Generating Bollywood Music Using Deep Learning

Introduction

Music generation is one of the classic applications of deep learning and has been a topic of much application and research in Deep Learning, especially recently. This paper intends to create a deep learning music generation model specialized for Bollywood music that is trained exclusively on Bollywood music.

Figure 1 Source : https://www.analyticsvidhya.com/blog/2020/01/how-to-perform-automatic-music-generation/

Source Of Training Data

In order to extract the features from Bollywood music, traditional mp3/wav/other typical audio file formats cannot be used, as they store the physical data of the music itself, whereas we require the actual notes and associated data for a given music piece. The MIDI (Musical Instrument Digital Interface) .mid file format, which itself is used to store recordings and production data of music, stores this data. For a given piece of music, MIDI data contains the 'identity' features of a given music piece that make it what it is, like what notes are played, for how long, their offsets and durations, the chords, etc. independent of any instrument or performance, whereas other file formats store the physical features of a given performance of that piece of music (amplitude/time, the associated frequencies, etc.) Thus, we use .mid files of Bollywood music pieces as our training data.

Usually, .mid file formats are harder to find and reserved for production uses; however, I was able to find a resource online, Jsplash, that contains the .mid file formats for a total of 71 Bollywood songs, with their lyrics removed. [1] This should provide ample data for the RNN.

Pre-processing Audio Data

In order to get the audio data in a temporal-axis format compatible to use as the input for the RNN, we use MIT's python music21 library to extract the notes of all the instruments for each song in our training set, along with their respective durations and offsets.

The majority of the process of first extracting the features from the input .mid files and then converting those features into a fixed matrix format for the RNN have been taken from this blog post [2], which deals with music generation of only classical piano music.

Much of the initial struggles in this project arose from processing this initial MIDI data. This is because they use undocumented instruments with their own uniquely encoded notes/keys for different sounds. For example, on partitioning by instrument and then iterating through all the instruments for a given .mid file:

```
parsing kattadi.mid...
Instruments in kattadi.mid:
['Flute', '0x0000020A78C57C08', 'Marimba', 'Recorder']
```

'0x0000020A78C57C08' is an instrument/sound undocumented by music21, and tended to provide broken data for notes, chords, durations and offsets. Many times this data was mixed with the data of instruments, requiring me to manually look through them to identify and clean discrepancies by finding those regions of the song where the undocumented instrument is being played and simply splicing it out.

Eventually, however, I was able to find and use a function in the music21 library to automatically partition by instrument (as done above) and remove all instances of non-instrument and unrecognized-instrument data automatically. This breakdown into one instrument/part for MIDI was done using a method similar to the one found in this research paper [4].

The other main difference/addition from the converter used in the blog post [2] and my data is that the Bollywood songs, as shown above, contain multiple instruments being played simultaneously on independent channels, whereas the converter from the blog post simply parses single-channel, single-instrument classical music. I want to extract the features of all the instruments used in the piece. Thus, I had two hypotheses on how to adapt this converter for my case:

1. Extract only the features of the most ‘dominant’ instrument (defined as the instrument of a given music piece/song with the most notes) from that song and use the notes, offsets & durations of that instrument to represent the feature vector for that particular training example. In this model, one Bollywood song would produce one training example, so m = 71. (Implemented as the function `parse_music_dominant_instrument()`)
2. Extract the features of all the instruments from that song and use the notes, offsets & durations of all instruments as separate vectors, each effectively being their own training example. This would allow each song to produce as many training examples as the number of ‘significant’ instruments used in that song (significant being >=5 notes played with that instrument), increasing the m to be >71 (m = ~240 as on average 3-4 significant instruments are in each file). (Implemented as the function `parse_music_all_instruments()`)

I initially felt that using the dominant instrument was the better option, as using all instruments would create a lot of variation in the training data: some instruments (like bass, for example) vary significantly in how frequently their notes are played from others that have many notes played quickly (like the sitar). The dominant instrument should result in overall better samples, even if it overall created less training data.

However, on experimenting with both methods, most instruments in the training data do have more than enough notes to constitute a good training example, and even those with few notes are valuable outlier training examples for the model. Using the extra training data does not skew the produced samples or cause any difference to the loss change per epoch; thus, for the final model, I used all instruments with a minimum significance requirement of 5 notes (tuned hyperparameter).

The RNN Model

The original version of the RNN model that I started with was also adapted from the same blog post [2]. After significant iteration, it was modified across multiple versions to become the final version, but still resembles the one from the blog post.

The Final RNN model is a Recurrent Neural Network with 3 major layers using LSTM units with many neurons in the first major layer (4x input size) consisting of LSTM (with activations, Batch Normalization and Dropout) -> Attention -> Dense w/ Reshaping, essentially a copy of the first layer in the second layer (4x input size), followed by a Dense Layer Resizer (to 1x input size) for the final LSTM. It's finally classified with another Dense Layer with Softmax activation to classify it into one of the 128 note values.

The model implements attention in the first two layers as well as the LSTM units in order to maximise retention and memory to remember sequence data from even one end of the temporal axis when it's predicting notes at the other end of the temporal axis.

The model is visualized by plotting it in keras (see appendix [vi]).

The full details of the model summary produced by keras is also provided (see appendix [v]).

The model also ultimately does not use bidirectionality in any of the layers, as bidirectionality proves to be mostly redundant for the sake of generating sequences, as only data from the end of the sequence already predicted/is given (the past) is known to the RNN anyway during prediction sampling. This reason for not using bidirectionality is further explained in more detail below.

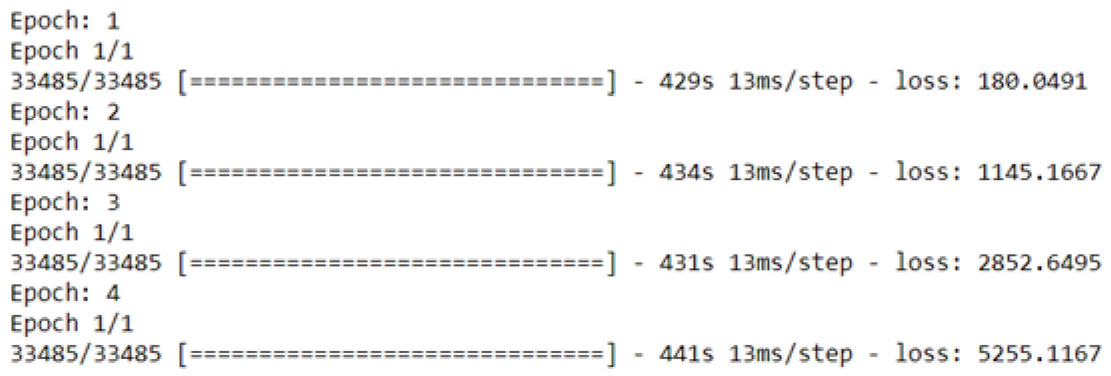
Sampling

After the model is trained, we sample it by feeding a max_len (number of past notes' activations used to predict the next note) number of sequential unit data points (i.e, notes) randomly taken from any song from the training matrix, and then let it predict the next note, and feed that note as the input for the next note and on and on.

Iterative Process

The original network v0 is a deep, 3-layer RNN with LSTM units [i] and light attention. Upon training it with the test data, the immediate problems with the first batch of sampled music were obvious:

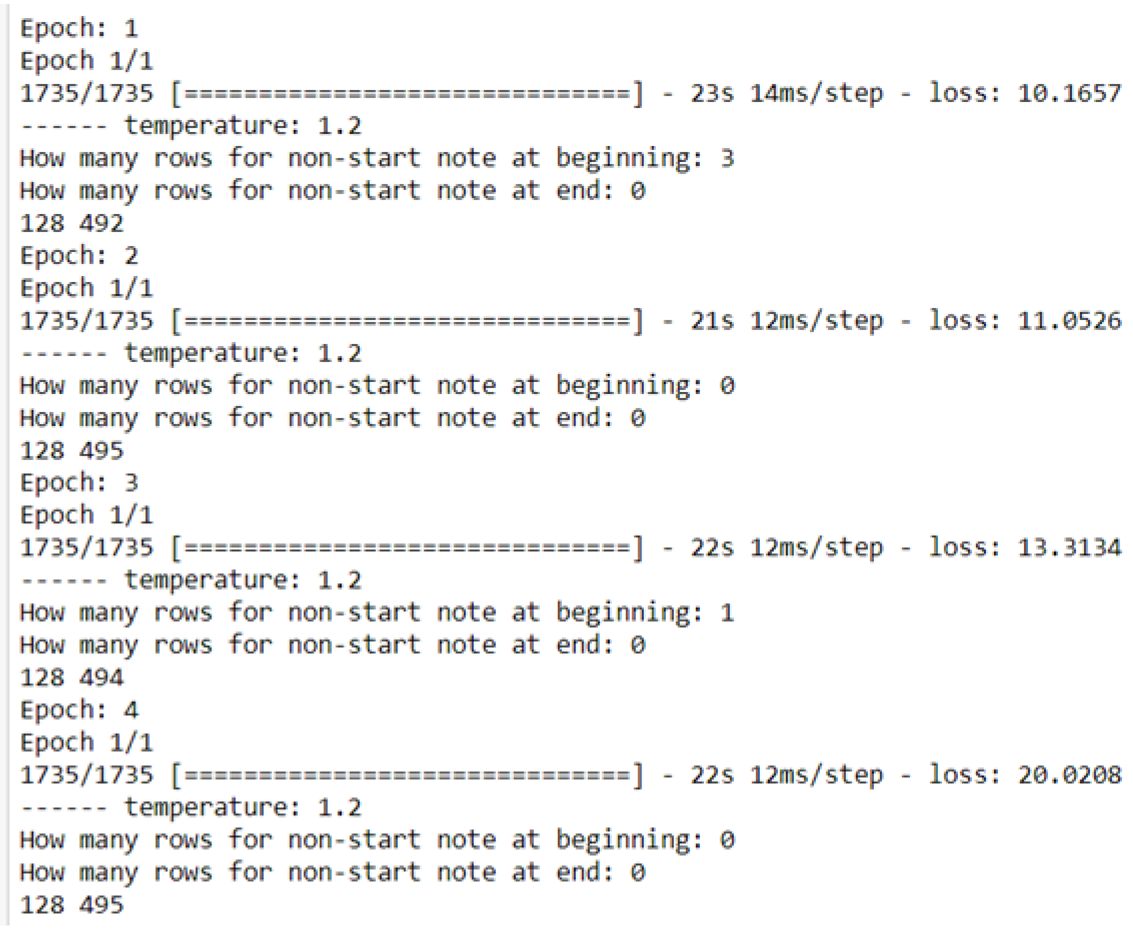
I. The loss does not decrease at all and starts constantly increasing right from the first epoch exponentially.



```
Epoch: 1
Epoch 1/1
33485/33485 [==============================] - 429s 13ms/step - loss: 180.0491
Epoch: 2
Epoch 1/1
33485/33485 [==============================] - 434s 13ms/step - loss: 1145.1667
Epoch: 3
Epoch 1/1
33485/33485 [==============================] - 431s 13ms/step - loss: 2852.6495
Epoch: 4
Epoch 1/1
33485/33485 [==============================] - 441s 13ms/step - loss: 5255.1167
```

II. For the output sampled audio, the first 4 seconds are the original, thematic, high quality samples taken randomly from the training data that the model then uses to predict future notes. However, the predicted notes immediately after tend to be of repeating very similar/same patterns that are different from the preceding original data. Essentially, there is a clear discrepancy between the audio from the data and the notes predicted by the model (the model's predicted notes are still musical/harmonious; it simply doesn't fit the 'theme'/pattern of the x values it's using to predict the next notes from.) (Extreme audio sample example of this problem [1].) After running diagnoses and viewing the gradient values across different parts of the RNN, the likely causes were addressed as follows:

The first thing I tested was tuning alpha; if alpha is too high, the gradients become too large and preventing the optimizer from reaching a minimum. I iteratively reduced alpha down up to 0.0001 and tested the same test case with extra epochs. It clearly had been too high, and a smaller alpha helped in reducing the initial loss. However, the same exploding loss was seen, just slower for smaller alphas.



```
Epoch: 1
Epoch 1/1
1735/1735 [==============================] - 23s 14ms/step - loss: 10.1657
------ temperature: 1.2
How many rows for non-start note at beginning: 3
How many rows for non-start note at end: 0
128 492
Epoch: 2
Epoch 1/1
1735/1735 [==============================] - 21s 12ms/step - loss: 11.0526
------ temperature: 1.2
How many rows for non-start note at beginning: 0
How many rows for non-start note at end: 0
128 495
Epoch: 3
Epoch 1/1
1735/1735 [==============================] - 22s 12ms/step - loss: 13.3134
------ temperature: 1.2
How many rows for non-start note at beginning: 1
How many rows for non-start note at end: 0
128 494
Epoch: 4
Epoch 1/1
1735/1735 [==============================] - 22s 12ms/step - loss: 20.0208
------ temperature: 1.2
How many rows for non-start note at beginning: 0
How many rows for non-start note at end: 0
128 495
```

Figure 2 (Still showing exploding loss; just slower for smaller alphas. Alpha = 0.0001)

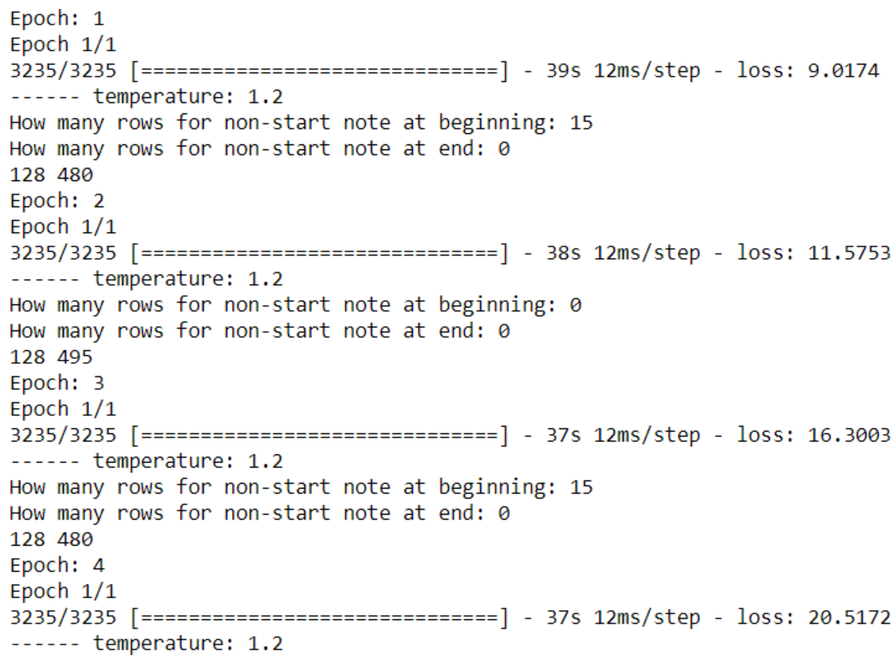
The sampled audio showing repetitive patterns then points to vanishing gradients – with the network being so deep, many of the weights end up being exponential reductions due to the higher power values of <1 decimal weights. This would explain the repetitive pattern: it defaults to a single few notes with so many redundant/useless weights vanishing to 0. I implemented attention with more layers and neurons into the network to address this.

With heavier and more attention weights being implemented into the first two layers of the network, although this helped with the problem of vanishing gradients, there is still something in the network causing the weights to tend to 0 and vanish. I then hypothesised that the additional or other major cause could be due to overfitting.

[1] Extreme example of 4s problem:
https://drive.google.com/file/d/1y0UKOVN98cXqWXSuX7PdpoybDESKCAIq/view?usp=sharing

Overfitting is the simplest explanation. The plotted learning curves also pointed to high variance. The network is too complex/deep for this kind of data and/or there is an insufficient amount of data, and many of the weights are redundant, forcing too many of the gradients to tend towards 0. This would cause the exponentially rising loss.

To address overfitting, I reduced the complexity of the network (halved the number of neurons in all LSTM layers and Dense layers except the final layer). (New v1 model summary with the new large attention layers and halved normal layers [ii]). This reduced the number of trained weights from 8 million to 6 million. This reduced the rate of the escalating loss, but it still refused to converge to any minimum and kept tending to increase after each iteration/epoch.



```
Epoch: 1
Epoch 1/1
3235/3235 [==============================] - 39s 12ms/step - loss: 9.0174
------ temperature: 1.2
How many rows for non-start note at beginning: 15
How many rows for non-start note at end: 0
128 480
Epoch: 2
Epoch 1/1
3235/3235 [==============================] - 38s 12ms/step - loss: 11.5753
------ temperature: 1.2
How many rows for non-start note at beginning: 0
How many rows for non-start note at end: 0
128 495
Epoch: 3
Epoch 1/1
3235/3235 [==============================] - 37s 12ms/step - loss: 16.3003
------ temperature: 1.2
How many rows for non-start note at beginning: 15
How many rows for non-start note at end: 0
128 480
Epoch: 4
Epoch 1/1
3235/3235 [==============================] - 37s 12ms/step - loss: 20.5172
------ temperature: 1.2
```

Figure 3 (With Alpha = 0.0002; loss rising more slowly relative to previous iterations)

Still, compared to the earlier exponential rise of loss, this was an improvement.

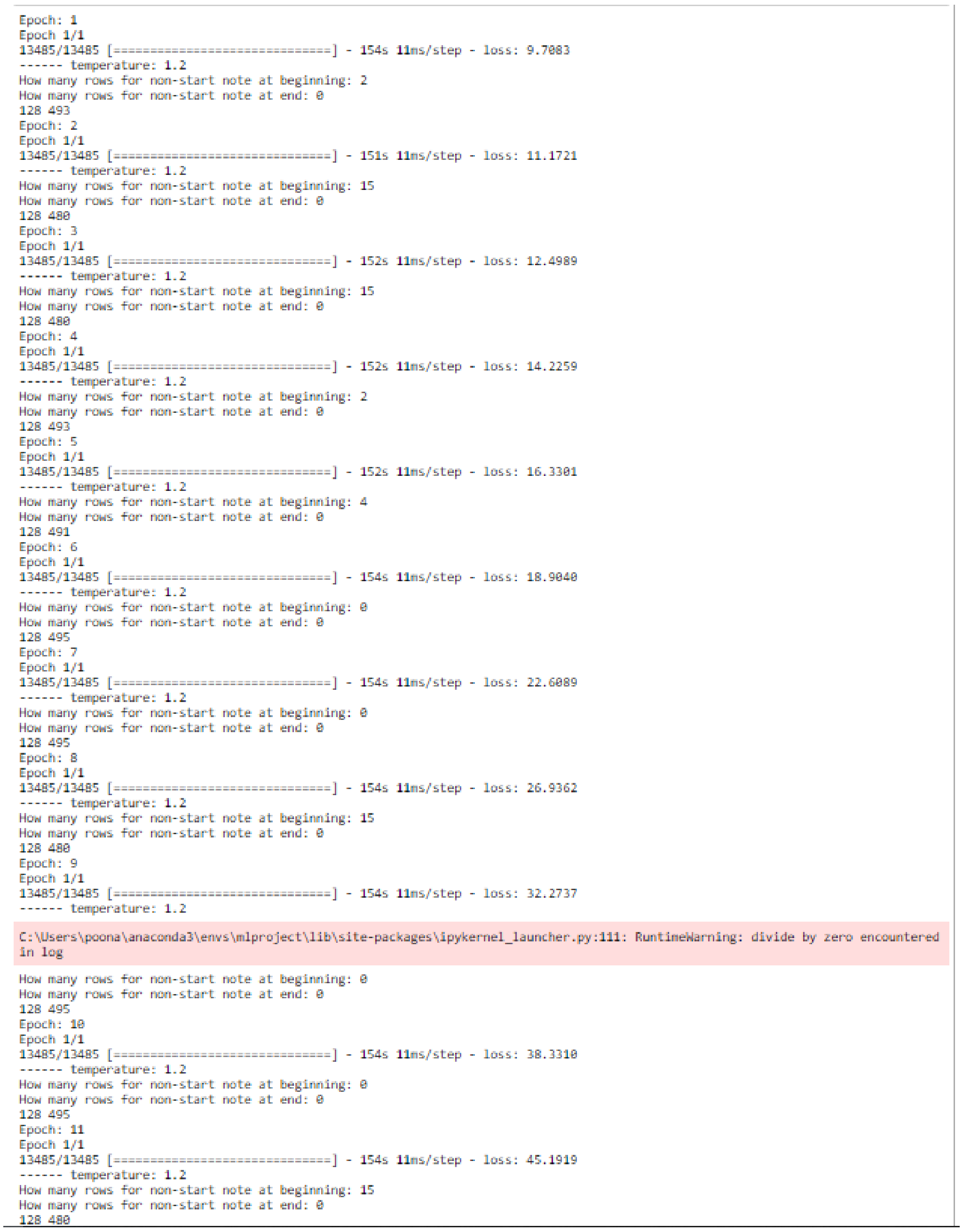
So next, I increased the size of the data used in this iterative process, as more data would help address overfitting. This again helped the problem, but only slightly reduced the exploding costs.

I also realized at this point that the '4-second' problem might have also been due to the fact that I could only train for 4 epochs at a time due to time constraints, as each epoch takes 5-7 minutes on the full training set. It just hasn't trained enough to predict the most appropriate notes for a given 4 second sample, even if it able to produce harmonious notes independently. Sure enough, the problem was significantly addressed simply by training it for longer (audio sample example from longer trained model [2]). However, the loss was still rising, but very slowly.

Additionally, all these optimisations (reduced alpha = 0.0001/0.0002, a larger data set for training, attention with many activations implemented into the second network, halved number of neurons) the problem of only the first 4 seconds sounding different was gone. The model could now produce music that is consistent.

[2] Harmonious Transition From given first 4s into predicted notes Example: https://drive.google.com/file/d/1O8r0V3GVzwyumCjRWv5ANF3i04zJu-BH/view?usp=sharing

Clearly, the problem of vanishing gradients by now had been significantly mitigated, but since the loss was still rising with each iteration, it was still present, and there was clearly still a problem with overfitting causing growing loss.



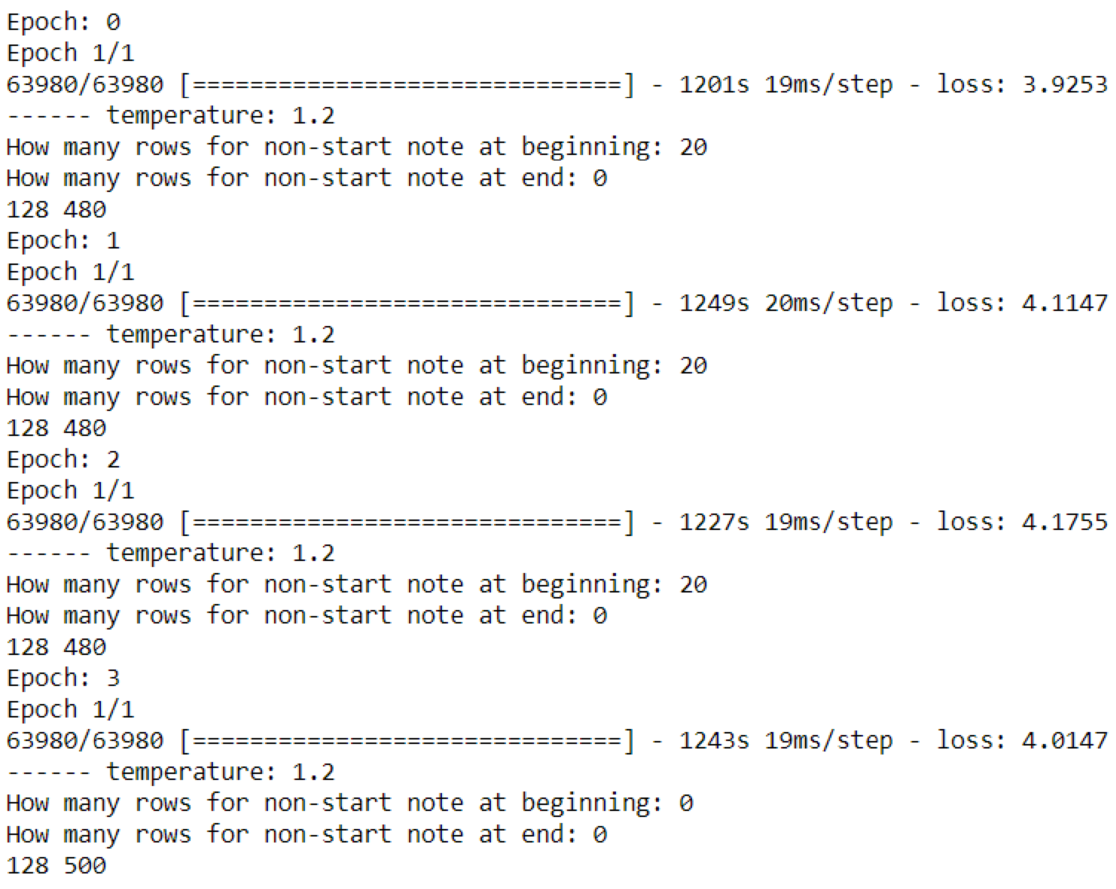
```
Epoch: 1
Epoch 1/1
13485/13485 [==============================] - 154s 11ms/step - loss: 9.7083
------ temperature: 1.2
How many rows for non-start note at beginning: 2
How many rows for non-start note at end: 0
128 493
Epoch: 2
Epoch 1/1
13485/13485 [==============================] - 151s 11ms/step - loss: 11.1721
------ temperature: 1.2
How many rows for non-start note at beginning: 15
How many rows for non-start note at end: 0
128 480
Epoch: 3
Epoch 1/1
13485/13485 [==============================] - 152s 11ms/step - loss: 12.4989
------ temperature: 1.2
How many rows for non-start note at beginning: 15
How many rows for non-start note at end: 0
128 480
Epoch: 4
Epoch 1/1
13485/13485 [==============================] - 152s 11ms/step - loss: 14.2259
------ temperature: 1.2
How many rows for non-start note at beginning: 2
How many rows for non-start note at end: 0
128 493
Epoch: 5
Epoch 1/1
13485/13485 [==============================] - 152s 11ms/step - loss: 16.3301
------ temperature: 1.2
How many rows for non-start note at beginning: 4
How many rows for non-start note at end: 0
128 491
Epoch: 6
Epoch 1/1
13485/13485 [==============================] - 154s 11ms/step - loss: 18.9040
------ temperature: 1.2
How many rows for non-start note at beginning: 0
How many rows for non-start note at end: 0
128 495
Epoch: 7
Epoch 1/1
13485/13485 [==============================] - 154s 11ms/step - loss: 22.6089
------ temperature: 1.2
How many rows for non-start note at beginning: 0
How many rows for non-start note at end: 0
128 495
Epoch: 8
Epoch 1/1
13485/13485 [==============================] - 154s 11ms/step - loss: 26.9362
------ temperature: 1.2
How many rows for non-start note at beginning: 15
How many rows for non-start note at end: 0
128 480
Epoch: 9
Epoch 1/1
13485/13485 [==============================] - 154s 11ms/step - loss: 32.2737
------ temperature: 1.2

C:\Users\poona\anaconda3\envs\mlproject\lib\site-packages\ipykernel_launcher.py:111: RuntimeWarning: divide by zero encountered
in log

How many rows for non-start note at beginning: 0
How many rows for non-start note at end: 0
128 495
Epoch: 10
Epoch 1/1
13485/13485 [==============================] - 154s 11ms/step - loss: 38.3310
------ temperature: 1.2
How many rows for non-start note at beginning: 0
How many rows for non-start note at end: 0
128 495
Epoch: 11
Epoch 1/1
13485/13485 [==============================] - 154s 11ms/step - loss: 45.1919
------ temperature: 1.2
How many rows for non-start note at beginning: 15
How many rows for non-start note at end: 0
128 480
```

Figure 4 Rising loss, but slowly across many iterations. Alpha = 0.0002

The 4 second problem did still occasionally seem like it was present in a few samples, but it could be explained as it simply learning that format from the nature of the data; almost all of the Bollywood songs used, there is a short unique introduction with relatively unique notes for a few seconds, followed by a very consistent beat pattern (similar notes, durations and offsets) for the rest of the song. The model could simply have learned this feature from the data itself and assume that the earliest notes of the song are supposed to be unique.

To significantly address overfitting, I then tried a new much simpler version v2 of the network [iii], where I again halved the number of all the neurons in the first layer of the network. This reduced the number of parameters from 6 million to 2.5 million.

At this point, although the loss was either growing or remaining on each iteration, the vanishing gradients problem had been sufficiently mitigated, and the output sampled audio was decent enough that the algorithm was clearly learning many complex features, even if it wasn't being perfectly optimised in terms of loss. (Music Sample from this model [3]).



```
Epoch: 0
Epoch 1/1
63980/63980 [==============================] - 1201s 19ms/step - loss: 3.9253
------ temperature: 1.2
How many rows for non-start note at beginning: 20
How many rows for non-start note at end: 0
128 480
Epoch: 1
Epoch 1/1
63980/63980 [==============================] - 1249s 20ms/step - loss: 4.1147
------ temperature: 1.2
How many rows for non-start note at beginning: 20
How many rows for non-start note at end: 0
128 480
Epoch: 2
Epoch 1/1
63980/63980 [==============================] - 1227s 19ms/step - loss: 4.1755
------ temperature: 1.2
How many rows for non-start note at beginning: 20
How many rows for non-start note at end: 0
128 480
Epoch: 3
Epoch 1/1
63980/63980 [==============================] - 1243s 19ms/step - loss: 4.0147
------ temperature: 1.2
How many rows for non-start note at beginning: 0
How many rows for non-start note at end: 0
128 500
```

Figure 5 Loss remaining static or slowly growing; Alpha = 0.0002

Thus, I decided to implement bidirectionality into both layers of the LTSM at this point (Summary [iv]). Combined with significantly large attention parameters and the LSTM units themselves, this would give the model a large amount of memory across the temporal axis of the data to be able to recognize very high-level patterns and features like choruses. Adding bidirectionality effectively doubled the number of outputs from the 2 Bidirectional LSTM layers, so the attention layer immediately following them were also doubled in the number of activations accordingly.

Surprisingly, adding bidirectionality further addressed the problem of vanishing gradients; the loss was now much more static and slower to rise. The samples produced by the audio were noticeably more complex in certain parts. However, I also noticed that it mostly tended to repeat similar patterns throughout the duration of their sampled audio.

I then trained this model on the complete dataset for a total of 28 epochs. After implementing GPU support, this took 10 hours using all the training data. This ultimately actually caused the loss to stay static and reduce between epochs for the first time, clearly indicating that it was overfitting earlier and was now a good fit for the amount and complexity of the given data. See Appendix [vii] for the loss from each epoch.

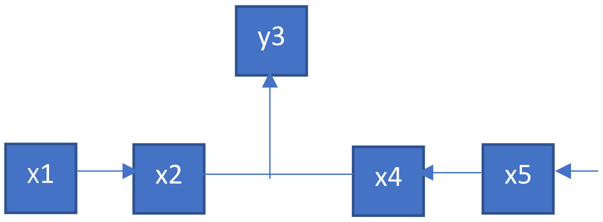
[3] Music Sample Example with simplified network v2: https://drive.google.com/file/d/1ZTVyusk_tSpQtuzLHJeihqSCWH2LwZnv/view?usp=sharing

However, there was a significant problem I realized when sampling the data from this trained model (sample from this model [4]).:

The Bidirectionality Problem During Sampling/Prediction

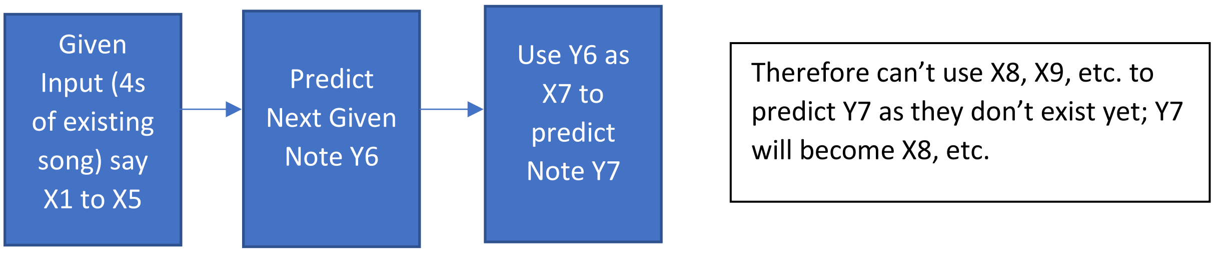
My initial hypothesis was that if the model understands all the temporal data across a song before making a prediction for the next note, it would be better able to estimate the next note. Visualisation of how bidirectionality is implemented during the training of the model:

If max_len = 2 (i.e, every 2 notes + their activations with LSTM memory for all the notes before and after these notes)



This works effectively for the training process; however, during the prediction process, all our future data is yet to be predicted; thus, half the weights are effectively redundant.

Sampling/Prediction Process:


Given Input (4s of existing song) say X1 to X5
Predict Next Given Note Y6
Use Y6 as X7 to predict Note Y7
Therefore can't use X8, X9, etc. to predict Y7 as they don't exist yet; Y7 will become X8, etc.

Thus, bidirectionality can cause a large amount of redundancy during the sampling process.

Regardless, I had still hoped that during training, it would serve as an effective way to make the activations of the weights before the to-be-predicted note understand the implications of the future weights (i.e,they will learn that there are complex patterns in future notes, even if they don't yet know what they are).

However, on iterating otherwise similar models, one with bidirectionality and one without, it seems to be ultimately be redundant, and only unnecessarily overcomplicates the model, further worsening the problem of overfitting. It also confuses the activations of the left side that do the actual predicting, creating very dull, simple and lifeless note patterns, as can be heard from the sample provided earlier. Thus, for the final version v4 model, I have removed the bidirectionality in the first two layers. The new v4 model is essentially a slightly modified version of v2 [v].

With bidirectionality removed, some of the model layers are now halved in their width, but we want to retain the same number of neurons as during bidirectionality as that model was a good fit for the data and removing it might cause underfitting. Thus, the number of neurons in all parts of the second layer are doubled to produce v4 of the model. [v]

[4]Bidirectional model sample: https://drive.google.com/file/d/19g8j21RiYk-iJg7cjlho6kE6Z-Ekji0C/view?usp=sharing

I then ran this final version of the model by training it with all the training data for 4 epochs. (Example sample [5]).

Postprocessing & Sampling

The output array is again reconverted into a MIDI format using the same methods from the referenced blog post [2].

The model is trained to only predict the next note but does not possess any encoding as to what instrument should be which note. Thus, the predicted notes are simply encoded with the music21 library using the basic music21.instrument.Piano() class to encode them all with Piano notes.

These MIDI files are then postprocessed by hand to create frequency groups (say, all notes between C1 and C3 will be of a particular instrument) and assigned different instruments using these partitions. This was done using Ableton (a Digital Audio Workstation software). By doing so, we can reapply Indian instruments to produce Bollywood (and other genres!) of music.

Unfortunately, music21 does not support modifying the instrument on a per-note basis; each stream can only consist of notes from one channel and one instrument [5]. Thus, all the output files must be of a single instrument and must be modified using Digital Audio Production software by hand (like Ableton).

Thus, I automated the postprocessing to randomly choose an Indian instrument (the Sitar, Shehnai, or Flute) for the automatically produced samples of the model.

Interesting Observations & Results

The biggest common feature across all the iterations was that the samples seems to always have one simple beat using only a handful of keys, mixed in with other more complex choruses. This beat also remains very similar across different samples, which is likely a limitation for not being trained for very long (only 4 epochs for the final model), but mainly seems to be a function of the training data, as Bollywood songs with vocals all employ one 'base' beat that allows the vocals to stay on a consistent, easy to follow rhythm relative to the rest of the song's more complex notes and instruments (good example of consistent beat plus complex evolving rhythms across the song [6]).

Additionally, for every iteration, I also made the model automatically produce a sample after every epoch, and then produce and postprocess a large number of samples after training. All the samples, weights, and model data of all my iterations across different models (even for each epoch for each model) can be found in my Github page for the project [3] (the 'SAMPLES & OLDER MODEL DATA' folder). The direct-from-model produced and postprocessed samples are in the 'produced samples' and 'postprocessed samples' folders respectively.

Using Ableton, I was able to add instruments by hand to some of the produced samples to make Bollywood and other genre songs using the frequency classification method mentioned earlier. In fact, although the note progressions most closely resemble Bollywood songs, I was able to simply play around with different instruments and apply them to notes in frequency groups to make them sound

[5] Sample from final version of model: https://drive.google.com/file/d/1Xv6Rg5nOuUeIsUPY_pbOfSCvCKaHzbt-/view?usp=sharing
[6] Consistent beat with evolving chorus good sample: https://drive.google.com/file/d/1JgP7-EsZP41KWOi32dtP_fu72OVJqSgn/view?usp=sharing

like songs of entirely different genres as well. These are the best final samples produced from this project [7] [8] [9].

Future Scope

There are several steps that could be taken to improve the algorithm and create more generalized music generators for Bollywood music.

1. More training data: in trying to acquire more training data, I found several sources online that can also provide more .mid MIDI data of Bollywood music, both paid and free. [6] [7] [8]
2. Longer training time: On my current model, one epoch with all the training data takes ~1 hour. The loss of the model currently seems to slowly decrease across epochs. Using the campus servers at DKU or other external high-end GPU resources will allow me to train for many more epochs and create a model that produces more unique and varying samples.
3. Encoder & Decoder model for Multi-instrument Support: Create another model to encode both the musical data of the notes, durations and offsets along with which instrument they're played by to then pass as input to this RNN and finally to a decoder that directly outputs low-level MIDI data to generate multi-instrument music.
4. More Optimizations: Due to time constraints of this project, I was mostly busy with parsing the music and tuning the RNN model itself, and could not try many of the other optimisations I found in my literature review (considering melodies as words and applying NLP methods, a simultaneous reinforcement learning model to provide a secondary optimisation objective, and some other optimisations mentioned in the proposal for this project, etc.). Many of these optimisations could be experimented with to improve the model.

Appendix

[i]: Original Network v0 Summary: See https://github.com/hedonistrh/bestekar/blob/master/LSTM_colab.ipynb

[ii]: Simplified Network v1 Summary:

```
Model: "model_1"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            (None, 15, 128)      0
__________________________________________________________________________________________________
lstm_1 (LSTM)                   (None, 15, 1024)     4722688     input_1[0][0]
__________________________________________________________________________________________________
leaky_re_lu_1 (LeakyReLU)       (None, 15, 1024)     0           lstm_1[0][0]
__________________________________________________________________________________________________
batch_normalization_1 (BatchNor (None, 15, 1024)     4096        leaky_re_lu_1[0][0]
__________________________________________________________________________________________________
dropout_1 (Dropout)             (None, 15, 1024)     0           batch_normalization_1[0][0]
```

[7] Full Song 1 (Synthwave): https://youtu.be/95FGbHAVr8E
[8] Full Song 2 (Upbeat Electronic): https://youtu.be/pqya_VlJ7Vs
[9] Full Song 3 (Modern Bollywood): https://youtu.be/ckcQ1K9efvk

```
__________________________________________________________________________________________
__________
dense_1 (Dense)                 (None, 15, 1)        1025        dropout_1[0][0]
__________________________________________________________________________________________
__________
flatten_1 (Flatten)             (None, 15)           0           dense_1[0][0]
__________________________________________________________________________________________
__________
activation_1 (Activation)       (None, 15)           0           flatten_1[0][0]
__________________________________________________________________________________________
__________
repeat_vector_1 (RepeatVector)  (None, 1024, 15)     0           activation_1[0][0]
__________________________________________________________________________________________
__________
permute_1 (Permute)             (None, 15, 1024)     0           repeat_vector_1[0][0]
__________________________________________________________________________________________
__________
multiply_1 (Multiply)           (None, 15, 1024)     0           dropout_1[0][0]
                                                                 permute_1[0][0]
__________________________________________________________________________________________
__________
dense_2 (Dense)                 (None, 15, 512)      524800      multiply_1[0][0]
__________________________________________________________________________________________
__________
dense_3 (Dense)                 (None, 15, 256)      131328      dense_2[0][0]
__________________________________________________________________________________________
__________
leaky_re_lu_2 (LeakyReLU)       (None, 15, 256)      0           dense_3[0][0]
__________________________________________________________________________________________
__________
batch_normalization_2 (BatchNor (None, 15, 256)      1024        leaky_re_lu_2[0][0]
__________________________________________________________________________________________
__________
dropout_2 (Dropout)             (None, 15, 256)      0           batch_normalization_2[0][0]
__________________________________________________________________________________________
__________
lstm_2 (LSTM)                   (None, 15, 256)      525312      dropout_2[0][0]
__________________________________________________________________________________________
__________
leaky_re_lu_3 (LeakyReLU)       (None, 15, 256)      0           lstm_2[0][0]
__________________________________________________________________________________________
__________
batch_normalization_3 (BatchNor (None, 15, 256)      1024        leaky_re_lu_3[0][0]
__________________________________________________________________________________________
__________
dropout_3 (Dropout)             (None, 15, 256)      0           batch_normalization_3[0][0]
__________________________________________________________________________________________
__________
dense_4 (Dense)                 (None, 15, 1)        257         dropout_3[0][0]
__________________________________________________________________________________________
__________
flatten_2 (Flatten)             (None, 15)           0           dense_4[0][0]
__________________________________________________________________________________________
__________
activation_2 (Activation)       (None, 15)           0           flatten_2[0][0]
__________________________________________________________________________________________
__________
```

```
repeat_vector_2 (RepeatVector)  (None, 256, 15)      0          activation_2[0][0]
________________________________________________________________________________
__________________
permute_2 (Permute)            (None, 15, 256)      0          repeat_vector_2[0][0]
________________________________________________________________________________
__________________
multiply_2 (Multiply)          (None, 15, 256)      0          dropout_3[0][0]
                                                               permute_2[0][0]
________________________________________________________________________________
__________________
dense_5 (Dense)                (None, 15, 128)      32896      multiply_2[0][0]
________________________________________________________________________________
__________________
dense_6 (Dense)                (None, 15, 128)      16512      dense_5[0][0]
________________________________________________________________________________
__________________
leaky_re_lu_4 (LeakyReLU)      (None, 15, 128)      0          dense_6[0][0]
________________________________________________________________________________
__________________
batch_normalization_4 (BatchNor (None, 15, 128)      512        leaky_re_lu_4[0][0]
________________________________________________________________________________
__________________
dropout_4 (Dropout)            (None, 15, 128)      0          batch_normalization_4[0][0]
________________________________________________________________________________
__________________
lstm_3 (LSTM)                  (None, 128)          131584     dropout_4[0][0]
________________________________________________________________________________
__________________
leaky_re_lu_5 (LeakyReLU)      (None, 128)          0          lstm_3[0][0]
________________________________________________________________________________
__________________
batch_normalization_5 (BatchNor (None, 128)          512        leaky_re_lu_5[0][0]
________________________________________________________________________________
__________________
dropout_5 (Dropout)            (None, 128)          0          batch_normalization_5[0][0]
________________________________________________________________________________
__________________
dense_7 (Dense)                (None, 128)          16512      dropout_5[0][0]
================================================================================
======================
Total params: 6,110,082
Trainable params: 6,106,498
Non-trainable params: 3,584
__________________________________________________________________
```

[iii]: Simplified network v2 Summary:

```
Model: "model_2"
________________________________________________________________________________
__________________
Layer (type)                   Output Shape         Param #     Connected to
================================================================================
======================
input_2 (InputLayer)           (None, 15, 128)      0
________________________________________________________________________________
__________________
lstm_4 (LSTM)                  (None, 15, 512)      1312768     input_2[0][0]
```

```
__________________________________________________________________________________________________
leaky_re_lu_6 (LeakyReLU)       (None, 15, 512)      0           lstm_4[0][0]
__________________________________________________________________________________________________
batch_normalization_6 (BatchNor (None, 15, 512)      2048        leaky_re_lu_6[0][0]
__________________________________________________________________________________________________
dropout_6 (Dropout)             (None, 15, 512)      0           batch_normalization_6[0][0]
__________________________________________________________________________________________________
dense_8 (Dense)                 (None, 15, 1)        513         dropout_6[0][0]
__________________________________________________________________________________________________
flatten_3 (Flatten)             (None, 15)           0           dense_8[0][0]
__________________________________________________________________________________________________
activation_3 (Activation)       (None, 15)           0           flatten_3[0][0]
__________________________________________________________________________________________________
repeat_vector_3 (RepeatVector)  (None, 512, 15)      0           activation_3[0][0]
__________________________________________________________________________________________________
permute_3 (Permute)             (None, 15, 512)      0           repeat_vector_3[0][0]
__________________________________________________________________________________________________
multiply_3 (Multiply)           (None, 15, 512)      0           dropout_6[0][0]
                                                                 permute_3[0][0]
__________________________________________________________________________________________________
dense_9 (Dense)                 (None, 15, 512)      262656      multiply_3[0][0]
__________________________________________________________________________________________________
dense_10 (Dense)                (None, 15, 256)      131328      dense_9[0][0]
__________________________________________________________________________________________________
leaky_re_lu_7 (LeakyReLU)       (None, 15, 256)      0           dense_10[0][0]
__________________________________________________________________________________________________
batch_normalization_7 (BatchNor (None, 15, 256)      1024        leaky_re_lu_7[0][0]
__________________________________________________________________________________________________
dropout_7 (Dropout)             (None, 15, 256)      0           batch_normalization_7[0][0]
__________________________________________________________________________________________________
lstm_5 (LSTM)                   (None, 15, 256)      525312      dropout_7[0][0]
__________________________________________________________________________________________________
leaky_re_lu_8 (LeakyReLU)       (None, 15, 256)      0           lstm_5[0][0]
__________________________________________________________________________________________________
batch_normalization_8 (BatchNor (None, 15, 256)      1024        leaky_re_lu_8[0][0]
__________________________________________________________________________________________________
dropout_8 (Dropout)             (None, 15, 256)      0           batch_normalization_8[0][0]
__________________________________________________________________________________________________
```

```
dense_11 (Dense)              (None, 15, 1)       257        dropout_8[0][0]
________________________________________________________________________________
__________________
flatten_4 (Flatten)           (None, 15)          0          dense_11[0][0]
________________________________________________________________________________
__________________
activation_4 (Activation)     (None, 15)          0          flatten_4[0][0]
________________________________________________________________________________
__________________
repeat_vector_4 (RepeatVector)  (None, 256, 15)   0          activation_4[0][0]
________________________________________________________________________________
__________________
permute_4 (Permute)           (None, 15, 256)     0          repeat_vector_4[0][0]
________________________________________________________________________________
__________________
multiply_4 (Multiply)         (None, 15, 256)     0          dropout_8[0][0]
                                                             permute_4[0][0]
________________________________________________________________________________
__________________
dense_12 (Dense)              (None, 15, 128)     32896      multiply_4[0][0]
________________________________________________________________________________
__________________
dense_13 (Dense)              (None, 15, 128)     16512      dense_12[0][0]
________________________________________________________________________________
__________________
leaky_re_lu_9 (LeakyReLU)     (None, 15, 128)     0          dense_13[0][0]
________________________________________________________________________________
__________________
batch_normalization_9 (BatchNor (None, 15, 128)   512        leaky_re_lu_9[0][0]
________________________________________________________________________________
__________________
dropout_9 (Dropout)           (None, 15, 128)     0          batch_normalization_9[0][0]
________________________________________________________________________________
__________________
lstm_6 (LSTM)                 (None, 128)         131584     dropout_9[0][0]
________________________________________________________________________________
__________________
leaky_re_lu_10 (LeakyReLU)    (None, 128)         0          lstm_6[0][0]
________________________________________________________________________________
__________________
batch_normalization_10 (BatchNo (None, 128)       512        leaky_re_lu_10[0][0]
________________________________________________________________________________
__________________
dropout_10 (Dropout)          (None, 128)         0          batch_normalization_10[0][0]
________________________________________________________________________________
__________________
dense_14 (Dense)              (None, 128)         16512      dropout_10[0][0]
================================================================================
==================
Total params: 2,435,458
Trainable params: 2,432,898
Non-trainable params: 2,560
________________________________________________________________________________
__________________
```

[iv] Network Model v3 Summary (Added Bidirectionality to first 2 LSTM layers):

```
Model: "model_4"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_7 (InputLayer)            (None, 15, 128)      0
__________________________________________________________________________________________________
bidirectional_5 (Bidirectional) (None, 15, 1024)     2625536     input_7[0][0]
__________________________________________________________________________________________________
leaky_re_lu_20 (LeakyReLU)      (None, 15, 1024)     0           bidirectional_5[0][0]
__________________________________________________________________________________________________
batch_normalization_20 (BatchNo (None, 15, 1024)     4096        leaky_re_lu_20[0][0]
__________________________________________________________________________________________________
dropout_20 (Dropout)            (None, 15, 1024)     0           batch_normalization_20[0][0]
__________________________________________________________________________________________________
dense_27 (Dense)                (None, 15, 1)        1025        dropout_20[0][0]
__________________________________________________________________________________________________
flatten_10 (Flatten)            (None, 15)           0           dense_27[0][0]
__________________________________________________________________________________________________
activation_10 (Activation)      (None, 15)           0           flatten_10[0][0]
__________________________________________________________________________________________________
repeat_vector_10 (RepeatVector) (None, 1024, 15)     0           activation_10[0][0]
__________________________________________________________________________________________________
permute_10 (Permute)            (None, 15, 1024)     0           repeat_vector_10[0][0]
__________________________________________________________________________________________________
multiply_10 (Multiply)          (None, 15, 1024)     0           dropout_20[0][0]
                                                                 permute_10[0][0]
__________________________________________________________________________________________________
dense_28 (Dense)                (None, 15, 512)      524800      multiply_10[0][0]
__________________________________________________________________________________________________
dense_29 (Dense)                (None, 15, 256)      131328      dense_28[0][0]
__________________________________________________________________________________________________
leaky_re_lu_21 (LeakyReLU)      (None, 15, 256)      0           dense_29[0][0]
__________________________________________________________________________________________________
batch_normalization_21 (BatchNo (None, 15, 256)      1024        leaky_re_lu_21[0][0]
__________________________________________________________________________________________________
dropout_21 (Dropout)            (None, 15, 256)      0           batch_normalization_21[0][0]
__________________________________________________________________________________________________
bidirectional_6 (Bidirectional) (None, 15, 512)      1050624     dropout_21[0][0]
```

```
__________________________________________________________________________________________________
leaky_re_lu_22 (LeakyReLU)      (None, 15, 512)      0           bidirectional_6[0][0]
__________________________________________________________________________________________________
batch_normalization_22 (BatchNo (None, 15, 512)      2048        leaky_re_lu_22[0][0]
__________________________________________________________________________________________________
dropout_22 (Dropout)            (None, 15, 512)      0           batch_normalization_22[0][0]
__________________________________________________________________________________________________
dense_30 (Dense)                (None, 15, 1)        513         dropout_22[0][0]
__________________________________________________________________________________________________
flatten_11 (Flatten)            (None, 15)           0           dense_30[0][0]
__________________________________________________________________________________________________
activation_11 (Activation)      (None, 15)           0           flatten_11[0][0]
__________________________________________________________________________________________________
repeat_vector_11 (RepeatVector) (None, 512, 15)      0           activation_11[0][0]
__________________________________________________________________________________________________
permute_11 (Permute)            (None, 15, 512)      0           repeat_vector_11[0][0]
__________________________________________________________________________________________________
multiply_11 (Multiply)          (None, 15, 512)      0           dropout_22[0][0]
                                                                 permute_11[0][0]
__________________________________________________________________________________________________
dense_31 (Dense)                (None, 15, 128)      65664       multiply_11[0][0]
__________________________________________________________________________________________________
dense_32 (Dense)                (None, 15, 128)      16512       dense_31[0][0]
__________________________________________________________________________________________________
leaky_re_lu_23 (LeakyReLU)      (None, 15, 128)      0           dense_32[0][0]
__________________________________________________________________________________________________
batch_normalization_23 (BatchNo (None, 15, 128)      512         leaky_re_lu_23[0][0]
__________________________________________________________________________________________________
dropout_23 (Dropout)            (None, 15, 128)      0           batch_normalization_23[0][0]
__________________________________________________________________________________________________
lstm_15 (LSTM)                  (None, 128)          131584      dropout_23[0][0]
__________________________________________________________________________________________________
leaky_re_lu_24 (LeakyReLU)      (None, 128)          0           lstm_15[0][0]
__________________________________________________________________________________________________
batch_normalization_24 (BatchNo (None, 128)          512         leaky_re_lu_24[0][0]
__________________________________________________________________________________________________
dropout_24 (Dropout)            (None, 128)          0           batch_normalization_24[0][0]
__________________________________________________________________________________________________
```

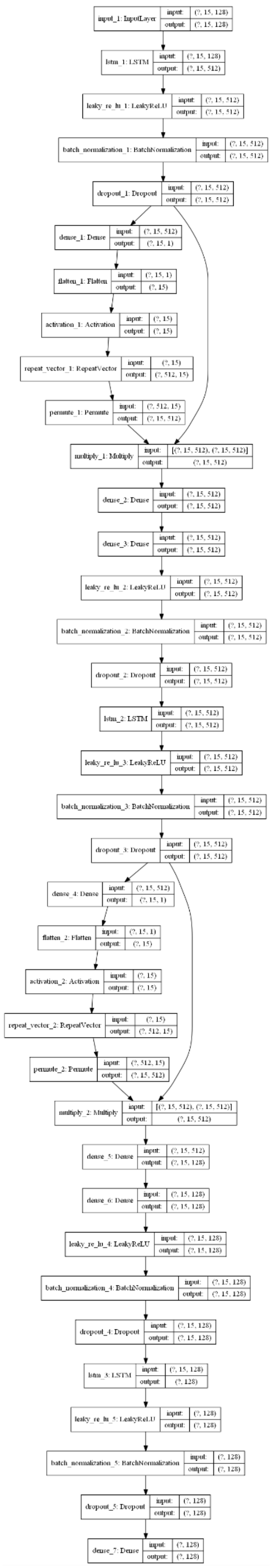
```
dense_33 (Dense)              (None, 128)         16512       dropout_24[0][0]
==================================================================================
=====================
Total params: 4,572,290
Trainable params: 4,568,194
Non-trainable params: 4,096
__________________________________________________________________________________
_____________
```

[v]: Removed bidirectionality and doubling number of activations in 2nd major layer, model v4:

```
Model: "model_1"
________________________________________________________________________________
__________________________________
Layer (type)                    Output Shape         Param #     Connec
ted to
================================================================================
==================================
input_1 (InputLayer)            (None, 15, 128)      0


________________________________________________________________________________
__________________________________
lstm_1 (LSTM)                   (None, 15, 512)      1312768     input_
1[0][0]
________________________________________________________________________________
__________________________________
leaky_re_lu_1 (LeakyReLU)       (None, 15, 512)      0           lstm_1
[0][0]
________________________________________________________________________________
__________________________________
batch_normalization_1 (BatchNor (None, 15, 512)      2048        leaky_
re_lu_1[0][0]
________________________________________________________________________________
__________________________________
dropout_1 (Dropout)             (None, 15, 512)      0           batch_
normalization_1[0][0]
________________________________________________________________________________
__________________________________
dense_1 (Dense)                 (None, 15, 1)        513         dropou
t_1[0][0]
________________________________________________________________________________
__________________________________
flatten_1 (Flatten)             (None, 15)           0           dense_
1[0][0]
________________________________________________________________________________
__________________________________
activation_1 (Activation)       (None, 15)           0           flatte
n_1[0][0]
________________________________________________________________________________
__________________________________
repeat_vector_1 (RepeatVector)  (None, 512, 15)      0           activa
tion_1[0][0]
________________________________________________________________________________
__________________________________
permute_1 (Permute)             (None, 15, 512)      0           repeat
_vector_1[0][0]
```

```
__________________________________________________________________________
____________________________
multiply_1 (Multiply)          (None, 15, 512)      0            dropou
t_1[0][0]
                                                                 permut
e_1[0][0]
__________________________________________________________________________
____________________________
dense_2 (Dense)                (None, 15, 512)      262656       multip
ly_1[0][0]
__________________________________________________________________________
____________________________
dense_3 (Dense)                (None, 15, 512)      262656       dense_
2[0][0]
__________________________________________________________________________
____________________________
leaky_re_lu_2 (LeakyReLU)      (None, 15, 512)      0            dense_
3[0][0]
__________________________________________________________________________
____________________________
batch_normalization_2 (BatchNor (None, 15, 512)     2048         leaky_
re_lu_2[0][0]
__________________________________________________________________________
____________________________
dropout_2 (Dropout)            (None, 15, 512)      0            batch_
normalization_2[0][0]
__________________________________________________________________________
____________________________
lstm_2 (LSTM)                  (None, 15, 512)      2099200      dropou
t_2[0][0]
__________________________________________________________________________
____________________________
leaky_re_lu_3 (LeakyReLU)      (None, 15, 512)      0            lstm_2
[0][0]
__________________________________________________________________________
____________________________
batch_normalization_3 (BatchNor (None, 15, 512)     2048         leaky_
re_lu_3[0][0]
__________________________________________________________________________
____________________________
dropout_3 (Dropout)            (None, 15, 512)      0            batch_
normalization_3[0][0]
__________________________________________________________________________
____________________________
dense_4 (Dense)                (None, 15, 1)        513          dropou
t_3[0][0]
__________________________________________________________________________
____________________________
flatten_2 (Flatten)            (None, 15)           0            dense_
4[0][0]
__________________________________________________________________________
____________________________
activation_2 (Activation)      (None, 15)           0            flatte
n_2[0][0]
__________________________________________________________________________
____________________________
repeat_vector_2 (RepeatVector) (None, 512, 15)      0            activa
tion_2[0][0]
```

```
________________________________________________________________________
___________________________
permute_2 (Permute)          (None, 15, 512)      0            repeat
_vector_2[0][0]
________________________________________________________________________
___________________________
multiply_2 (Multiply)        (None, 15, 512)      0            dropou
t_3[0][0]
                                                               permut
e_2[0][0]
________________________________________________________________________
___________________________
dense_5 (Dense)              (None, 15, 128)      65664        multip
ly_2[0][0]
________________________________________________________________________
___________________________
dense_6 (Dense)              (None, 15, 128)      16512        dense_
5[0][0]
________________________________________________________________________
___________________________
leaky_re_lu_4 (LeakyReLU)    (None, 15, 128)      0            dense_
6[0][0]
________________________________________________________________________
___________________________
batch_normalization_4 (BatchNor (None, 15, 128)   512          leaky_
re_lu_4[0][0]
________________________________________________________________________
___________________________
dropout_4 (Dropout)          (None, 15, 128)      0            batch_
normalization_4[0][0]
________________________________________________________________________
___________________________
lstm_3 (LSTM)                (None, 128)          131584       dropou
t_4[0][0]
________________________________________________________________________
___________________________
leaky_re_lu_5 (LeakyReLU)    (None, 128)          0            lstm_3
[0][0]
________________________________________________________________________
___________________________
batch_normalization_5 (BatchNor (None, 128)       512          leaky_
re_lu_5[0][0]
________________________________________________________________________
___________________________
dropout_5 (Dropout)          (None, 128)          0            batch_
normalization_5[0][0]
________________________________________________________________________
___________________________
dense_7 (Dense)              (None, 128)          16512        dropou
t_5[0][0]
========================================================================
===========================
Total params: 4,175,746
Trainable params: 4,172,162
Non-trainable params: 3,584
________________________________________________________________________
___________________________
```
[vi]: Final Model Visualisation:


input_1: InputLayer
input: (?, 15, 128)
output: (?, 15, 128)
lstm_1: LSTM
input: (?, 15, 128)
output: (?, 15, 512)
leaky_re_lu_1: LeakyReLU
input: (?, 15, 512)
output: (?, 15, 512)
batch_normalization_1: BatchNormalization
input: (?, 15, 512)
output: (?, 15, 512)
dropout_1: Dropout
input: (?, 15, 512)
output: (?, 15, 512)
dense_1: Dense
input: (?, 15, 512)
output: (?, 15, 1)
flatten_1: Flatten
input: (?, 15, 1)
output: (?, 15)
activation_1: Activation
input: (?, 15)
output: (?, 15)
repeat_vector_1: RepeatVector
input: (?, 15)
output: (?, 512, 15)
permute_1: Permute
input: (?, 512, 15)
output: (?, 15, 512)
multiply_1: Multiply
input: [(?, 15, 512), (?, 15, 512)]
output: (?, 15, 512)
dense_2: Dense
input: (?, 15, 512)
output: (?, 15, 512)
dense_3: Dense
input: (?, 15, 512)
output: (?, 15, 512)
leaky_re_lu_2: LeakyReLU
input: (?, 15, 512)
output: (?, 15, 512)
batch_normalization_2: BatchNormalization
input: (?, 15, 512)
output: (?, 15, 512)
dropout_2: Dropout
input: (?, 15, 512)
output: (?, 15, 512)
lstm_2: LSTM
input: (?, 15, 512)
output: (?, 15, 512)
leaky_re_lu_3: LeakyReLU
input: (?, 15, 512)
output: (?, 15, 512)
batch_normalization_3: BatchNormalization
input: (?, 15, 512)
output: (?, 15, 512)
dropout_3: Dropout
input: (?, 15, 512)
output: (?, 15, 512)
dense_4: Dense
input: (?, 15, 512)
output: (?, 15, 1)
flatten_2: Flatten
input: (?, 15, 1)
output: (?, 15)
activation_2: Activation
input: (?, 15)
output: (?, 15)
repeat_vector_2: RepeatVector
input: (?, 15)
output: (?, 512, 15)
permute_2: Permute
input: (?, 512, 15)
output: (?, 15, 512)
multiply_2: Multiply
input: [(?, 15, 512), (?, 15, 512)]
output: (?, 15, 512)
dense_5: Dense
input: (?, 15, 512)
output: (?, 15, 128)
dense_6: Dense
input: (?, 15, 128)
output: (?, 15, 128)
leaky_re_lu_4: LeakyReLU
input: (?, 15, 128)
output: (?, 15, 128)
batch_normalization_4: BatchNormalization
input: (?, 15, 128)
output: (?, 15, 128)
dropout_4: Dropout
input: (?, 15, 128)
output: (?, 15, 128)
lstm_3: LSTM
input: (?, 15, 128)
output: (?, 128)
leaky_re_lu_5: LeakyReLU
input: (?, 128)
output: (?, 128)
batch_normalization_5: BatchNormalization
input: (?, 128)
output: (?, 128)
dropout_5: Dropout
input: (?, 128)
output: (?, 128)
dense_7: Dense
input: (?, 128)
output: (?, 128)

[vii]: Loss From Each Epoch from 28 epoch training of Bidirectional Model v3:

```
Epoch: 1

Epoch 1/1

63980/63980 [==============================] - 1172s 18ms/step -
loss: 3.9004

Epoch: 2

Epoch 1/1

63980/63980 [==============================] - 1181s 18ms/step -
loss: 4.1034

Epoch: 3

Epoch 1/1

63980/63980 [==============================] - 1194s 19ms/step -
loss: 4.1888

Epoch: 4

Epoch 1/1

63980/63980 [==============================] - 1189s 19ms/step -
loss: 4.0126

Epoch: 5

Epoch 1/1

63980/63980 [==============================] - 1192s 19ms/step -
loss: 3.7785

Epoch: 6

Epoch 1/1

63980/63980 [==============================] - 1197s 19ms/step -
loss: 3.6259

Epoch: 7

Epoch 1/1

63980/63980 [==============================] - 1197s 19ms/step -
loss: 3.5493

Epoch: 8

Epoch 1/1

63980/63980 [==============================] - 1196s 19ms/step -
loss: 3.5359

Epoch: 9

Epoch 1/1
```

```
63980/63980 [==============================] - 1196s 19ms/step -
loss: 3.5352

Epoch: 10

Epoch 1/1

63980/63980 [==============================] - 1195s 19ms/step -
loss: 3.5439

Epoch: 11

Epoch 1/1

63980/63980 [==============================] - 1195s 19ms/step -
loss: 3.5742

Epoch: 12

Epoch 1/1

63980/63980 [==============================] - 1195s 19ms/step -
loss: 3.5961

Epoch: 13

Epoch 1/1

63980/63980 [==============================] - 1195s 19ms/step -
loss: 3.6463

Epoch: 14

Epoch 1/1

63980/63980 [==============================] - 1195s 19ms/step -
loss: 3.6884

Epoch: 15

Epoch 1/1

63980/63980 [==============================] - 1196s 19ms/step -
loss: 3.7323

Epoch: 16

Epoch 1/1

63980/63980 [==============================] - 1196s 19ms/step -
loss: 3.7728

Epoch: 17

Epoch 1/1

63980/63980 [==============================] - 1194s 19ms/step -
loss: 3.8092

Epoch: 18

Epoch 1/1
```

```
63980/63980 [==============================] - 1195s 19ms/step -
loss: 3.8395

Epoch: 19

Epoch 1/1

63980/63980 [==============================] - 1195s 19ms/step -
loss: 3.9040

Epoch: 20

Epoch 1/1

63980/63980 [==============================] - 1195s 19ms/step -
loss: 3.9423

Epoch: 21

Epoch 1/1

63980/63980 [==============================] - 1195s 19ms/step -
loss: 3.9724

Epoch: 22

Epoch 1/1

63980/63980 [==============================] - 1196s 19ms/step -
loss: 4.0245

Epoch: 23

Epoch 1/1

63980/63980 [==============================] - 1195s 19ms/step -
loss: 4.0530

Epoch: 24

Epoch 1/1

63980/63980 [==============================] - 1196s 19ms/step -
loss: 4.1011

Epoch: 25

Epoch 1/1

63980/63980 [==============================] - 1198s 19ms/step -
loss: 4.1610

Epoch: 26

Epoch 1/1

63980/63980 [==============================] - 1196s 19ms/step -
loss: 4.2250

Epoch: 27

Epoch 1/1
```

```
63980/63980 [==============================] - 1211s 19ms/step -
loss: 4.2316

Epoch: 28

Epoch 1/1

63980/63980 [==============================] - 1185s 19ms/step -
loss: 4.2805
```

References

[1]: Source of training data audio: http://jsplash.imess.net/midisite/hindi.htm

[2]: The blog post that the model is most similar to: https://hedonistrh.github.io/2018-04-27-Music-Generation-with-LSTM/

[3]: Github Repository Of Project With All Documentation, Code & Samples: https://github.com/Aryan-Poonacha/bollywoodmusicgenerator

[4]: Docevski, Marko & Zdravevski, Eftim & Lameski, Petre & Kulakov, Andrea. (2018). Towards Music Generation With Deep Learning Algorithms.

[5]: Limitation of music21 for changing instruments for different notes: https://stackoverflow.com/questions/63032999/output-more-than-one-instrument-using-music-21-and-python

[6]: Paid access to more training data (Bollywood MIDI piano files): https://bollypiano.com/product-category/hindi-song-midi-files/

[7]: More free training data: https://freemidibyarupix.blogspot.com/2019/01/free-midi-by-arupix-it-is-temprory-link.html

[8]: More paid training data: https://www.midiindia.com/