In our last model in file "Music_Generation_Train1.ipynb", we got only 82% accuracy. However, in order to generate melodious music, we need at least 90% accuracy.

So, we have loaded the weights of last epoch from our previous model into this model and also we have added 2 extra layers of LSTM here with more LSTM units.

Here, we are fine-tuning our old layers and we have added more layers. In short, here we are doing "Transfer Learning" from old to new model.

In [1]:

```
import os
import json
import numpy as np
import pandas as pd
from keras.models import Sequential
from keras.layers import LSTM, Dropout, TimeDistributed, Dense, Activation, Embedding
```

```
C:\Users\GauravP\Anaconda3\lib\site-packages\h5py\__init__.py:36: FutureWarning: Conversion of the second argument of i
ssubdtype from `float` to `np.floating` is deprecated. In future, it will be treated as `np.float64 == np.dtype(float).
type`.
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

In [2]:

```
data_directory = "../Data2/"
data_file = "Data_Tunes.txt"
charIndex_json = "char_to_index.json"
model_weights_directory = '../Data2/Model_Weights/'
BATCH_SIZE = 16
SEQ_LENGTH = 64
```

In [3]:
```
def read_batches(all_chars, unique_chars):
    length = all_chars.shape[0]
    batch_chars = int(length / BATCH_SIZE) #155222/16 = 9701

    for start in range(0, batch_chars - SEQ_LENGTH, 64):  #(0, 9637, 64)  #it denotes number of batches. It runs everytim
        #new batch is created. We have a total of 151 batches.
        X = np.zeros((BATCH_SIZE, SEQ_LENGTH))    #(16, 64)
        Y = np.zeros((BATCH_SIZE, SEQ_LENGTH, unique_chars))   #(16, 64, 87)
        for batch_index in range(0, 16):  #it denotes each row in a batch.
            for i in range(0, 64):  #it denotes each column in a batch. Each column represents each character means
                #each time-step character in a sequence.
                X[batch_index, i] = all_chars[batch_index * batch_chars + start + i]
                Y[batch_index, i, all_chars[batch_index * batch_chars + start + i + 1]] = 1 #here we have added '1' becau
                #correct label will be the next character in the sequence. So, the next character will be denoted by
                #all_chars[batch_index * batch_chars + start + i] + 1.
        yield X, Y
```

In [7]:
```
def built_model(batch_size, seq_length, unique_chars):
    model = Sequential()

    model.add(Embedding(input_dim = unique_chars, output_dim = 512, batch_input_shape = (batch_size, seq_length), name =

    model.add(LSTM(256, return_sequences = True, stateful = True, name = "lstm_first"))
    model.add(Dropout(0.2, name = "drp_1"))

    model.add(LSTM(256, return_sequences = True, stateful = True))
    model.add(Dropout(0.2))

    model.add(LSTM(256, return_sequences = True, stateful = True))
    model.add(Dropout(0.2))

    model.add(TimeDistributed(Dense(unique_chars)))
    model.add(Activation("softmax"))

    model.load_weights("../Data/Model_Weights/Weights_80.h5", by_name = True)

    return model
```

In [8]:
```
def training_model(data, epochs = 90):
    #mapping character to index
    char_to_index = {ch: i for (i, ch) in enumerate(sorted(list(set(data))))}
    print("Number of unique characters in our whole tunes database = {}".format(len(char_to_index))) #87

    with open(os.path.join(data_directory, charIndex_json), mode = "w") as f:
        json.dump(char_to_index, f)

    index_to_char = {i: ch for (ch, i) in char_to_index.items()}
    unique_chars = len(char_to_index)

    model = built_model(BATCH_SIZE, SEQ_LENGTH, unique_chars)
    model.summary()
    model.compile(loss = "categorical_crossentropy", optimizer = "adam", metrics = ["accuracy"])

    all_characters = np.asarray([char_to_index[c] for c in data], dtype = np.int32)
    print("Total number of characters = "+str(all_characters.shape[0])) #155222

    epoch_number, loss, accuracy = [], [], []

    for epoch in range(epochs):
        print("Epoch {}/{}".format(epoch+1, epochs))
        final_epoch_loss, final_epoch_accuracy = 0, 0
        epoch_number.append(epoch+1)

        for i, (x, y) in enumerate(read_batches(all_characters, unique_chars)):
            final_epoch_loss, final_epoch_accuracy = model.train_on_batch(x, y) #check documentation of train_on_batch he
            print("Batch: {}, Loss: {}, Accuracy: {}".format(i+1, final_epoch_loss, final_epoch_accuracy))
            #here, above we are reading the batches one-by-one and train our model on each batch one-by-one.
        loss.append(final_epoch_loss)
        accuracy.append(final_epoch_accuracy)

        #saving weights after every 10 epochs
        if (epoch + 1) % 10 == 0:
            if not os.path.exists(model_weights_directory):
                os.makedirs(model_weights_directory)
            model.save_weights(os.path.join(model_weights_directory, "Weights_{}.h5".format(epoch+1)))
            print('Saved Weights at epoch {} to file Weights_{}.h5'.format(epoch+1, epoch+1))

    #creating dataframe and record all the losses and accuracies at each epoch
    log_frame = pd.DataFrame(columns = ["Epoch", "Loss", "Accuracy"])
```

```
    log_frame["Epoch"] = epoch_number
    log_frame["Loss"] = loss
    log_frame["Accuracy"] = accuracy
    log_frame.to_csv("../Data2/log.csv", index = False)
```

In [14]:

```
file = open(os.path.join(data_directory, data_file), mode = 'r')
data = file.read()
file.close()
if __name__ == "__main__":
    training_model(data)
```

In [10]:
```
log = pd.read_csv(os.path.join(data_directory, "log.csv"))
log
```

Out[10]:

	Epoch	Loss	Accuracy
0	1	2.570057	0.293945
1	2	1.850464	0.488281
2	3	1.504380	0.562500
3	4	1.353270	0.593750
4	5	1.247315	0.617188
5	6	1.169375	0.625000
6	7	1.088979	0.658203
7	8	1.049507	0.677734
8	9	0.988916	0.684570
9	10	0.946073	0.682617
10	11	0.922529	0.703125
11	12	0.904022	0.722656
12	13	0.890658	0.714844
13	14	0.844449	0.726562
14	15	0.815944	0.741211
15	16	0.829617	0.728516
16	17	0.775039	0.749023
17	18	0.766915	0.750000
18	19	0.767771	0.750000
19	20	0.733762	0.756836
20	21	0.705088	0.777344
21	22	0.708641	0.775391
22	23	0.675398	0.776367

	Epoch	Loss	Accuracy
23	24	0.719725	0.765625
24	25	0.662180	0.779297
25	26	0.635798	0.789062
26	27	0.614068	0.794922
27	28	0.621199	0.795898
28	29	0.608465	0.804688
29	30	0.592249	0.793945
...	...	...	...
60	61	0.385737	0.867188
61	62	0.356167	0.883789
62	63	0.371307	0.878906
63	64	0.357482	0.884766
64	65	0.340871	0.877930
65	66	0.372424	0.870117
66	67	0.359206	0.881836
67	68	0.323794	0.891602
68	69	0.349235	0.883789
69	70	0.341775	0.885742
70	71	0.328699	0.879883
71	72	0.302101	0.902344
72	73	0.348871	0.882812
73	74	0.324025	0.900391
74	75	0.297615	0.895508
75	76	0.331783	0.886719
76	77	0.315978	0.900391
77	78	0.301240	0.900391

	Epoch	Loss	Accuracy
78	79	0.314003	0.913086
79	80	0.330850	0.891602
80	81	0.307386	0.898438
81	82	0.312513	0.896484
82	83	0.303039	0.888672
83	84	0.293686	0.911133
84	85	0.325840	0.899414
85	86	0.295506	0.904297
86	87	0.295797	0.903320
87	88	0.292568	0.893555
88	89	0.276495	0.912109
89	90	0.268679	0.916016

90 rows × 3 columns