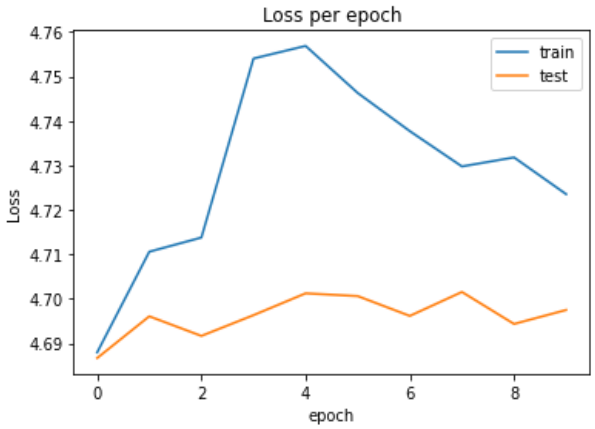

Figure 17 Loss per epoch for RNN with Dropout

Adding Dropout to LSTM

In [ ]:
```
# define model
model = Sequential()

# Simple RNN layer
model.add(LSTM(128,input_shape=(input_250ms_shape, 1)))
# Dropout Layer
model.add(Dropout(0.5))
# Final Prediction
model.add(Dense(notes,activation='softmax'))

model.compile(optimizer='adam', loss='categorical_crossentropy')

model.summary()
```

```
Model: "sequential_14"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_3 (LSTM)               (None, 128)               66560

 dropout_1 (Dropout)         (None, 128)               0

 dense_15 (Dense)            (None, 108)               13932

=================================================================
Total params: 80,492
Trainable params: 80,492
Non-trainable params: 0
_________________________________________________________________
```

```
Epoch 1/10
311/311 [==============================] - 79s 249ms/step - loss: 4.6832 - val_loss: 4.6841
Epoch 2/10
311/311 [==============================] - 77s 248ms/step - loss: 4.6817 - val_loss: 4.6844
Epoch 3/10
311/311 [==============================] - 77s 248ms/step - loss: 4.6804 - val_loss: 4.6887
Epoch 4/10
311/311 [==============================] - 77s 248ms/step - loss: 4.6816 - val_loss: 4.6872
Epoch 5/10
311/311 [==============================] - 77s 248ms/step - loss: 4.6772 - val_loss: 4.6073
Epoch 6/10
311/311 [==============================] - 78s 251ms/step - loss: 4.6080 - val_loss: 4.6873
Epoch 7/10
311/311 [==============================] - 78s 251ms/step - loss: 4.6444 - val_loss: 4.5948
Epoch 8/10
311/311 [==============================] - 78s 252ms/step - loss: 4.6011 - val_loss: 4.6968
Epoch 9/10
311/311 [==============================] - 78s 251ms/step - loss: 4.6060 - val_loss: 4.4704
Epoch 10/10
311/311 [==============================] - 79s 254ms/step - loss: 4.7035 - val_loss: 4.7526
CPU times: user 10min 9s, sys: 2min 47s, total: 12min 57s
Wall time: 12min 59s
```

Figure 18 Loss per epoch and run time

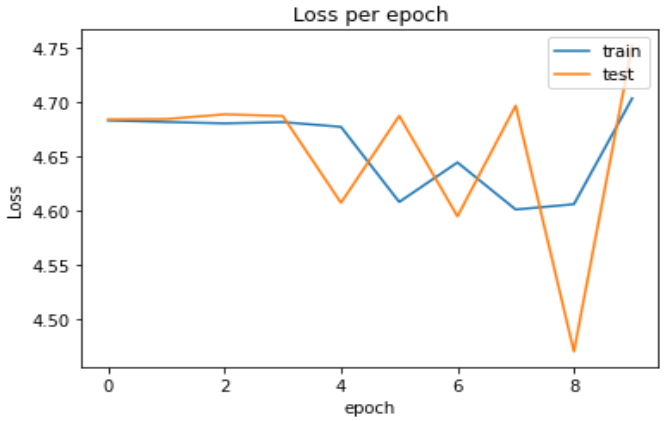

Figure 19 Loss per epoch

Adding Dropout to GRU

In [ ]:
```
# define model
model = Sequential()

# Simple RNN layer
model.add(GRU(128,input_shape=(input_250ms_shape, 1)))
# Dropout Layer
model.add(Dropout(0.5))
# Final Prediction
model.add(Dense(notes,activation='softmax'))

model.compile(optimizer='adam', loss='categorical_crossentropy')

model.summary()
```

```
Model: "sequential_15"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_4 (LSTM)               (None, 128)               66560

 dropout_2 (Dropout)         (None, 128)               0

 dense_16 (Dense)            (None, 108)               13932

=================================================================
Total params: 80,492
Trainable params: 80,492
Non-trainable params: 0
_________________________________________________________________
```

```
Epoch 1/10
311/311 [==============================] - 80s 250ms/step - loss: 4.6833 - val_loss: 4.6838
Epoch 2/10
311/311 [==============================] - 81s 260ms/step - loss: 4.6830 - val_loss: 4.6800
Epoch 3/10
311/311 [==============================] - 77s 247ms/step - loss: 4.6817 - val_loss: 4.6866
Epoch 4/10
311/311 [==============================] - 81s 260ms/step - loss: 4.6791 - val_loss: 4.6740
Epoch 5/10
311/311 [==============================] - 78s 252ms/step - loss: 4.7292 - val_loss: 4.7196
Epoch 6/10
311/311 [==============================] - 83s 266ms/step - loss: 4.7431 - val_loss: 4.6950
Epoch 7/10
311/311 [==============================] - 79s 254ms/step - loss: 4.7252 - val_loss: 4.6971
Epoch 8/10
311/311 [==============================] - 79s 255ms/step - loss: 4.7134 - val_loss: 4.6959
Epoch 9/10
311/311 [==============================] - 79s 255ms/step - loss: 4.7344 - val_loss: 4.7040
Epoch 10/10
311/311 [==============================] - 79s 254ms/step - loss: 4.7251 - val_loss: 4.6947
CPU times: user 10min 12s, sys: 2min 50s, total: 13min 3s
Wall time: 13min 16s
```

Figure 20 Run time and loss per epoch

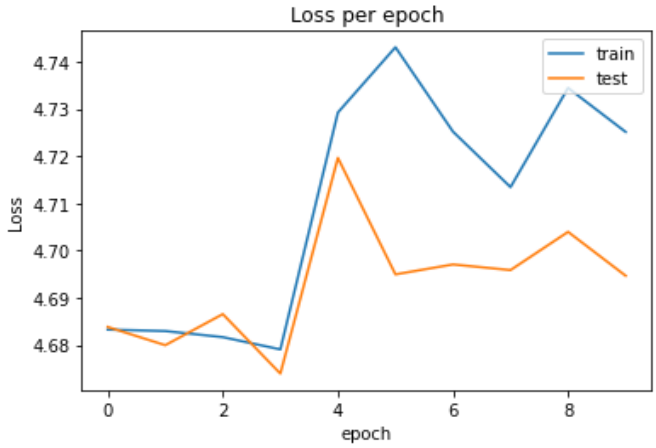

Figure 21 Loss per epoch

Part C: Trying different time frame

Note that increasing time frame made the model's performance better. Because with small time frame, the model is unable to learn the patterns. The best time frame is 250ms.

Time Frame	Loss
35 ms	4.6790
70 ms	4.69
150 ms	4.7
300 ms	4.5

Time Frame = 35ms

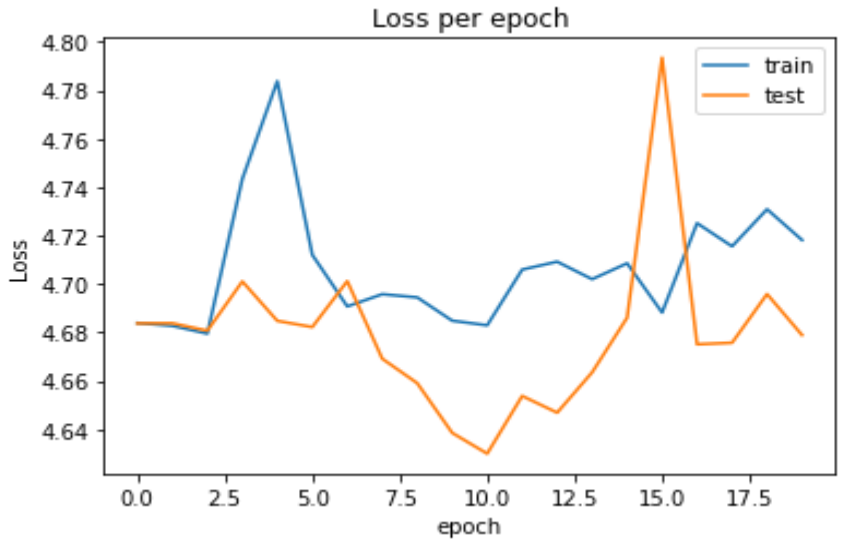

Figure 22 Loss per epoch

Time Frame = 70ms

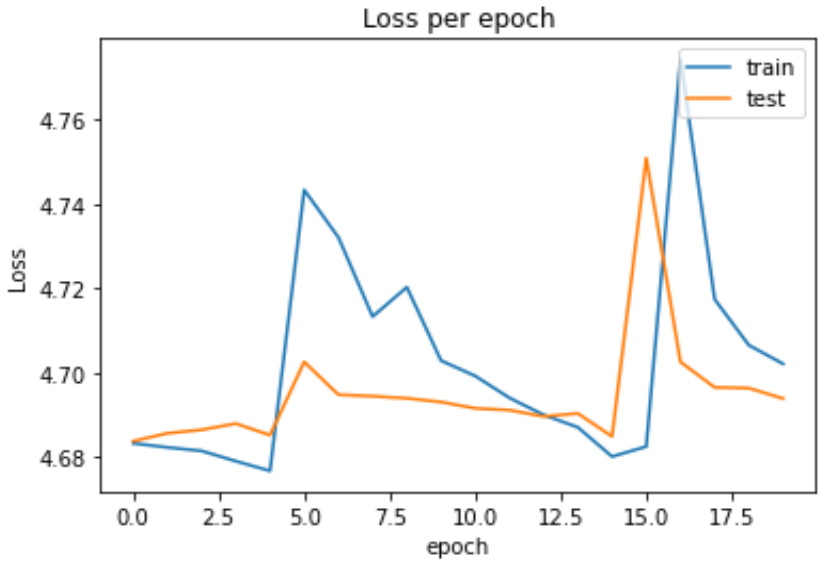

Figure 23 Loss per epoch

Time Frame = 150ms

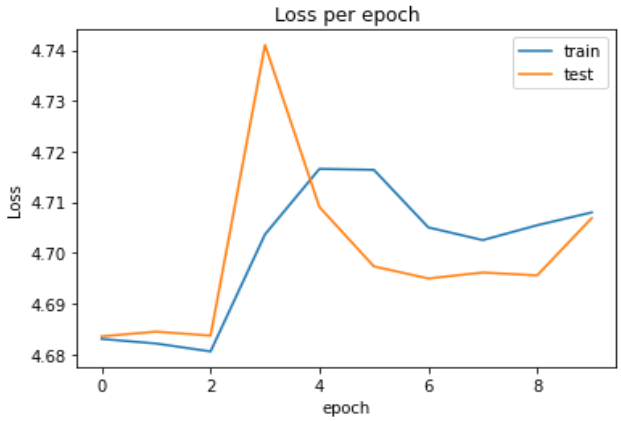

Figure 24 Loss per epoch

Time Frame = 300ms

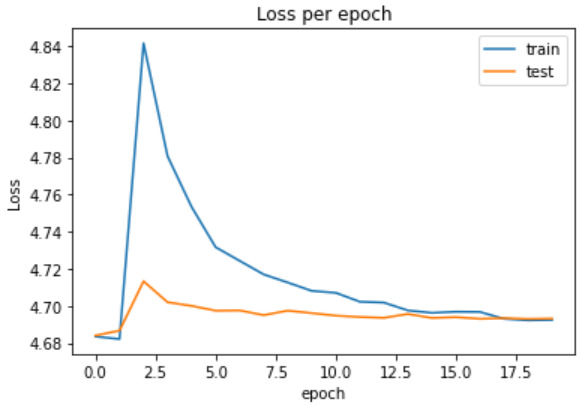

Figure 25 Loss per epoch

Part D: Extract Features

In this part, we're going to use **MFCC**. Feature Extraction made model's performance **significantly better**. MFCC removes **noise** and **unnecessary** signals.

1. Zero Crossing Rate

The zero crossing rate is the rate of sign-changes along a signal, i.e., the rate at which the signal changes from positive to negative or back. This feature has been used heavily in both speech recognition and music information retrieval. It usually has higher values for highly percussive sounds like those in metal and rock.

2. Spectral Centroid

It indicates where the ”centre of mass” for a sound is located and is calculated as the weighted mean of the frequencies present in the sound. If the frequencies in music are same throughout then spectral centroid would be around a centre and if there are high frequencies at the end of sound then the centroid would be towards its end.

3.MFCC — Mel-Frequency Cepstral Coefficients

This feature is one of the most important method to extract a feature of an audio signal and is used majorly whenever working on audio signals. The mel frequency cepstral coefficients (MFCCs) of a signal are a small set of features (usually about 10–20) which concisely describe the overall shape of a spectral envelope.

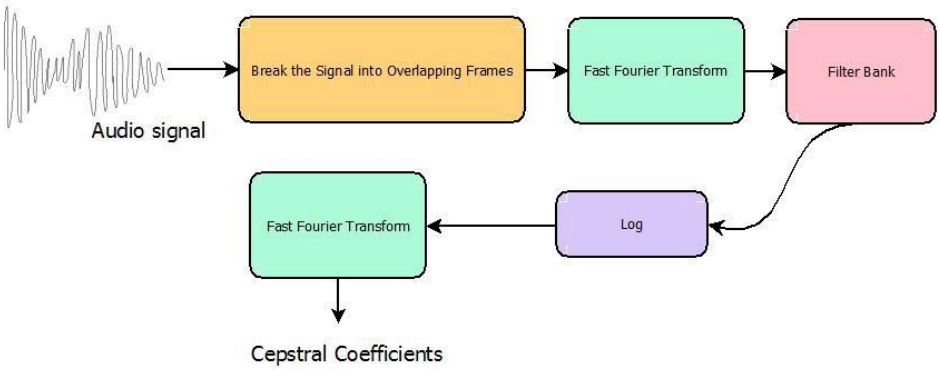

Figure 26 How MFCC works

In [33]:
```
def features_extractor(file,time_frame):
    audio, sample_rate = librosa.load(file, res_type='kaiser_fast',duration=time_frame)
    mfccs_features = librosa.feature.mfcc(y=audio, sr=sample_rate, n_mfcc=40)
    mfccs_scaled_features = np.mean(mfccs_features.T,axis=0)

    return mfccs_scaled_features.tolist()
```

In [69]:
```
X_train_features = []
X_val_features = []
X_test_features = []

time_frame  = 0.250
for train_path in X_train_path.tolist():
  data = features_extractor(train_path,time_frame)
  X_train_features.append(data)

for val_path in X_val_path.tolist():
  data = features_extractor(val_path,time_frame)
  X_val_features.append(data)

for test_path in X_test_path.tolist():
  data = features_extractor(test_path,time_frame)
  X_test_features.append(data)
```

```
311/311 [==============================] - 2s 6ms/step - loss: 0.9246 - val_loss: 1.5231
Epoch 33/50
311/311 [==============================] - 2s 6ms/step - loss: 0.9653 - val_loss: 1.6530
Epoch 34/50
311/311 [==============================] - 2s 6ms/step - loss: 0.9608 - val_loss: 1.5236
Epoch 35/50
311/311 [==============================] - 2s 6ms/step - loss: 0.8878 - val_loss: 1.4947
Epoch 36/50
311/311 [==============================] - 2s 6ms/step - loss: 0.8499 - val_loss: 1.5180
Epoch 37/50
311/311 [==============================] - 2s 6ms/step - loss: 0.8677 - val_loss: 1.5567
Epoch 38/50
311/311 [==============================] - 2s 6ms/step - loss: 0.8671 - val_loss: 1.5299
Epoch 39/50
311/311 [==============================] - 2s 6ms/step - loss: 0.8256 - val_loss: 1.5295
Epoch 40/50
311/311 [==============================] - 2s 6ms/step - loss: 0.8070 - val_loss: 1.5159
Epoch 41/50
311/311 [==============================] - 2s 6ms/step - loss: 0.7817 - val_loss: 1.5428
Epoch 42/50
311/311 [==============================] - 2s 6ms/step - loss: 0.7705 - val_loss: 1.5356
Epoch 43/50
311/311 [==============================] - 2s 6ms/step - loss: 0.7757 - val_loss: 1.4979
Epoch 44/50
311/311 [==============================] - 2s 6ms/step - loss: 0.7501 - val_loss: 1.5384
Epoch 45/50
311/311 [==============================] - 2s 6ms/step - loss: 0.8226 - val_loss: 1.5552
Epoch 46/50
311/311 [==============================] - 2s 6ms/step - loss: 0.8055 - val_loss: 1.5528
Epoch 47/50
311/311 [==============================] - 2s 6ms/step - loss: 0.7415 - val_loss: 1.5446
Epoch 48/50
311/311 [==============================] - 2s 6ms/step - loss: 0.7113 - val_loss: 1.5232
Epoch 49/50
311/311 [==============================] - 2s 6ms/step - loss: 0.6896 - val_loss: 1.5982
Epoch 50/50
311/311 [==============================] - 2s 6ms/step - loss: 0.7244 - val_loss: 1.5302
CPU times: user 1min 33s, sys: 7.64 s, total: 1min 41s
Wall time: 1min 29s
```

Figure 27 Run time and loss per epoch

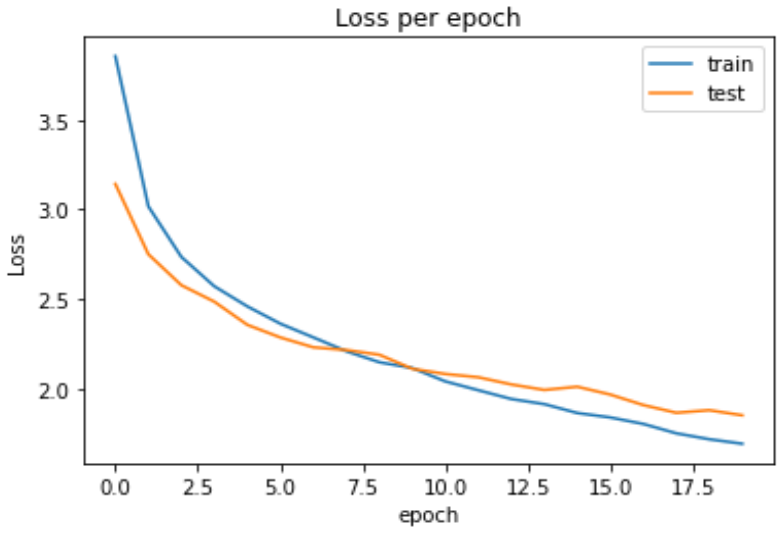

Figure 28 Loss per epoch

	precision	recall	f1-score	support
accuracy			0.56	1383
macro avg	0.58	0.56	0.55	1383
weighted avg	0.60	0.56	0.56	1383

Figure 29 Classification Report

Part E: Data Augmentation

1. **Noise Injection:** It simply add some random value into data by using numpy.
2. **Shifting Time:**The idea of shifting time is very simple. It just shifts audio to left/right with a random second.
3. **Changing Speed:** It stretches times series by a fixed rate.

In this part we're going to use **Noise Injection** method for data augmentation. By applying noise injection precision of the model became 2% better.

```
In [85]: def manipulate(data, noise_factor):
             noise = np.random.randn(len(data))
             augmented_data = data + noise_factor * noise
             # Cast back to same data type
             augmented_data = augmented_data.astype(type(data[0]))
             return augmented_data
```

```
In [89]: def features_extractor_noise(file,time_frame):
             audio, sample_rate = librosa.load(file, res_type='kaiser_fast',duration=time_frame)
             audio = manipulate(librosa_audio_data,1)
             mfccs_features = librosa.feature.mfcc(y=audio, sr=sample_rate, n_mfcc=40)
             mfccs_scaled_features = np.mean(mfccs_features.T,axis=0)

             return mfccs_scaled_features.tolist()
```

```
In [86]: librosa_audio_data,librosa_sample_rate=librosa.load('converted/0_0.wav',duration=0.25)
```

```
In [88]: manipulate(librosa_audio_data,1)
```

```
Out[88]: array([ 0.85480744,  0.6420496 ,  0.5167906 , ..., -1.0347534 ,
                -0.39788714,  1.2686436 ], dtype=float32)
```

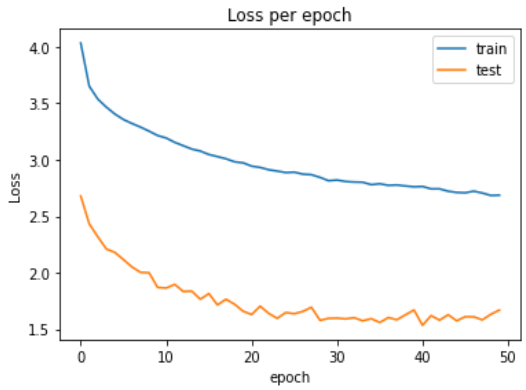

Figure 30 Loss per epoch

	precision	recall	f1-score	support
accuracy			0.56	1383
macro avg	0.59	0.55	0.55	1383
weighted avg	0.62	0.56	0.56	1383

Figure 31 Classification Report

Part F: Using CNN and RNN

Combining CNN and RNN made the precision of the **model 6%** better.

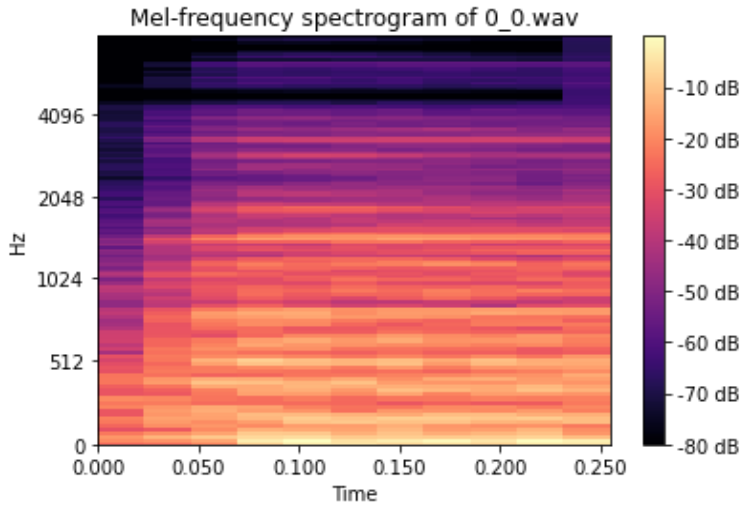

Figure 32 Spectrogram of 0_0.wav

In [167]:
```
# Create CNN/RNN model
model = Sequential()
model.add(TimeDistributedConvolution2D(64, 3, 3, border_mode='same', input_shape=(IMAGE_WIDTH, IMAGE_HEIGHT,
                                                                                  N_CHANNELS)))
model.add(Activation('relu'))
model.add(TimeDistributedConvolution2D(64, 3, 3, border_mode='same'))
model.add(Activation('relu'))
model.add(TimeDistributedMaxPooling2D(pool_size=(2, 2)))
model.add(Activation('relu'))
model.add(TimeDistributedConvolution2D(128, 3, 3, border_mode='same'))
model.add(Activation('relu'))
model.add(TimeDistributedConvolution2D(128, 3, 3))
model.add(Activation('relu'))
model.add(TimeDistributedMaxPooling2D(pool_size=(2, 2)))
model.add(Activation('relu'))
model.add(TimeDistributedConvolution2D(128, 3, 3, border_mode='same'))
model.add(Activation('relu'))
model.add(TimeDistributedConvolution2D(128, 3, 3))
model.add(Activation('relu'))
model.add(TimeDistributedMaxPooling2D(pool_size=(2, 2)))
model.add(Activation('relu'))
model.add(TimeDistributedConvolution2D(256, 3, 3, border_mode='same'))
model.add(Activation('relu'))
model.add(TimeDistributedConvolution2D(256, 3, 3))
model.add(Activation('relu'))
model.add(TimeDistributedMaxPooling2D(pool_size=(2, 2)))
model.add(Activation('relu'))
model.add(TimeDistributedFlatten())
model.add(LSTM(512, return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(512, return_sequences=True))
#Output
model.add(TimeDistributedDense(notes))
model.add(Activation('softmax'))

# Compile model
model.compile(
    loss='categorical_crossentropy',
    optimizer='adam'
)
```

```
Epoch 27/50
311/311 [==============================] - 1s 5ms/step - loss: 0.7544 - val_loss: 2.1111
Epoch 28/50
311/311 [==============================] - 2s 5ms/step - loss: 0.7265 - val_loss: 2.1887
Epoch 29/50
311/311 [==============================] - 1s 5ms/step - loss: 0.7231 - val_loss: 2.9088
Epoch 30/50
311/311 [==============================] - 1s 5ms/step - loss: 0.7221 - val_loss: 2.2725
Epoch 31/50
311/311 [==============================] - 1s 5ms/step - loss: 0.7137 - val_loss: 2.4225
Epoch 32/50
311/311 [==============================] - 2s 5ms/step - loss: 0.7011 - val_loss: 3.9371
Epoch 33/50
311/311 [==============================] - 2s 5ms/step - loss: 0.6804 - val_loss: 4.6255
Epoch 34/50
311/311 [==============================] - 2s 5ms/step - loss: 0.6843 - val_loss: 1.9302
Epoch 35/50
311/311 [==============================] - 1s 5ms/step - loss: 0.6715 - val_loss: 2.8176
Epoch 36/50
311/311 [==============================] - 2s 5ms/step - loss: 0.6535 - val_loss: 1.8841
Epoch 37/50
311/311 [==============================] - 2s 5ms/step - loss: 0.6495 - val_loss: 2.5448
Epoch 38/50
311/311 [==============================] - 2s 5ms/step - loss: 0.6471 - val_loss: 1.9254
Epoch 39/50
311/311 [==============================] - 1s 5ms/step - loss: 0.6353 - val_loss: 2.3455
Epoch 40/50
311/311 [==============================] - 1s 5ms/step - loss: 0.6295 - val_loss: 2.3603
Epoch 41/50
311/311 [==============================] - 2s 5ms/step - loss: 0.6460 - val_loss: 2.3992
Epoch 42/50
311/311 [==============================] - 1s 5ms/step - loss: 0.6205 - val_loss: 3.2276
Epoch 43/50
311/311 [==============================] - 2s 5ms/step - loss: 0.6177 - val_loss: 2.1013
Epoch 44/50
311/311 [==============================] - 1s 5ms/step - loss: 0.6261 - val_loss: 1.8360
Epoch 45/50
311/311 [==============================] - 1s 5ms/step - loss: 0.6200 - val_loss: 2.5537
Epoch 46/50
311/311 [==============================] - 1s 5ms/step - loss: 0.6121 - val_loss: 2.2328
Epoch 47/50
311/311 [==============================] - 1s 5ms/step - loss: 0.6099 - val_loss: 1.9950
Epoch 48/50
311/311 [==============================] - 2s 5ms/step - loss: 0.5887 - val_loss: 2.1567
Epoch 49/50
311/311 [==============================] - 2s 5ms/step - loss: 0.6057 - val_loss: 1.9668
Epoch 50/50
311/311 [==============================] - 2s 5ms/step - loss: 0.5875 - val_loss: 2.3837
CPU times: user 1min 20s, sys: 7.65 s, total: 1min 28s
Wall time: 1min 17s
```

Figure 33 Run time and loss per epoch

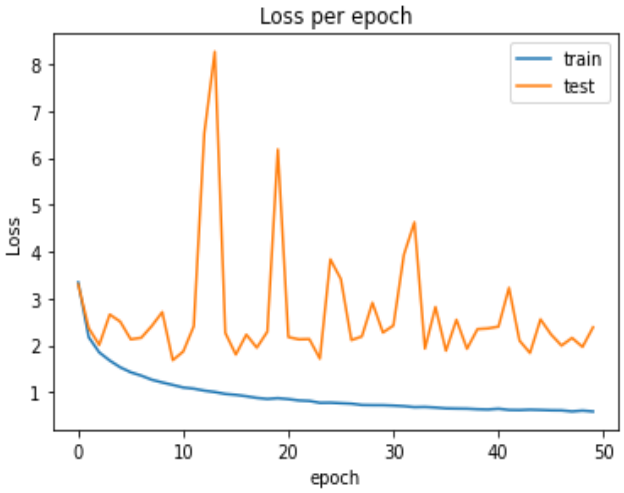

Figure 34 Loss per epoch

	precision	recall	f1-score	support
accuracy			0.57	1383
macro avg	0.63	0.56	0.56	1383
weighted avg	0.66	0.57	0.57	1383

Figure 35 Classification Report

3. Question 3: Lyrics Generation using RNN

Saving lyrics in a text file

In [2]:
```
df = pd.read_csv('LYRICS_DATASET.csv')
df.head()
```

Out[2]:

	Artist Name	Song Name	Lyrics
0	Phoebe Bridgers	Motion Sickness	I hate you for what you did And I miss you li...
1	Phoebe Bridgers	Killer	Sometimes I think I'm a killer I scared you i...
2	Phoebe Bridgers	Georgia	Georgia, Georgia, I love your son And when he...
3	Phoebe Bridgers	Kyoto	Day off in Kyoto Got bored at the temple Look...
4	Phoebe Bridgers	Would You Rather	Playing "would you rather" When it comes to f...

In [3]:
```
df['Lyrics'].to_csv(r'all_lyrics.txt', header=None, index=None, sep='\n', mode='a')
```

Part 1

Reading all_lyrics.txt

In [2]:
```
path_to_file = 'all_lyrics.txt'
```

In [3]:
```
text = open(path_to_file, 'rb').read().decode(encoding='utf-8')

print ('{} characters'.format(len(text)))
```

```
389962 characters
```

In [4]:
```
print(text[:50])
```

```
I hate you for what you did
And I miss you like a
```

Getting a list of unique characters

In [5]:
```
vocab = sorted(set(text))
print ('{} unique characters'.format(len(vocab)))
```

```
97 unique characters
```

Pre-processing

The model doesn't understand characters. So we need to convert it into **numbers**. Here we'll going to build a dictionary for character=>number and number=>character.

In [6]:
```
char2idx = {u:i for i, u in enumerate(vocab)}
idx2char = np.array(vocab)

text_as_int = np.array([char2idx[c] for c in text])
```

In [7]:
```
for char,_ in zip(char2idx, range(20)):
    print('  {:4s}: {:3d},'.format(repr(char), char2idx[char]))
```

```
  '\n':   0,
  ' ' :   1,
  '!' :   2,
  '"' :   3,
  '&' :   4,
  "'" :   5,
  '(' :   6,
  ')' :   7,
  '*' :   8,
  ',' :   9,
  '-' :  10,
  '.' :  11,
  '/' :  12,
  '0' :  13,
  '1' :  14,
  '2' :  15,
  '3' :  16,
  '4' :  17,
  '5' :  18,
  '6' :  19,
```

Let's say we want to map this string to intergers:

I hate you for what

In [8]:
```
print ('{}  {}'.format(repr(text[:20]), text_as_int[:20]))
```

```
'I hate you for what '  [34  1 60 53 72 57  1 77 67 73  1 58 67 70  1 75 60 53 72  1]
```

Creating training examples and targets

Here we're going to break the text into sequence of seq_length + 1 , for example let' say the string is Hello:

1. Then input sequence becomes: "Hell"
2. The output sequence becomes: "ello"

In [9]:
```
seq_length = 100
examples_per_epoch = len(text)//(seq_length+1)

char_dataset = tf.data.Dataset.from_tensor_slices(text_as_int)

for i in char_dataset.take(5):
  print(idx2char[i.numpy()] , end = "")
```

```
I hat
```

In [10]:
```
sequences = char_dataset.batch(seq_length+1, drop_remainder=True)

for item in sequences.take(5):
  print(repr(''.join(idx2char[item.numpy()])))
```

```
"I hate you for what you did\nAnd I miss you like a little kid\nI faked it every time\nBut that's alright"
'\nI can hardly feel anything\nI hardly feel anything at all\nYou gave me fifteen hundred\nTo see your hyp'
'notherapist\nI only went one time\nYou let it slide\nFell on hard times a year ago\nWas hoping you would '
'let it go, and you did\nI have emotional motion sickness\nSomebody roll the windows down\nThere are no w'
"ords in the English language\nI could scream to drown you out\nI'm on the outside looking through\nYou'r"
```

Mapping function

This function will split a string into input and target format:

In [11]:
```
def split_input_target(chunk):
    input_text = chunk[:-1]
    target_text = chunk[1:]
    return input_text, target_text

dataset = sequences.map(split_input_target)
```

Creating training batches

In [12]:
```
# Batch size
BATCH_SIZE = 64
BUFFER_SIZE = 10000
dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE, drop_remainder=True)
dataset
```

Out[12]:
```
<BatchDataset shapes: ((64, 100), (64, 100)), types: (tf.int64, tf.int64)>
```

In [13]:
```
vocab_size = len(vocab)

embedding_dim = 256

rnn_units = 1500
```

Define the model

1. Embedding layer : The input layer. A trainable lookup table that will map the numbers of each character to a vector with embedding_dim dimensions

2. GRU layer : A type of RNN with size units=rnn_units (LSTM could also be used here.)

3. Dense layer : The output layer, with vocab_size outputs and 'RELU' as the activation fuction

4. Dropout layer : Benifits regularisation and prevents overfitting

In [38]:
```
def build_model(vocab_size, embedding_dim, rnn_units, batch_size):
  model = tf.keras.Sequential([

    tf.keras.layers.Embedding(vocab_size, embedding_dim,
                              batch_input_shape=[batch_size, None]),

    tf.keras.layers.GRU(rnn_units,
                        return_sequences=True,
                        stateful=True,
                        recurrent_initializer='glorot_uniform'),

    tf.keras.layers.Dense(vocab_size,activation='relu'),

    tf.keras.layers.Dropout(0.2),
  ])
  return model
```

In [39]:
```
model = build_model(
  vocab_size = len(vocab),
  embedding_dim=embedding_dim,
  rnn_units=rnn_units,
  batch_size=BATCH_SIZE)
```

In [41]:
```
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 embedding_2 (Embedding)      (64, None, 256)           24832

 gru_2 (GRU)                  (64, None, 1500)          7911000

 dense_2 (Dense)              (64, None, 97)            145597

 dropout_2 (Dropout)          (64, None, 97)            0

=================================================================
Total params: 8,081,429
Trainable params: 8,081,429
Non-trainable params: 0
_________________________________________________________________
```

In [42]:
```
def loss(labels, logits):
  return tf.keras.losses.sparse_categorical_crossentropy(labels, logits, from_logits=True)

example_batch_loss  = loss(target_example_batch, example_batch_predictions)
print("Prediction shape: ", example_batch_predictions.shape, " # (batch_size, sequence_length, vocab_size)")
print("scalar_loss:      ", example_batch_loss.numpy().mean())
```

```
Prediction shape:  (64, 100, 97)  # (batch_size, sequence_length, vocab_size)
scalar_loss:       4.5752373
```

In [43]:
```
model.compile(optimizer='adam', loss=loss)
```

Training

```
In [44]: EPOCHS=15

In [45]: history = model.fit(dataset, epochs=EPOCHS)
```

```
Epoch 1/15
60/60 [==============================] - 16s 226ms/step - loss: 3.8047
Epoch 2/15
60/60 [==============================] - 14s 230ms/step - loss: 3.0052
Epoch 3/15
60/60 [==============================] - 14s 226ms/step - loss: 2.8308
Epoch 4/15
60/60 [==============================] - 14s 230ms/step - loss: 2.6977
Epoch 5/15
60/60 [==============================] - 14s 229ms/step - loss: 2.5962
Epoch 6/15
60/60 [==============================] - 14s 226ms/step - loss: 2.5086
Epoch 7/15
60/60 [==============================] - 14s 230ms/step - loss: 2.4329
Epoch 8/15
60/60 [==============================] - 15s 230ms/step - loss: 2.3640
Epoch 9/15
60/60 [==============================] - 14s 231ms/step - loss: 2.3023
Epoch 10/15
60/60 [==============================] - 14s 228ms/step - loss: 2.2419
Epoch 11/15
60/60 [==============================] - 14s 227ms/step - loss: 2.1842
Epoch 12/15
60/60 [==============================] - 14s 230ms/step - loss: 2.1323
Epoch 13/15
60/60 [==============================] - 15s 231ms/step - loss: 2.0592
Epoch 14/15
60/60 [==============================] - 14s 226ms/step - loss: 2.0040
Epoch 15/15
60/60 [==============================] - 14s 229ms/step - loss: 1.9413
```

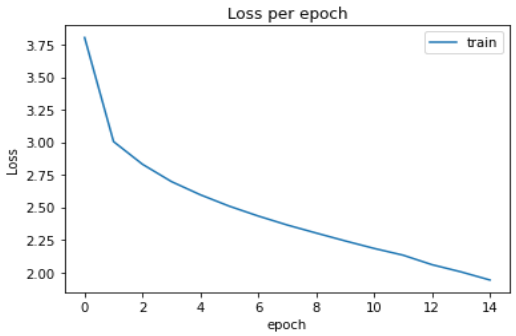

Figure 36 Loss per epoch

Part B: Trying different loss functions

The best loss function for this problem is Sparse Categorical Crossentropy.

Loss Function	**Loss**
Sparse Categorical Crossentropy	1.2670
Mean Squared Error(MSE)	6.6186
Mean Absolute Error(MAE)	5.56
Optimizer	Loss
Adam	1.2670
SGD	3.6936
Nadam	1.8315

Part C: Generating new lyrics

In [ ]:
```
def generate_text(model, chars_to_generate , temp , start_string):
  num_generate = chars_to_generate

  input_eval = [char2idx[s] for s in start_string]
  input_eval = tf.expand_dims(input_eval, 0)

  text_generated = []

  temperature = temp

  model.reset_states()
  for i in range(num_generate):
      predictions = model(input_eval)

      predictions = tf.squeeze(predictions, 0)

      predictions = predictions / temperature
      predicted_id = tf.random.categorical(predictions, num_samples=1)[-1,0].numpy()

      input_eval = tf.expand_dims([predicted_id], 0)

      text_generated.append(idx2char[predicted_id])

  return (start_string + ''.join(text_generated))
```

In [ ]:
```
from numpy import arange

chars_to_generate = 500

print(generate_text(model , chars_to_generate , 0.35 , start_string=u"Baby "))
```

The generated lyric

```
Baby out to the sturt gives to make it because I believe in us
Tell me how long
There's a fire too much
I think I love you better now
I'm in love with your body


Oh honey
I see the stars are the floor
Masier the summer
That you can see the stars that we say


But I will never be the same


But the fell in love with your body
Oh many times, how many times, how much I wanna see you now
I see the sumeching back a shoulders
I'm sunch in the back of my through
The sun line the lovers we need
Like the sta
```

We can note some observations about the generated text.

- It generally conforms to the line format observed in the original text of less than 80 characters before a new line.
- The characters are separated into word-like groups and most groups are actual English words (e.g. “the”, “little” and “was”), but many do not (e.g. “`sumeching`”).
- Some of the words in sequence make sense(e.g. “`how much I wanna see you now` “), but many do not (e.g. “`Like the sta` “).

The fact that this character based model of the lyrics produces output like this is very impressive. This model works well because of LSTMs are a special kind of RNN, **capable of learning long-term dependencies**.

The results are not perfect. We can improve the results with larger LSTM cell.

Part D: Ways to reduce computational cost

1. **Hyper-parameter tuning:** Selecting the right set of hyper-parameters quickly and with le ss number of iterations is critical for reducing computational cost. Here are the top thre e hyper-parameters to focus on:

    - **Mini Batch Size:** A mini-batch size should be large enough to keep the GPU fully utilized but no larger.

    - **Learning Rate:** Selecting the optimum learning rate is in general a useful exercise . However, it becomes even more important for reducing computational cost.

2. **Pre-processing the data:** pre-processing is an important method for reducing computati onal cost. For example in this problem we can remove unnecessary spaces and characte rs.

3. **Smaller Models:** It is obvious that choosing smaller model with less layers and neurons would r educe computational cost.

4. **Know when to stop:** Too many epochs can lead to overfitting of the training dataset, whereas to o few may result in an underfit model. Early stopping is a method that allows you to specify an a rbitrary large number of training epochs and stop training once the model performance stops im proving on a hold out validation dataset.

Part E

LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!

For predicting data in sequence we can use deep learning models like RNN or LSTM. LSTM can be used to predict the next word. The neural network take sequence of words as input and output will be a **matrix of probability** for each word from dictionary to be next of given sequence.