CS60075:

Natural Language Processing

Project:

Music Generation using LSTM

Group : 32

Team Members:

- Sourya Roy (19EC30053)
- Omkar Modi (19EE30018)
- Mansi Uniyal (19EE10039)
- Anubhav Krishna Saraf (19IE10043)

TASK:

The main workflow has been to identify the problem statement, visualize and analyse the same, and then to come up with the solution to utmost performance. Here, in the project we have worked on automated music generation using various sequential models achieving a realistic performance on generation. The code provided alongside the report consists of major sections of:

1. Data exploration and Visualization
2. Data preprocessing
3. Model building
4. Evaluation and Results

In Natural Language Processing, we generally deal with classical languages consisting of words which not only contain a superficial meaning but also an underlying sentiment. But, here in this project instead of language having words we are going to deal with music containing notes. Just as words have meanings, notes represent special sounds, just as words have sentiments, music has tempo, just as words follow a particular grammar, notes follow a particular rhythm, etc.

There exist many models to predict/generate words. But here we propose to design an algorithm which generates Musical notes. These notes are represented in musical instrument digital interface (MIDI) files. We will try to understand more about these MIDI files and the data they hold and how we can retrieve it to make sense for human understanding. Main objective here is to provide an algorithm which can be used to generate musical notes using Recurrent Neural Networks (RNN), principally Long Short-Term Memory (LSTM) networks.

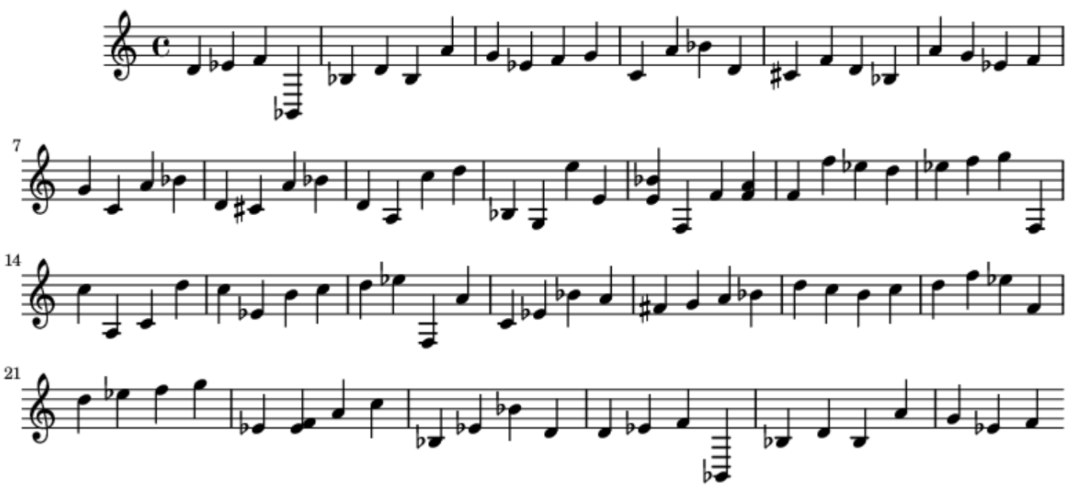
DATASETS:

Just like in languages where words are formed by letters, music is expressed via notes which basically represent sounds with different pitch, amplitude, bass, etc. Now, to store and work with these in a digital medium we have Musical Instrument Digital Interface(MIDI) format. MIDI itself does not make sound, it is just a series of messages like "note on", "note of", "note/pitch", "pitch bend", and many more. These messages are interpreted by a MIDI instrument to produce sound. A MIDI instrument can be a piece of hardware (electronic keyboard, synthesizer) or part of a software environment (ableton, garageband, digital performer,etc.)

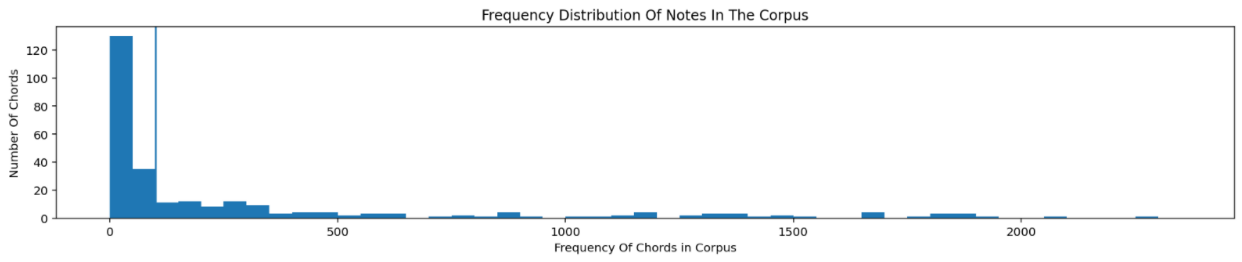
The datasets we are working on are in MIDI format. Now to deal with these datasets we are going to use the library ‘music21’. The dataset includes music from various artists. The work here has been focused on Beethoven's compositions. Now the first task that we have on our hand is to retrieve data from these MIDI files into something more humanly understandable. For this we retrieve data from the MIDI files as musical notes and chords. In music, certain specific combinations of sound waves consisting of standardized wavelength and frequency are notes. A musical note consists of its ‘name’ and its ‘octave’ (e.g. note ‘G2’ here ‘G’ is the name and ‘2’ is its octave). A chord is a collection of multiple notes sounded at the same time. So from each MIDI file we retrieve these data as an array of notes and chords.

ANALYSING DATASET:

The corpus consists of notes like, *'C4', 'C5', 'G3'*... The length of the corpus thus formed is 86436, which consists of 280 unique notes/chords. Moreover, the average frequency of each note is 308, and the most frequent note is “*C4*”, which appeared 2388 time in the corpus.



The extracted information in form of chords and notes, can be visualised for human interpretation using the *Music21* library. The original song is also played in form of .wav file format to provide an audio interface. To analyse the frequency distribution of the data sample, the histogram depicts how the low frequency chords are more frequent, whereas the high frequency chords are very rare. This skewness in the data space can be eliminated by filtering notes above a certain frequency. On applying a frequency threshold of 100, the number of rare notes in the corpus are 165. Thus the length of the modified corpus remains as 82791, with 220 unique notes/chords.



PREPROCESSING DATASET:

With a length of sequence as 40, the corpus is then divided into 82751 uniform sequences. Mapping is done for each unique note to an integer, creating a dictionary to encode the samples. This is also helpful to decode and retrieve the values at the time of prediction. Similarly, the target labels are one hot encoded for training. This creates the resulting structured data with features and target vectors of shape (*82751, 40, 1*) and (*82751, 220*) respectively.
This entire preprocessed data set is then divided into train and test sets for validating the model training and performance. For the same, a (*test_split=0.2, random_state=42*) is used. Thus, dividing the train and test sets with 66200 and 16551 samples, respectively. Having divided the data helps to reduce the issues like overfitting and underfitting of the data, which may cause a loss in generality of the model. The performance on the test set ensures the performance on any unseen data.

METHODOLOGY:

Why Sequential Model?

A sequential model is basically a linear stack of layers. The current output is dependent on the previous input and the length of the input is not fixed.

We can see in the music generation project, the true output at timestep 't' is dependent on all the input notes that the model has seen up to the time step 't'. Here we are considering sequential notes length of 40. Since we don't know the true relationship, we need to come up with an approximation such that the function would depend on all the previous inputs.

$$y(t) = f(x_1,x_2,x_3\ldots.x_t)$$

The function we are defining for sequential model has following characteristic

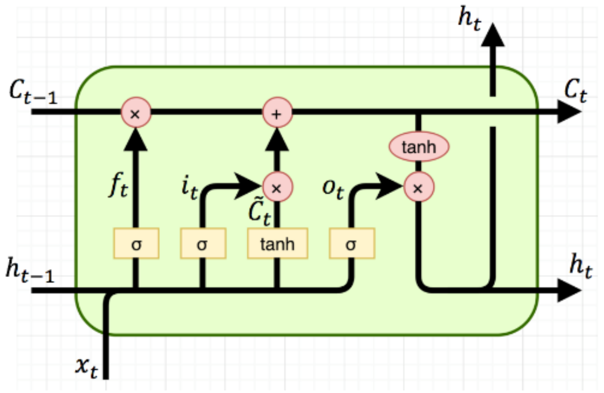
- output Y_t is dependent on previous input
- function can deal with a variable number of inputs
- function executed at each time step is the same

Here we try to predict the next possible musical note based on the previous available notes, thus we need a sequential model to do the same.

Why LSTM?

To understand and predict sequences, the information about previous elements occurring in a sequence is crucial. To predict the next element in a sequence, it is necessary to understand the pattern using previous occurring elements.

Traditional neural networks cannot do this. We need a looping system in a network, allowing the information to persist so that a pattern can be recognized. Such a recurrent network can be thought of as multiple copies of the same network, each passing a message to a successor. This chain-like nature reveals that recurrent neural networks are intimately related to sequences and lists. They're the natural architecture of a neural network to use for such data.
Simple recurrent networks with loops are prone to vanishing gradient problems. That is why we are using a structure with long term dependencies (like Long-short term memory model) to deal with this problem.



In cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information. In the music generation model we are using musical notes played in the past and the gap between relevant information is not as small as to fit it in simple recurrent networks. This is why we are using a special kind of Recurrent Neural Networks called LSTM.

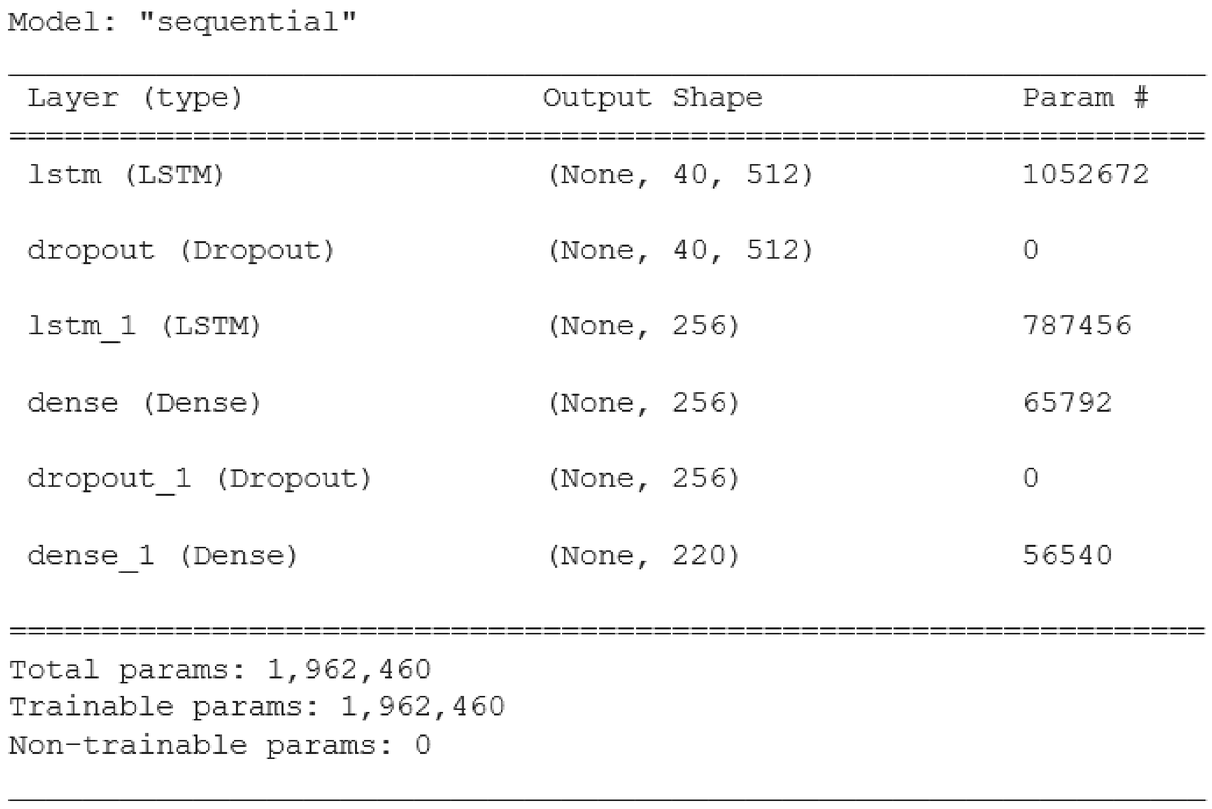
LSTM networks are well-suited to classifying, processing and making predictions based on time series data, since there can be lags of unknown duration between important events in a time series.

Model architecture and parameters:

We have used the following model –

LSTM (seq length 40) $\Longrightarrow$ *dropout(0.1)* $\Longrightarrow$ *Dense(256)* $\Longrightarrow$ *dropout(0.1)* $\Longrightarrow$ *Dense(1)*

Model Summary:

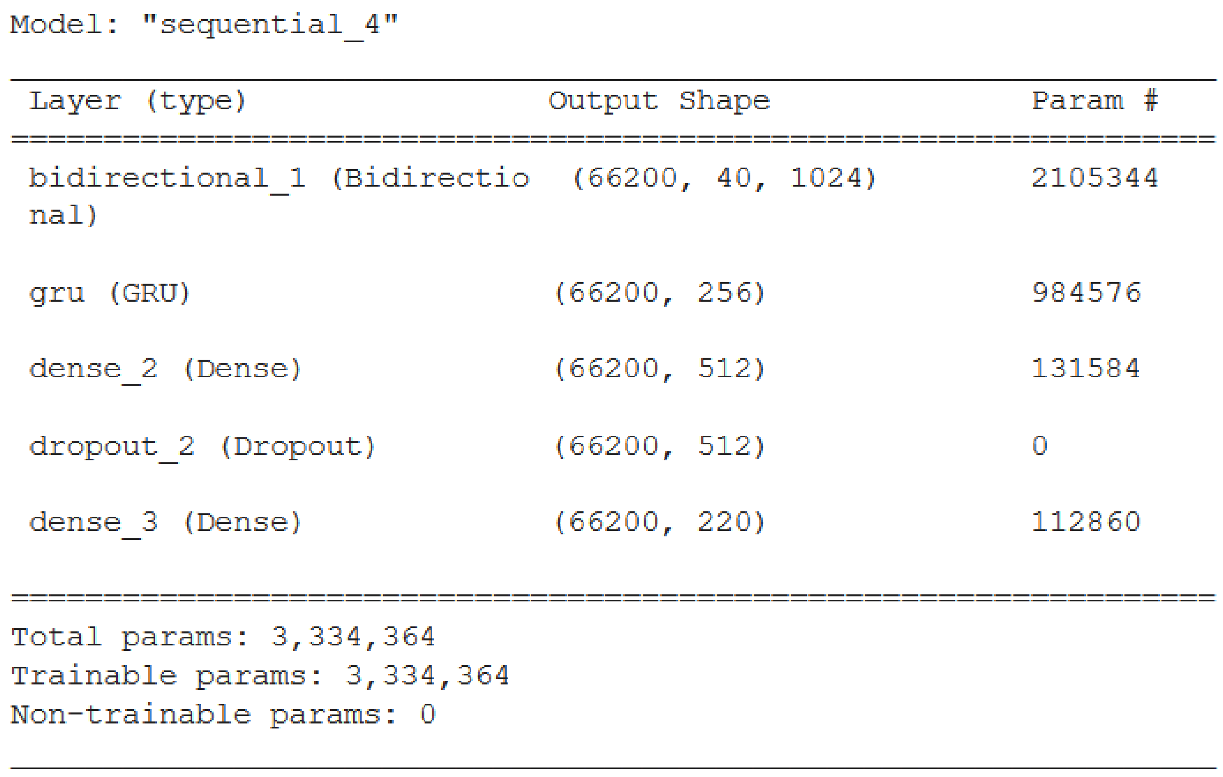


```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 40, 512)           1052672

 dropout (Dropout)           (None, 40, 512)           0

 lstm_1 (LSTM)               (None, 256)               787456

 dense (Dense)               (None, 256)               65792

 dropout_1 (Dropout)         (None, 256)               0

 dense_1 (Dense)             (None, 220)               56540

=================================================================
Total params: 1,962,460
Trainable params: 1,962,460
Non-trainable params: 0
_________________________________________________________________
```

Number of Epochs: ***100***
Batch Size: ***128***

As an alternative approach we have also designed a Bi-LSTM model with a wider history span.

Model Summary for Bi-LSTM Model:



```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 bidirectional_1 (Bidirectio  (66200, 40, 1024)        2105344
 nal)

 gru (GRU)                   (66200, 256)              984576

 dense_2 (Dense)             (66200, 512)              131584

 dropout_2 (Dropout)         (66200, 512)              0

 dense_3 (Dense)             (66200, 220)              112860

=================================================================
Total params: 3,334,364
Trainable params: 3,334,364
Non-trainable params: 0
_________________________________________________________________
```

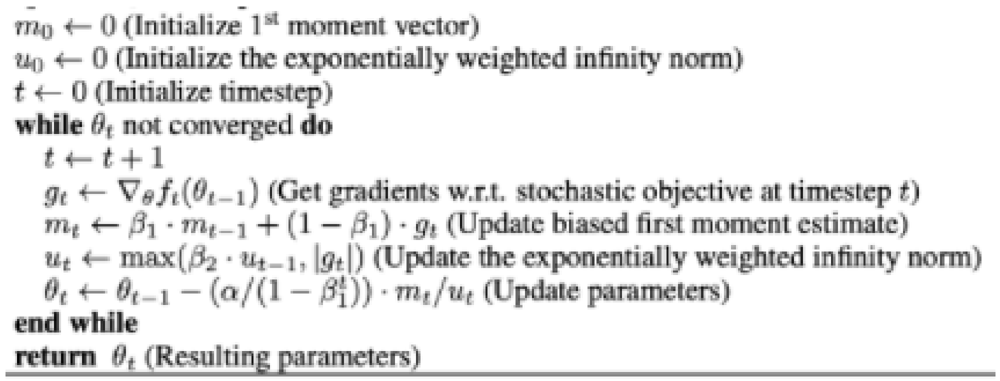
RESULTS:

LOSSES, OPTIMIZER, and METRICS:

While compiling the model the *'categorical_crossentropy'* loss was used for the multiclass classification problem, where the labels are provided in a one hot encoded representation. The loss values are generated from the predicted and target labels with the formula as follows:

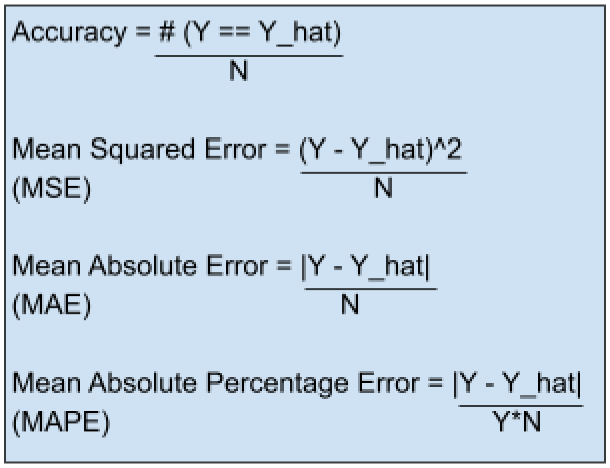
$$-(y_i \log(p_i) + (1 - y_i) \log(1 - p_i))$$

For learning the training weights of the model from the losses, the optimizer used is *Adamax(learning_rate=0.01)*. Adamax is a variant of Adam based on the infinity norm. It is sometimes superior to Adam, especially in models with embeddings. Where the parameters to tune are:
learning_rate: floating point value or a schedule for learning rate. Default: 0.002
beta_1: exponential decay rate for the 1st moment estimates. Default: 0.9
beta_2: exponential decay rate for exponentially weighted infinity norm. Default: 0.999
epsilon: small constant for numerical stability.



```
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$u_0 \leftarrow 0$ (Initialize the exponentially weighted infinity norm)
$t \leftarrow 0$ (Initialize timestep)
while $\theta_t$ not converged do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $u_t \leftarrow \max(\beta_2 \cdot u_{t-1}, |g_t|)$ (Update the exponentially weighted infinity norm)
    $\theta_t \leftarrow \theta_{t-1} - (\alpha/(1 - \beta_1^t)) \cdot m_t/u_t$ (Update parameters)
end while
return $\theta_t$ (Resulting parameters)
```

For training the model, metrics of *accuracy, mean squared error (mse), mean absolute error (mae),* and *mean absolute percentage error (mape)* are used. It evaluates the performance of the thus trained weights and comment on how each parameter enables enhancement in the learning. This enabled the model to focus on performing better by reducing the loss between the predicted and the target label. The figure below denotes the formula for the metrics where,
N = number of samples
Y = actual target value
Y_hat = forecasted value



Accuracy = $\frac{\#\,(Y == Y\_hat)}{N}$

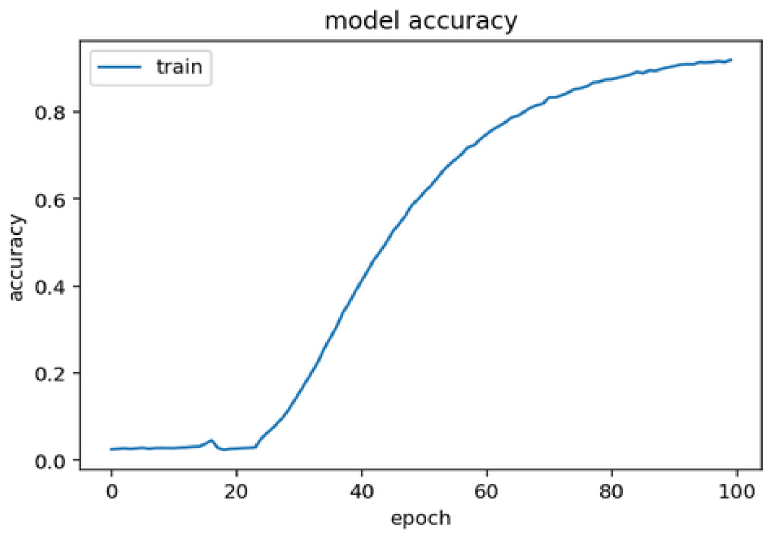
Mean Squared Error (MSE) = $\frac{(Y - Y\_hat)^2}{N}$

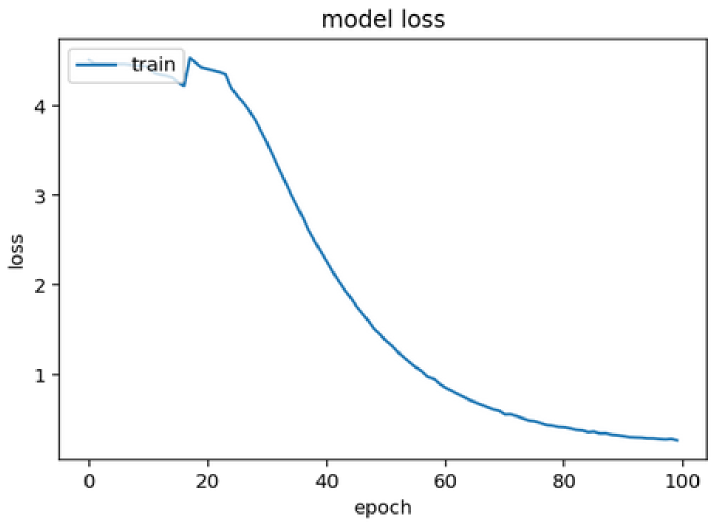
Mean Absolute Error (MAE) = $\frac{|Y - Y\_hat|}{N}$

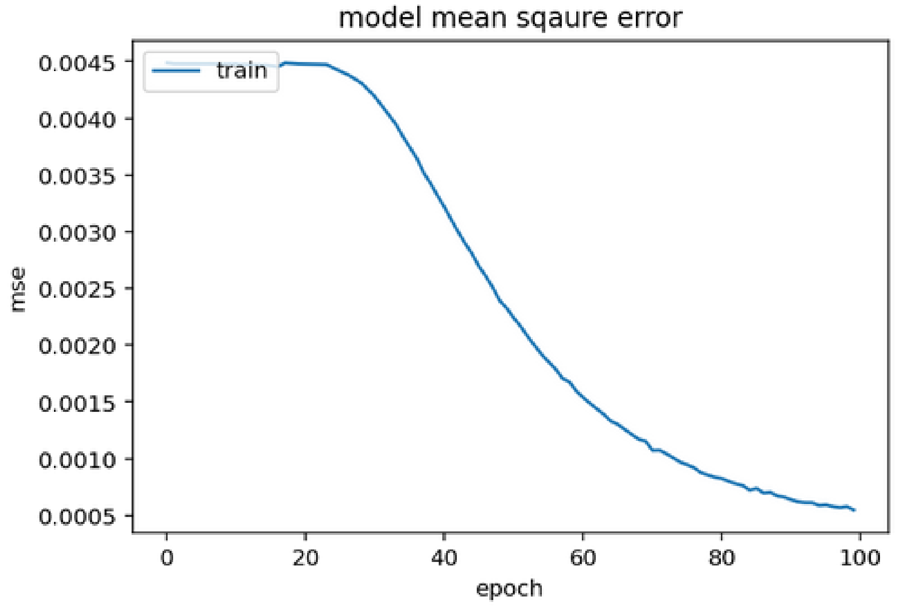
Mean Absolute Percentage Error (MAPE) = $\frac{|Y - Y\_hat|}{Y*N}$

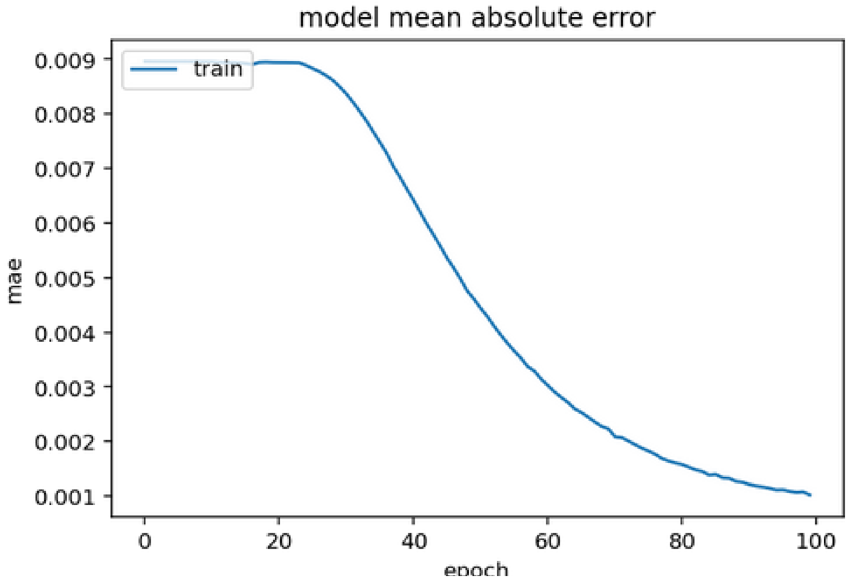
EXPERIMENTS:

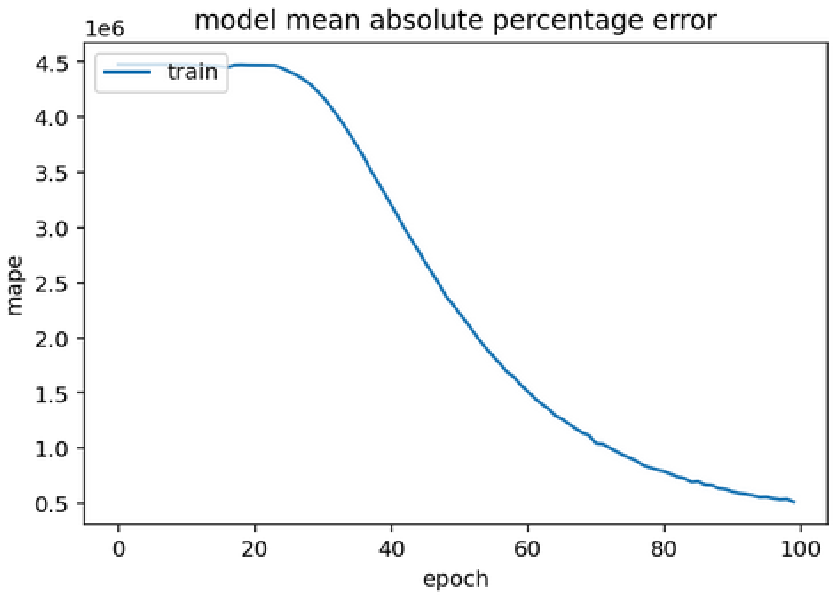
At the end of 100 epochs, the model stabilized to 92% accuracy on the training sample and 92% accuracy on the validation sample. The validation split of 0.2 was ensured while training the model to examine the trend of the learning curves and the losses per epoch. This helps visualize if the model is learning well.





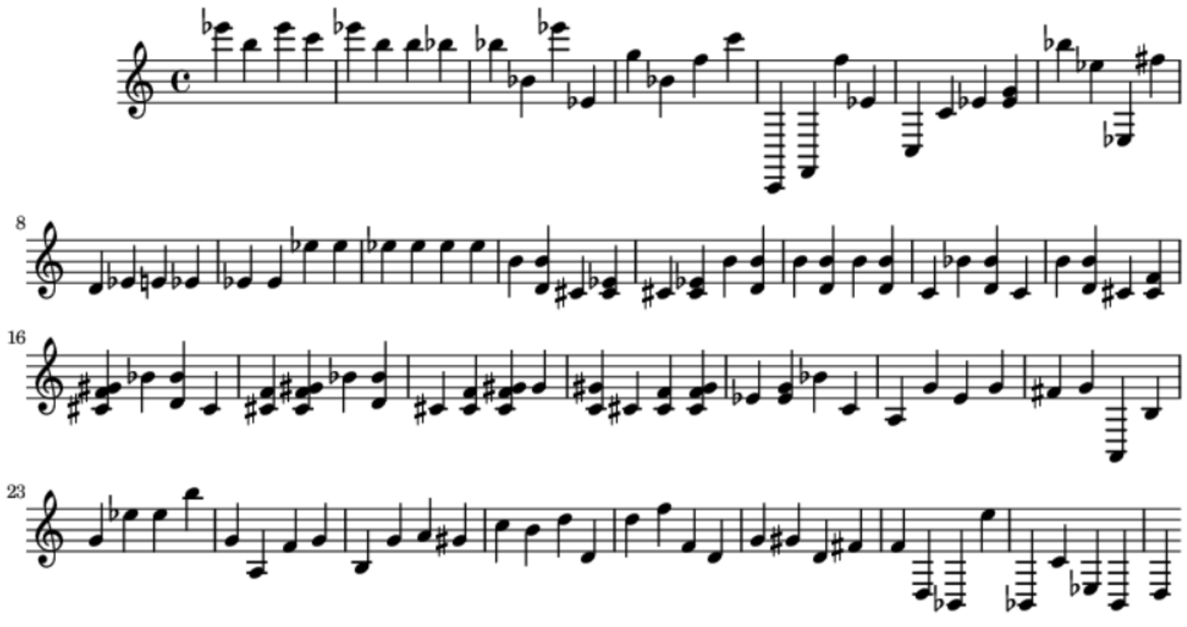

model mean sqaure error
train
mse
0.0045
0.0040
0.0035
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0
20
40
60
80
100
epoch


model mean absolute error
train
mae
0.009
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
20
40
60
80
100
epoch


model mean absolute percentage error
1e6
train
mape
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0
20
40
60
80
100
epoch

In this project we have used Keras because of its simplicity, but we have also tried to replicate the model in Pytorch, which can be seen in the code section.

Lastly, for a random seed from the test sample, we automate generation of musical notes from the model. The predicted class of the notes are then decoded back to the alpha numeric format and visualized. The predicted outcome is even stored in .wav file format to hear the generated music. Here is the predicted outcome for 100 note counts:



To get an audible feel of output we have also attached the mp3 version(externally converted) of the generated music in the zipped file.

CONCLUSION AND FURTHER WORKS:

To conclude, this project successfully generates notes of music from hashed labelled data. A memory network architecture has been employed to do the same.

Several implementations (wrapper class, sequential node for pytorch and sequential model for keras) were used and have been left in the notebook for referral. However, the keras LSTM model was finally used due to simplicity of execution.

An alternate model has been proposed (can be trained with minor changes in fitting process) which employs a bidirectional memory network for future dependencies (indeed, future notes may have an impact on the present note). This model is also, in a sense, “wider” as the dense/ linear layer is of a higher output dimension spanning a larger portion of features. However, using the mellow LSTM model, decent accuracy was reached.

Some scope of improvement would be the modularization of the data loader, i.e. wrapping the composer selection in a class such that it can be chosen by the user as a cmd line argument. Further, the temporal spacing of notes and the overlay of chords is neglected in this project, mainly due to computational limitations. Factoring those in would provide even better results. Finally, the hyperparameter tuning could have been more refined, but that is somewhat heuristic in nature given the characteristics of the data. This project can easily be extended into other instruments/ combinations of instruments as well.

STEPS TO EXECUTE THE CODE:

We have made our project in Google Colab keeping in mind that it makes the code more segmented, readable and easy to execute. All one has to do is to upload the following dataset from the given link to their drive: (https://drive.google.com/drive/folders/1agXGyTBNHXGacOOpz6aEd2N-4ZmZ3cqR?usp=sharing).
Change the filepath appropriately and execute it for a specified composer (in this case we have executed it for Beethoven) and run all the sections in proper sequence. For the prediction section you can vary the length of the generated music by varying the parameter 'note_length'. The generated music is saved, along with the data set, in midi format as well.
One may also execute the file in other workspaces other than Google Colab which can execute .ipynb files like Kaggle Workspace.

INDIVIDUAL CONTRIBUTIONS:

From the ideation to presentation the whole project has witnessed more or less equal contributions from all 4 members. For all sections of the project from data set retrieval, analysis and pre-processing to modelling and result analysis and even the report, the work was equally divided among us and all of us have made notable contributions in each section.