******

***An assignment report on***

**“Music generation using Bidirectional Recurrent Neural Nets and comparison with Markov Chains”**

**Topics in Deep Learning**

**Bachelor of Technology**

**Computer Science & Engineering**

***Submitted by :***

***01FB16ECS008: Abhijith Venugopal***

***01FB16ECS319 : S.S.Shreya***

***01FB16ECS340 : Sandya Sivakumar***

**August – December 2019**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

FACULTY OF ENGINEERING

**PES UNIVERSITY**

**(Established under Karnataka Act No. 16 of 2013)**

**100ft Ring Road, Bengaluru – 560 085, Karnataka, India**

**Problem statement:**

Generating music as a sequence of notes using bidirectional recurrent neural networks with Long Short Term Memory (LSTM) cells and comparing with music generated on same dataset using Markov chains.

**Introduction:**

Deep Learning classifiers and generators have already been used in several applications, over a wide spectrum of domains. A domain in which the use of deep learning algorithms is becoming popular is in the generation of music for different applications and software. This involves training some kind of a generator model on a large set of “music” data and coaxing it to produce similar kind of music with some variation. This is an interesting application of generators, as music as a dataset is much more complicated than images or text for direct application of generation- with each track composing of different notes, pitches, timbre values and so on. Therefore, analysis of conversion of music into some format which can easily be understood by models is a very important step which can make or break our “music generator”.

Our project is focused on application of Bidirectional RNNs (with LSTM cells) and Markov chains for music generation. These models have been trained on MIDI files which contain the music data. The data has been pre-processed in different ways depending on the models and then used for training the models.

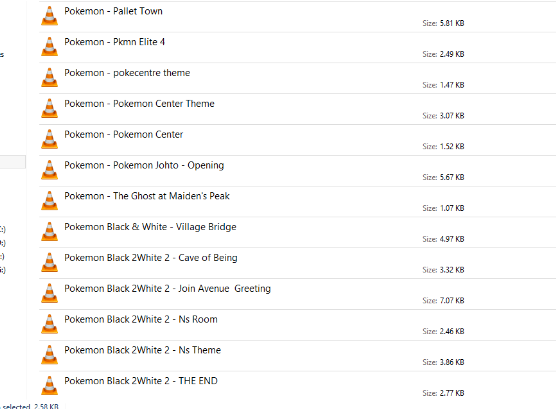
**Dataset Description:**

As stated earlier, music data is complex and difficult to represent as it composes of different attributes. Single instrumental music or track is much simpler to process, especially Piano instrumental tracks which have clear, distinct notes that can come from a wide range. Also using MIDI files helps in pre-processing as MIDI files contain metadata which can be used to convert the tones into some other format suitable to our model. Hence, we chose a corpus of piano music to train our models.

The dataset consists of a set of 307 piano tracks from Pokémon games, all in the form of MIDI files of length 1 minute approximately. The tracks neither have pauses nor vocals, which makes it easier for the model to train from it. The tracks used for training are stored in a folder named “Pokémon MIDIs”.

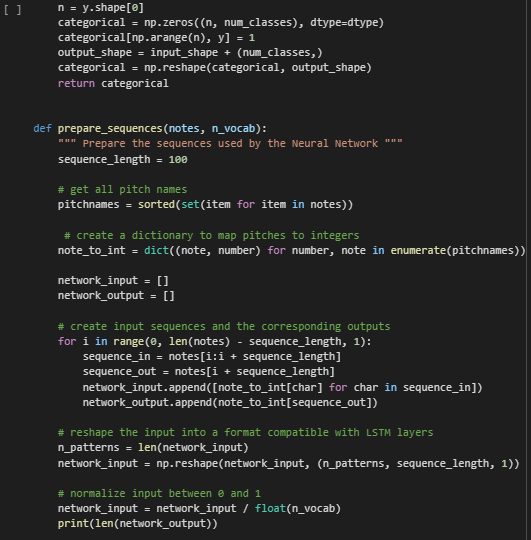
**Dataset Pre-processing:**

For pre-processing and extracting information from the music files, we used a software tool called Music21. Music21 is a set of tools which can be incorporated in python to do in-depth analysis and generation of musical tones and ragas. It builds on pre-existing frameworks and technologies such as Humdrum, MusicXML, MuseData, MIDI, and Lilypond, but Music21 uses an object-oriented skeleton that makes it easier to handle complex data. The pre-processing and representation of data vary with the model.



**Pre-processing Bidirectional LSTM:**





**Pre-processing Markov Chains:**

*def* \_parse(*self*, *verbose*=False):

"""

This function handles the reading of the midi and chunks the

notes into sequenced "chords", which are inserted into the

markov chain.

"""

midi = mido.MidiFile(self.filename)

self.ticks\_per\_beat = midi.ticks\_per\_beat

previous\_chunk = []

current\_chunk = []

for track in midi.tracks:

for message in track:

print(previous\_chunk)

print(current\_chunk)

print(message)

if verbose:

print(message)

if message.type == "set\_tempo":

self.tempo = message.tempo

elif message.type == "note\_on":

if message.time == 0:

current\_chunk.append(message.note)

else:

self.\_sequence(previous\_chunk,

current\_chunk,

message.time)

previous\_chunk = current\_chunk

current\_chunk = []

current\_chunk.append(message.note)

*def* \_sequence(*self*, *previous\_chunk*, *current\_chunk*, *duration*):

"""

Given the previous chunk and the current chunk of notes as well

as an averaged duration of the current notes, this function

permutes every combination of the previous notes to the current

notes and sticks them into the markov chain.

"""

for n1 in previous\_chunk:

for n2 in current\_chunk:

self.markov\_chain.add(

n1, n2, self.\_bucket\_duration(duration))

*def* \_bucket\_duration(*self*, *ticks*):

"""

This method takes a tick count and converts it to a time in

milliseconds, bucketing it to the nearest 250 milliseconds.

"""

try:

ms = ((ticks / self.ticks\_per\_beat) \* self.tempo) / 1000

return *int*(ms - (ms % 250) + 250)

except *TypeError*:

raise *TypeError*(

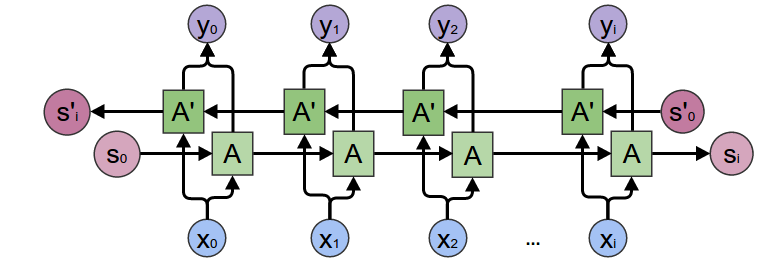
"Could not read a tempo and ticks\_per\_beat from midi")

**Model and Architecture:**

**Bidirectional LSTM:**

Bidirectional recurrent neural networks (BiRNN) are really just putting two independent RNNs together. The input sequence is fed in normal time order for one network, and in reverse time order for another. The outputs of the two networks are usually concatenated at each time step, though there are other options, e.g. summation.

This structure allows the networks to have both backward and forward information about the sequence at every time step.



Analysis of our MIDI files shows that the dataset can effectively be treated as text data, with each note interpreted as a “word”. This analogy makes it evident that we can effectively use LSTM cells on the processed music notes, along with temporal sequence to generate new “words”, a.k.a notes in our case. Therefore we pre-process the data and represent it as a numerical sequence of dimensions (num\_inputs, 100, 1). The idea while training is that given a sequence of ‘n’ notes in sequence, the model should predict the ‘n+1’th note efficiently. So we have a label set, in which we have used one hot encoding for the 497 classes of notes. Also since we are using Tensorflow backend, the input and output need to be converted as a set of tensors, each with the correct input and output dimensions. These are then used for training the network.

Network Architecture:

The Bidirectional LSTM consists of an input layer, 3 hidden layers (Bidirectional LSTM cells) and an output layer. Since we are implementing using Tensorflow, the input and output layers have not been explicitly defined and are implemented using tensors and matrix operations.

Input layer: Implemented using a sequence of tensors, each of shape (batch\_size,input\_dim). We have used input\_dim as 1 since each input is a single note and a batch size of 100.

Hidden layer 1: Layer consisting of 512 hidden units, where each unit consists of a forward and backward Bidirectional LSTM cells with forget-bias=0.1.

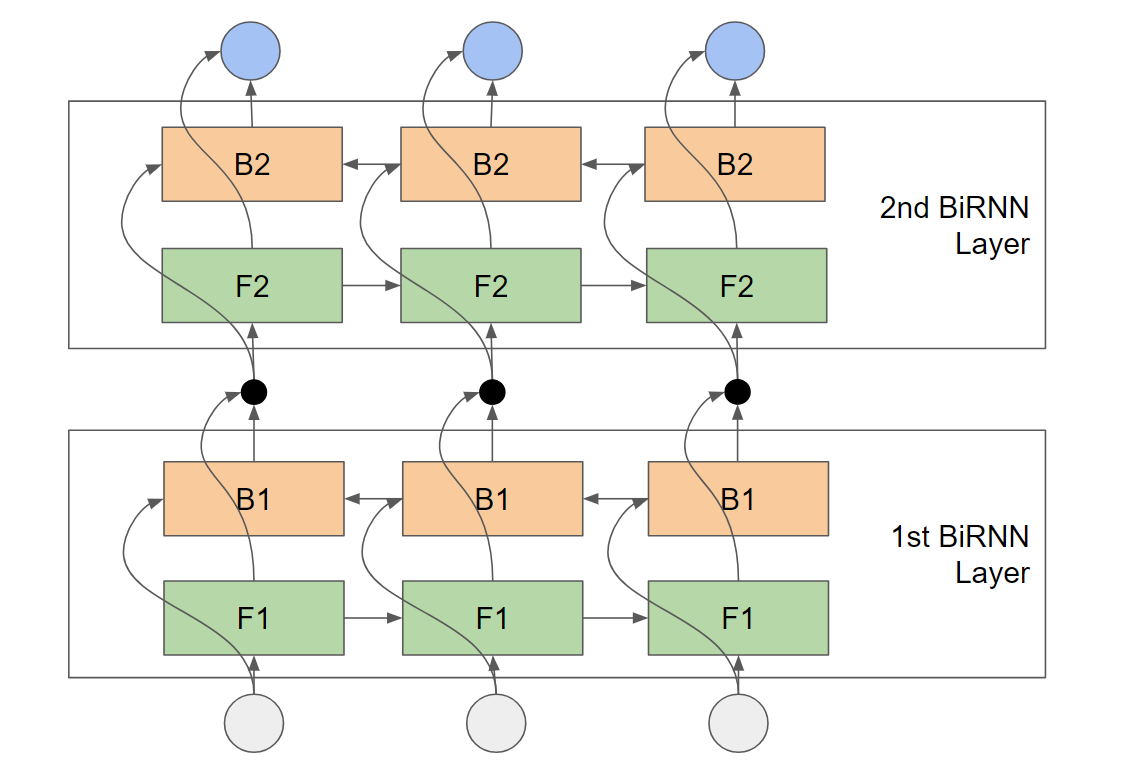
Hidden layer 2: Layer consisting of 512 hidden units, where each unit consists of a forward and backward Bidirectional LSTM cells with forget-bias=0.1. On this we have added dropout with drop probability of 0.2.

Hidden layer 3: Layer consisting of 512 hidden units, where each unit consists of a forward and backward Bidirectional LSTM cells with forget-bias=0.1.

Output layer: The output is a single output of dimensions (1,497) which is a one hot vector for the output note class. This is implemented by matrix multiplication of Weight matrix with the output of the previous layer and adding the bias matrix.

Stacked Bi-directional layers:

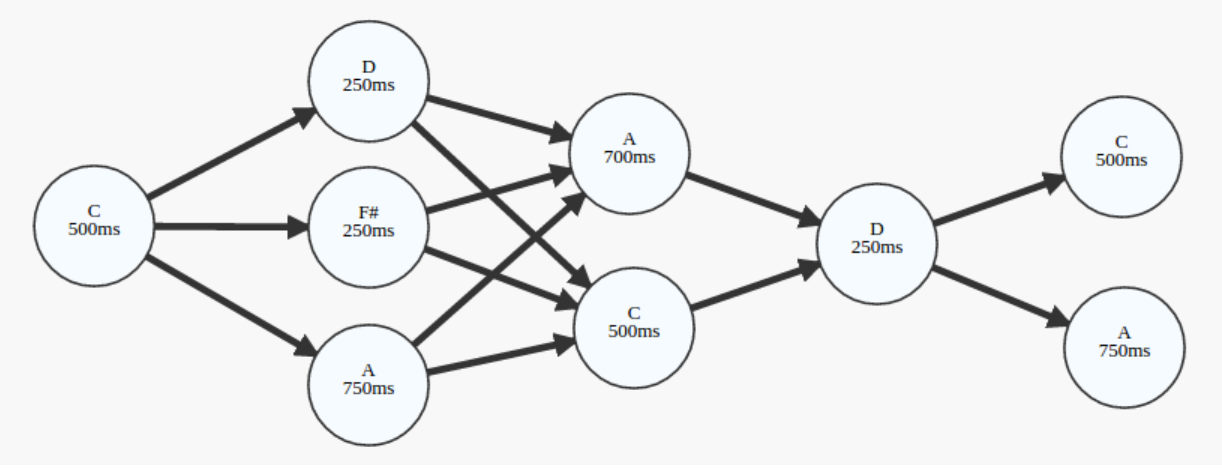
In our network, we have used a stacked architecture for the bidirectional hidden layers, where-in each layer’s output is passed as input for the next hidden layer. This provides a cumulative effect on the processing as compared to alternately stacked hidden layers. The initial input is given only to the first hidden layer and we use the output of the last hidden layer to compute final output.



**Markov Chain model:**

A Markov chain is a stochastic model describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event. This means that Markov chains define the problem in terms of a “state space”, where each state represents some particular configuration of the system and the transitions represent change from one state to another. Probability plays a major role in Markov chains as probabilities are associated with each transition which is used to determine the next state from a sequence of states.

Since we are trying to create a generator model using Markov chains, we need to represent our data as a finite set of discrete states, with transitions and probabilities associated with them. Since the tracks consist of a sequence of notes, we can group the notes which need to happen simultaneously into states, with the transitions denoting the shift from one set of notes to another. Therefore we need to group the states based on their times of occurrence and then assign probabilities with the transitions. We have used the metadata in the MIDI files to pre-process our notes into discrete states, with the link between states ‘i’ and ‘j’ indicating the number of times note ‘i’ transitions to note ‘j’



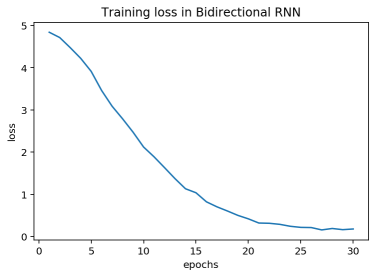
A screenshot of a computer

Description automatically generated

**Results and Analysis**

After training the Bidirectional model, we tested it by feeding a seed sequence of 100 notes. The model then generated the next 500 notes, which was converted into a track and stored as a MIDI file. The track generated was 2 min long and had similar note-sequence as the dataset.

|  |  |
| --- | --- |
| No. of epochs | Accuracy |
| 10 | 68% |
| 15 | 87% |
| 30 | 99% |



The Markov Model was also seeded with a single note chosen randomly. The output sequence of notes was converted into a MIDI file.