August 24, 2020

**Generated Classical Music Project**

**Version 1.0.0**

**Idea:**

This project idea is inspired by [another deep learning project](https://www.youtube.com/watch?v=SacogDL_4JU), which generated new classical music sympathy based on Bach's work. While Bach music is a little easier to work with because he has a mathematical pattern for each of his music pieces, I was interested in training a model that generates music based on Claude Debussy. Claude Debussy's work didn’t have any general pattern to his music, which makes this project a little more challenging to create.

**Objective:**

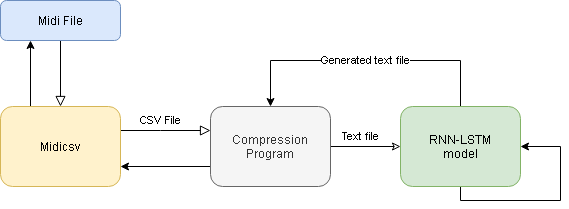
The main objective of this project is to get practice and understanding in using TensorFlow API and Recurrent Neural Networks Models. In addition, I would also learn about data handling and data manipulation, and how I can better improve and develop my neural networks model.

**Technology used:**

* Python 3.8
* Tensorflow 2.0
* midicsv 1.

**Workflow of the prediction model**

First thing thirst, is to download many midi files, and then convert them to CSV file using midicsv. Once we have CSV files, We then proceed to compress the files into a bunch of text using a customized program. The compressed text would then feed it to an LSTM-RNN model which would predict a new set of generated text. To get the new generated music we simply feed our new generated text file back to the compressor to be decompressed into CSV. From there, we can then proceed to pass the CSV file into midicsv (or more correctly csvmidi) to generate the new midi file.



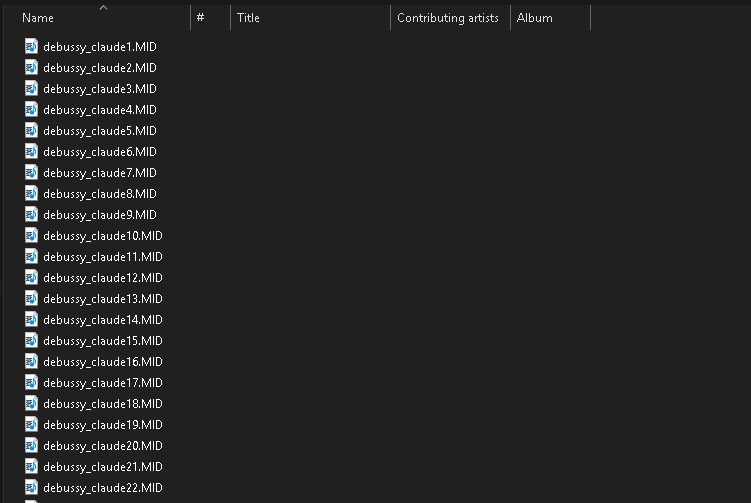
*Visual representation of my proposed workflow*

**Collecting the data:**

I needed to collect midi files which just played the keys of the piano. I was not interested in having any other interments played in the midi file but just plain piano keys. This is because it would be very difficult to extract the piano keys from the file when other interments are intertwined in the background.

I was able to find 43 midi files of Claude Debussy work from 3 websites, the website links can be found in \Generated Classical Music\README.txt

I make sure to rename all the files numeracy from 1-43 like so:



**Developing the Custom Compressor:**

There are a few main things that the compressor needs to preserve:

1. Each of the possible encountered piano keys must be mapped to a unique character

2. The text file must preserve the sequence of keys being played

3. The text file must preserve the clock time and hold the length of each key

The first part of the problem is relatively easy when coding in python. We can simply use *chr()* and *ord()* to translate back and forth between each keys. midicsv already transform the keys in the file into integers so it would be easy to compute. When creating the program, I noticed that when converting the integer-note to a character, I would need to start from the 33rd integer to avoid getting an unrecognized character.

The second part of the problem can be represented as a string of keys held together. For example, a C-major chord, which is C-E-G keys being held together, are translated in midicsv as 60-64-68, and can be represented in a text file as "jae" (without quotations).

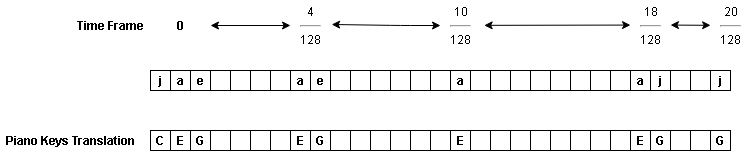
\*\*\* you can try this example out by the following python program \*\*\*

print(chr(60+33) + chr(64+33) + chr(68+33))

The last part of the problem is to keep track of the clock time and the pressed length of each key. This can be solved by using spaces, which each space represents the time length of a 1/128 of a note. Notes that are being played longer would have a wider space length.

For example: "jae ae a aj j"

'jae' (or C-major chord 1) is currently being played at the start of the mid file. 1/32 of note later, the C key was released, following the G key, and E key being held the longest. At the very end, G was pressed again while holding E, before E released 2/128 of note later while holding G.



midicsv can let you know the relationship of clock time and the length of a note at the start of the file, at the Header (which called division, and it represents the number of clock pulses per quarter note).

For example: 0, 0, Header, 0, 1, 430

430 represents the number of clock pulses per quarter note.

**Limitations:**

While the compressor sounds great in theory, it still has a huge limitation, and while it shrinks the file size greatly, it loses a lot of data from the original CSV file.

Possible data variable loss when compressing CSV file:

1. Cases were notes are not perfectly-being played at a specific time frame

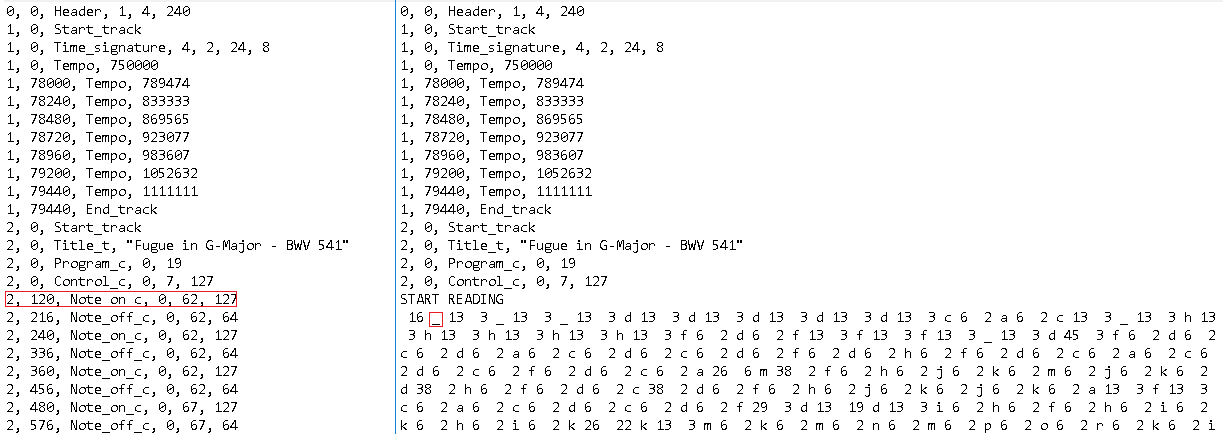
2. The velocity of the key being pressed is lower or higher than average

3. The pitch of the sound being played from the key

4. The pedal being pressed in different velocity and in specific time frames

Those are significant data information which is being lost and reduce the quality of the compressed midi file.

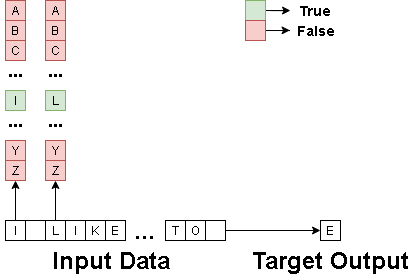
I was able to handle the preservation of the customization of each CSV file by copying line by line until I reached the first played a note. From there, I made a bookmark called "START READING", which is used by the decompression program to know which line the compressed data is being held onto.



*Side by side comparison of the CSV (left) and compressed file (right), the red box indicating the first note being seen*

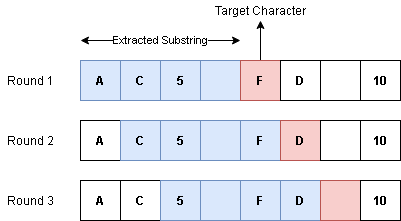
**Setting up the data for the model:**

Before I can think about designing the model, I need to understand what the input matrix would look like, and how the model can learn from the data. One way to do it is to break the input text into substrings, and get the next character and use it as the target prediction for the model. Each index of the substring would be attached to an array that represents all of the unique characters that can be seen in a text file. The array of the unique characters will be consist of True/False Values with only the character at the current index being True.



*Please note that the example above is generalized and the values of the data are vastly different*

Because we can't pass characters as an input for our LSTM cell, we simply create a 3D matrix based on the current line number, the index number of the array, and the mapped number of the unique character. True and False values can be represented as 1 and 0 respectively.

This process of data extraction of substrings would continue by shifting the subarray window index by until the end of the file like so:

*In the example above, we using a window size of 4, this would continue until the end of the file*

**Setting up the RNN model**

At first, I was not sure what type of model would be best for training on the data set. So I test one data set on a variety of models that I created.

The first model testing was to decide if the shape of the first layer would affect the loss. I ran over a substring length of 60, batch size of 128, and on 5 epochs. For optimization, I used Adam with a learning rate of 0.01.

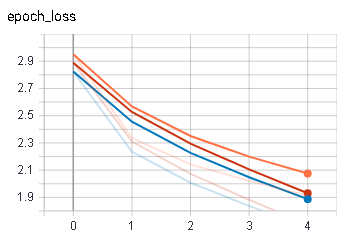
SETUP #1

Models Used:

1. LSTM(shape=64)->DROPOUT(0.2)->DENSE

2. LSTM(shape=128)->DROPOUT(0.2)->DENSE

3. LSTM(shape=256)->DROPOUT(0.2)->DENSE



(Blue = Model\_1 , Orange = Model\_2, Red = Model\_3)

Based on the graph, increasing the shape of the first LSRM cell would not help improve the model learning. However, when the LSTM shape was increased to 256, it did better than Model\_2 but not Model\_1. Therefore, I decided to decrease the LSTM shape to 32, and add other layers to see how it will react.

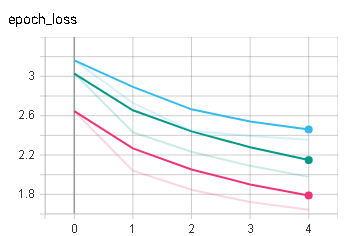
SETUP #2

Models Used:

1. LSTM(shape=32)->DROPOUT(0.2)-> LSTM(shape=32)->DROPOUT(0.3)->DENSE

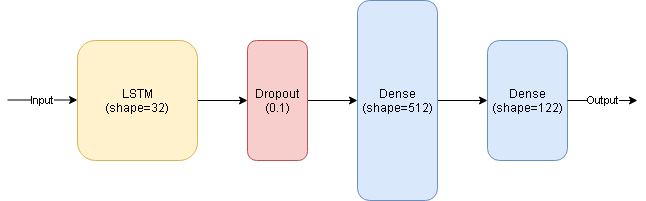
2. LSTM(shape=32)->DROPOUT(0.2)->DENSE(shape=32)->DROPOUT(0.3)->DENSE

3. LSTM(shape=32)->DROPOUT(0.2)->DENSE(shape=512)->DROPOUT(0.3)->DENSE



(Light Blue = Model\_1, Green = Model\_2, Pink = Model\_3)

Based on the above results I decided to use Model\_3 and tweak it even more to get even better results. This is the best model the reduced the loss so far:



**Training the model:**

The very first few tests I did on the model was to over fit it on one song. The reason behind this is to understand whenever the model can learn in general the style of a particular song. I also used over fitting to see which values I can fine-tune, such as the batch size, epochs, and sequence size, in order to speed up the learning process and decrease the loss.

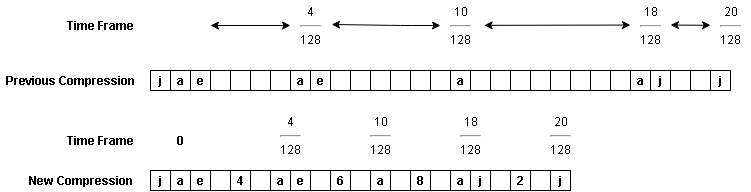
**Problems during training:**

During the period when I was training the model on the data, the model always failed to understand how to write proper music. The biggest issue was that the generated text file was 95% composed of spaces. The key reason behind it was because the data I was training on, every singular or a compound note that was played, was followed by an arbitrarily long length of spaces; which represent the length of time of the particular note being played. This problem was also pretty common for many of the files because Claude Debussy Is known to have created slow-paced music.

What does that mean when it comes to training the model? the probability of the model seeing space as the next predicted character is vastly huge compare to the rest of the unique characters. This means that the probability of the model choosing a space as the next character and being correct is also vastly huge.

solution:

I needed to reduce the probability of encountering space as well as creating a new representation for the length of time keys where being pressed or added. The solution I came up with is that each set of notes which being played will be followed by a space, an integer, and space. The integer would be used to represent how many 1/128 notes to hold before adding or removing a new note.

****

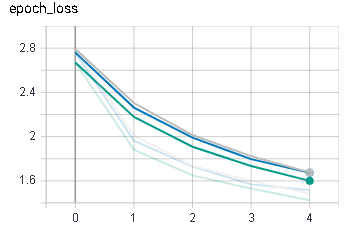
Problem with the solution:

there is one minor flaw by using an integer to represent the space between notes being played, and that is that some notes are being translated into integers. My hope is that the model will learn that an integer between two spaces or other integers meaning it’s a time frame, and an integer between characters is a note. Even so, there is a chance to encounter a situation where the note that being played is translated to just integers.

I did not want to risk that situation, so I added a unique character (which is not mapped to a piano key) before the integer timeframe to distinguish the difference between keys and clock pulses. I also used this unique character (0x9a) as a special case where notes are being played faster than 1/128 of a note.

**Over-fitting-1 Test over text size:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Color** | **Optimizer** | **Batch Size** | **Epochs Size** | **Input Size** |
| Grey | Adam(lr=0.01) | 128 | 5 | 128 |
| Green | Adam(lr=0.01) | 128 | 5 | 64 |
| Orange | Adam(lr=0.01) | 128 | 5 | 16 |



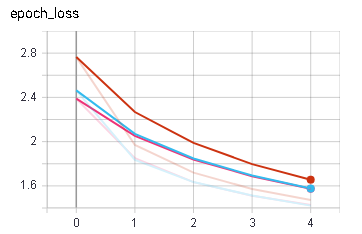
(Grey = size\_128 , Green = size\_64, Orange = size\_16)

Observation:

While there is not much difference when changing the substring size the model analyzes, it seems like that the model is learning the best when it looks at a substring size of 64. A smaller substring (or looking at fewer notes) compare to a larger substring (looking at more notes) doesn’t have much of a difference.

**Over-fitting-2 Test over batch size:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Color** | **Optimizer** | **Batch Size** | **Epochs Size** | **Input Size** |
| Brown | Adam(lr=0.01) | 128 | 5 | 64 |
| Light Blue | Adam(lr=0.01) | 64 | 5 | 64 |
| Pink | Adam(lr=0.01) | 16 | 5 | 64 |

****

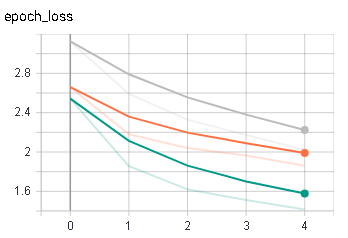
(Brown = batch\_128 , Light Blue = batch\_64, Pink = batch\_16)

Observation:

A smaller batch size makes the model learning much better compared to a larger one. While a batch size of 16 did somewhat better than a batch size of 64, I decided to stick with a batch size of 64 anyway for speeding up the learning time.

**Over-fitting-3 Test over leaning rate:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Color** | **Optimizer** | **Batch Size** | **Epochs Size** | **Input Size** |
| Green | Adam(lr=0.01) | 64 | 5 | 64 |
| Grey | Adam(lr=0.001) | 64 | 5 | 64 |
| Orange | Adam(lr=0.05) | 64 | 5 | 64 |



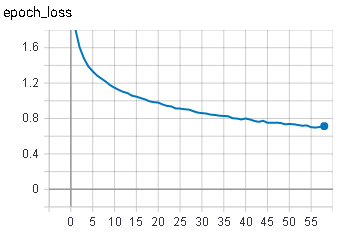
(Green = LR\_0.01 , Grey = LR\_0.001, Orange = LR\_0.05)

Observation:

Based on the results, it is suggested that the data is very noisy, I can assume that learning the data would be very hard, but in order to truly see how the model handles it, I would need to increase the epochs size to a very large number.

**Over-fitting-final:**

Here I would check the optimal epochs size



Observation:

Base on the results, I decided that after 5-10 epochs the training is not improving too much and simply over fitting.

Its look like the model is having a hard time predicting the compressed music data as it could not do better than a loss of 0.8

There are several reasons as to why that is the case. For once, it could be that the data is too noisy to be understood. It could be that the way we compressed the data into a text file is not a good data interpretation for the model to learn from. It could also be possible that our model itself is not built in a way that could understand the general format of the text pattern.

In either way, we could assume that the final results when testing the model on the entire data set , it would have a very hard time reducing its loss. Because of Claude Debussy's work not having a general style, could be a reason why the model is having a hard time reducing the loss. Therefore on the final training test, I would to have to test the model on compressed midi files based on Claude Debussy's works, as well as on Bach's work. If the model would not be able to learn from Claude Debussy's work but it would be able to learn from Bach's work, we can conclude that the problem is simply Claude Debussy dataset. If that is the case, I might need to develop a different model altogether in order to try to learn Claude Debussy's music style.

You can find the over fitted model at ./Model/Model\_Overfit\_x64x64x80.h5

**Final Results (Claude Debussy):**

The first training was done over Claude Debussy's works over 5 epochs. With no surprise, as it turns out the model could not predict properly the pattern of the composer. On average, the model loss ended between 1.2 to 1.0. When generating new music using the trained model, it seems to be stuck between a few different notes. However, when increasing the diversity of the generated text, it was able to generate some kind of music with hardly any patterns to it.

The model can be found in ./Models/Model\_1\_x64x64x5.h5

The Generated midi could be found in ./Generated/Midis/ generated\_Model\_1 (Debussy).MID

**Final Results (Bach):**

After the first test results, I downloaded a bunch of midi files based on Bach's works. I then proceeded to executed the program again for 1 epoch as there where around 250 files to work with. The results were much more impressive. The loss was highly dispersed around 0.9-0.4 suggesting that Bach's work is much easier to predict then Claude Debussy. However, based on these results, I can conclude that either my model was not well suited for learning on the data, or my data itself was too noisy to understand. Hopefully, for my next version of this project, I would be able to develop an even better model then I did now.

The trained model based on Bach can be found in ./Models/Model\_2\_x64x64x1.h5

The generated music based on Bach training was much more appealing then Claude Debussy. Just like the first test, I had to use high diversity in order to have a unique type of music. You can find the files in:

Generated/Midis/ generated\_Model\_2 (Bach) 01.MID

Generated/Midis/ generated\_Model\_2 (Bach) 02.MID

**References**

https://towardsdatascience.com/generate-piano-instrumental-music-by-using-deep-learning-80ac35cdbd2e

https://towardsdatascience.com/generating-text-using-a-recurrent-neural-network-1c3bfee27a5e

https://www.section.io/engineering-education/text-generation-nn/