Music Generation Using Deep Learning

Real World Problem

This case-study focuses on generating music automatically using Recurrent Neural Network(RNN).
We do not necessarily have to be a music expert in order to generate music. Even a non expert can generate a decent quality music using RNN.
We all like to listen interesting music and if there is some way to generate music automatically, particularly decent quality music then it's a big leap in the world of music industry.

Task: Our task here is to take some existing music data then train a model using this existing data. The model has to learn the patterns in music that we humans enjoy. Once it learns this, the model should be able to generate new music for us. It cannot simply copy-paste from the training data. It has to understand the patterns of music to generate new music. We here are not expecting our model to generate new music which is of professional quality, but we want it to generate a decent quality music which should be melodious and good to hear.

Now, what is music? In short music is nothing but a sequence of musical notes. Our input to the model is a sequence of musical events/notes. Our output will be new sequence of musical events/notes. In this case-study we have limited our self to single instrument music as this is our first cut model. In future, we will extend this to multiple instrument music.

Data Source:

1. [http://abc.sourceforge.net/NMD/ (http://abc.sourceforge.net/NMD/)](http://abc.sourceforge.net/NMD/)
2. [http://trillian.mit.edu/~jc/music/book/oneills/1850/X/ (http://trillian.mit.edu/~jc/music/book/oneills/1850/X/)](http://trillian.mit.edu/~jc/music/book/oneills/1850/X/)

From first data-source, we have downloaded first two files:

- Jigs (340 tunes)
- Hornpipes (65 tunes)

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 [3]:
```
data_directory = "../Data/"
data_file = "Data_Tunes.txt"
charIndex_json = "char_to_index.json"
model_weights_directory = '../Data/Model_Weights/'
BATCH_SIZE = 16
SEQ_LENGTH = 64
```

In [4]:
```
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 [8]:
```
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)))

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

    model.add(LSTM(128, return_sequences = True, stateful = True))
    model.add(Dropout(0.2))

    model.add(TimeDistributed(Dense(unique_chars)))

    model.add(Activation("softmax"))

    return model
```

In [13]:
```
def training_model(data, epochs = 80):
    #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("../Data/log.csv", index = False)
```

In [5]:
```
file = open(os.path.join(data_directory, data_file), mode = 'r')
data = file.read()
file.close()
if __name__ == "__main__":
    training_model(data)
```

In [4]:
```
log = pd.read_csv(os.path.join(data_directory, "log.csv"))
log
```

Out[4]:

	Epoch	Loss	Accuracy
0	1	2.643317	0.290039
1	2	1.873376	0.496094
2	3	1.548782	0.557617
3	4	1.417467	0.597656
4	5	1.348234	0.585938
5	6	1.265394	0.618164
6	7	1.186394	0.630859
7	8	1.145774	0.642578
8	9	1.097427	0.656250
9	10	1.073594	0.650391
10	11	1.052364	0.674805
11	12	1.011208	0.666016
12	13	1.004766	0.672852
13	14	0.980474	0.685547
14	15	0.972413	0.681641
15	16	0.935533	0.703125
16	17	0.930730	0.695312
17	18	0.896762	0.713867
18	19	0.896243	0.708008
19	20	0.886423	0.713867
20	21	0.853774	0.722656
21	22	0.858230	0.716797
22	23	0.861040	0.707031

	Epoch	Loss	Accuracy
23	24	0.835705	0.721680
24	25	0.818098	0.735352
25	26	0.807396	0.725586
26	27	0.800719	0.735352
27	28	0.797581	0.740234
28	29	0.786037	0.737305
29	30	0.769117	0.744141
...	...	...	...
50	51	0.649517	0.786133
51	52	0.646988	0.793945
52	53	0.635747	0.794922
53	54	0.626719	0.812500
54	55	0.643305	0.793945
55	56	0.628394	0.803711
56	57	0.639661	0.797852
57	58	0.620944	0.800781
58	59	0.593881	0.812500
59	60	0.609697	0.798828
60	61	0.592073	0.807617
61	62	0.591695	0.807617
62	63	0.593241	0.805664
63	64	0.592210	0.815430
64	65	0.600351	0.801758
65	66	0.546197	0.823242
66	67	0.582400	0.817383
67	68	0.582353	0.817383

	Epoch	Loss	Accuracy
68	69	0.560872	0.809570
69	70	0.562345	0.805664
70	71	0.562496	0.828125
71	72	0.558382	0.818359
72	73	0.558365	0.821289
73	74	0.576193	0.820312
74	75	0.592619	0.817383
75	76	0.537521	0.838867
76	77	0.558197	0.821289
77	78	0.541944	0.835938
78	79	0.534475	0.825195
79	80	0.515541	0.828125

80 rows × 3 columns