Easy Tunes: Simplified Music Generation Using Transformers

Michael Ingram

1 Introduction

From amplifying emotions in a movie scene, to improving focus during a study session, to helping people relax during meditation, music is an important part of the human experience. While music can have many powerful applications, many don't have the knowledge or abilities to create music themselves. The goal of this project is to use consumer grade hardware to generate simple songs for people who lack musical talent, so they can use music for marketing, game development, or their personal enjoyment.

Others have looked into music generation before, with decent success. OpenAI has two projects dedicated to this: MuseNet which generates sequences of notes [1], and Jukebox which generates raw audio [2]. While these models sound great, they take tremendous computation power. For example, Jukebox requires 3 hours on an NVIDIA Tesla V100 to generate only 20 seconds of music [3]. To make the task of music generation more accessible, this model will reduce the features available and use a simpler architecture.

With the success of the transformer architecture in the news lately, it is a prime candidate for this task. The attention mechanism can help in identifying the repeating patterns in music. The model will take the first few notes as an input, which can be randomly generated based on a probability distribution in the final deployment. It will use these notes to predict the next notes from the training set until it has reached the end of a song. The training set will be 2,946 MIDI files downloaded from Ambrose Piano Tabs, a teaching website with songs specifically written for piano and keyboard [4]. Finally, the model can be fine-tuned using genre or artist specific training sets to generate music in the desired style.

To measure the success of the model, train and test set loss can be initial indicators. These will provide a rough idea of how the training is going, but it won't describe the quality of the music being output. To solve this, the model will be evaluated on pre-defined prompts. The predictions will be subjectively evaluated by looking for musical qualities, such as repeating sequences versus random notes, staying within key, and chord formations. With more resources, this could be a survey to get quantitative results, but in this case, it will just use the team's evaluation.

Our hypothesis is that decent music can be generated on consumer grade hardware using the transformer architecture, by simplifying the model and limiting the features compared to other solutions.

2 Related Work

As mentioned before, others have had decent success in music generation. Google's Magenta is one of the first applications of the transformer architecture being used for music generation [5]. The paper provides results showing the transformer performs better than previous models, like LSTMs or CNNs. However, the project is no longer maintained and the codebase has deprecated.

OpenAI's MuseNet is another similar project which mentions the use of transformers for music generation [1]. This project has no official paper or codebase, however the blogpost offers ideas for training on MIDI files, such as representing encoding the dataset with tokens which combine the pitch, volume, instrument, and duration. For example "piano:v72:G1" would use the piano track to play a G1 note at 72% volume and "wait:5" would indicate a rest.

Another similar project is Museformer, by Mircosoft [6]. Museformer uses fine and coarse grain attention to capture basic music theory, as well as repeating parts of the music, like choruses and verses. It is limited to 4/4 time signature and uses 6 standard instruments. The computation is expensive, requiring an NVIDIA Tesla V100 32GB graphics card.

As shown above, all of these models either require intense compute power or do not have a public codebase. This work will provide a public code base and aims to generate music on consumer grade hardware.

A similarity in these models is that they use many tokens to give the music more flavor, but provide diminishing returns. For example, by reducing the number of instruments, the vocabulary size will decrease and the model won't need to spend tokens providing instruments. Likewise, while volume is important to the feel of the music, it doesn't contribute as much as the pitch and timings.

3 Background

The training set for this model uses MIDI files. MIDI files are binary files that contain a series of instructions for the computer in order to recreate a song. Each note in a MIDI file has the following properties: note type (on/off), channel, note or pitch, velocity or volume, and time since the last event.

```
Message('note_on', channel=1, note=48, velocity=100, time=0),
Message('note_on', channel=1, note=52, velocity=100, time=0),
Message('note_off', channel=1, note=48, velocity=0, time=48),
Message('note_off', channel=1, note=52, velocity=0, time=0),
```

Figure 1 – Example MIDI note messages. The first two notes turn on immediately. The third message waits 48 time steps and then turns off note 48, at the same time as message four turns off note 52.

The note_on and note_off pose a significant problem because the file will become corrupt if the model forgets to turn the notes off. To solve this issue, each track in the MIDI file was stepped through to determine when notes were starting and stopping, as shown in Figure 2. This converted the on/off input into a duration and grouped the notes by the time they occur, instead of grouping by track. To simplify things even more, the velocity was discarded and only set to 100 or 0 depending on if the note was on or off and all notes are played on a single channel.

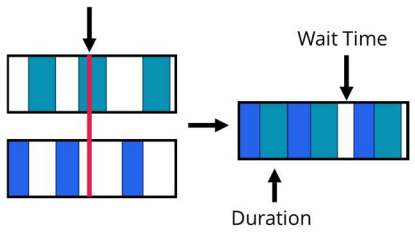

Figure 2 – Time Stepping through each track to determine when notes start and stop, and converting this into a duration and a time since the last event.

4 Method

The data from the MIDI files was extracted using the Mido package in Python [7]. The pitch, wait time, and calculated duration were saved as a numpy array to represent a note. Each note was then concatenated to form a song as a numpy array. Finally, these songs were concatenated to create the dataset. Songs were padded or trimmed to be 600 notes long, so that the final dataset had a shape of 2,946 songs by 600 notes by 3 note parameters.

There were 87 unique pitches, 124 unique waiting times, and 94 unique durations of notes, as shown in Figure *3*. To create a unique token for each combination would require over a million tokens. Instead, each note parameter was tokenized and embedded separately before being added together. This will significantly reduce the total amount of tokens.

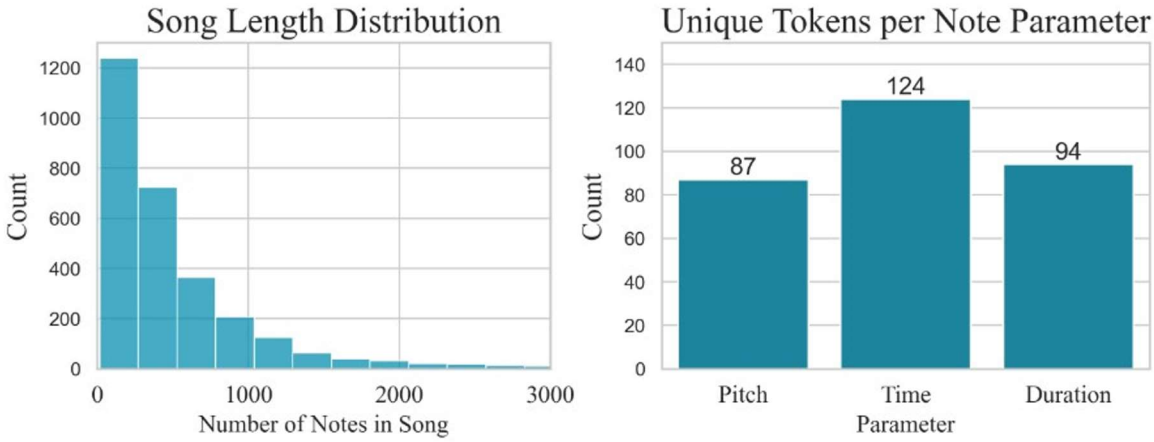

Figure 3 – Distributions of song length (left) and the number of unique tokens per note parameter (right) in the dataset.

The model will take a batch of input sequences and separate it into sequences for each of the note parameters. It will perform the embeddings separately, with a dimension of 1,024, and sum them together before being ran through a positional encoder. This feeds into a transformer encoder with 4,096 parameters, 4 encoder blocks, and 8 attention heads. The output of this feeds into a separate decoder for each of the note parameters to predict the next note in the sequence. This architecture is shown below in Figure 4.

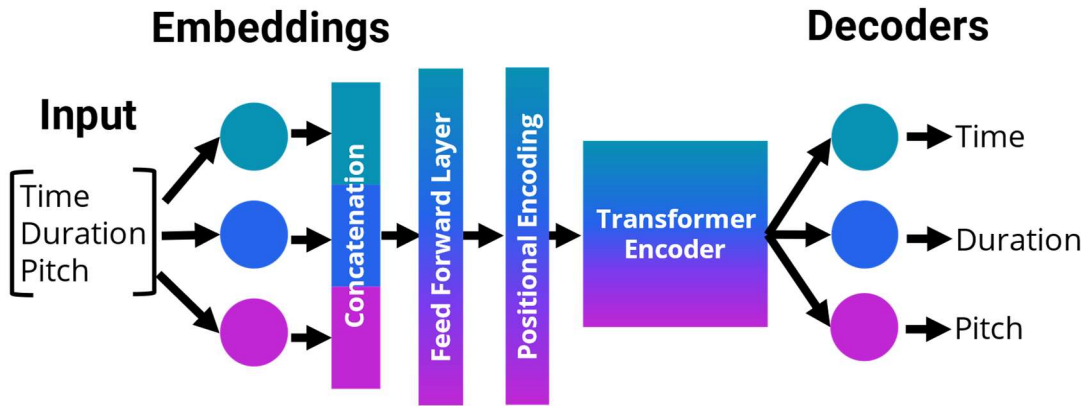

Figure 4 – A representation of the model, which takes an input sequence of time, duration, and pitch and aims to predict the next note in the sequence.

In addition to the separate embeddings and decoders, each note parameter also receives its own cross entropy loss function which is summed into a combined loss, shown in Equation 1.

$$Combined\ Loss = Time\ Loss + Duration\ Loss + Pitch\ Loss \quad \text{Equation 1}$$

5 Experiments

The model was trained with a batch size of 1,024 using an Adam optimizer and step learning rate decay. The validation loss was always similar to the training loss, indicating the model was not overfitting, however training performance using multiple different optimizer and scheduler settings did not lower the total loss much. This may be due to the diverse dataset and limited number of parameters.

After training the model, the first 10 notes of 10 songs in the test set were used to test the model's time performance across 3 different consumer grade devices: an Nvidia 2070 RTX Super graphics card, a Ryzen 9 3900X Desktop CPU, and an Intel 1185G7 mobile CPU. Figure 5 shows that when using consumer grade CPUs, the devices take around a minute and 20 seconds to generate a song that is around 3 minutes. The model sees 10 times faster performance when ran on the graphics card.

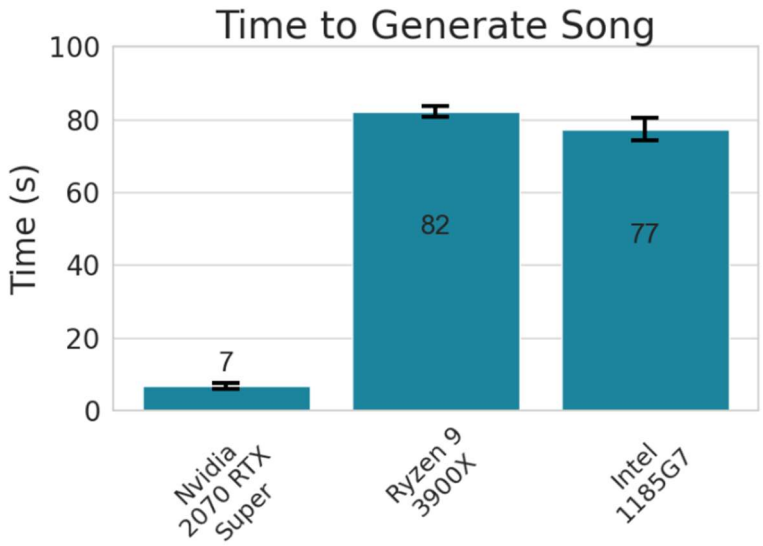

Figure 5 – Time to generate a song using the model. Songs were approximately 3 minutes in duration and 600 notes.

While the model just needs one note to start, which could be randomly selected from a distribution, the MIDI file needs certain metadata qualities like beats per tick. For each of the songs in the test set, we copied the metadata from the original track to simplify the testing process. However, users could select their own parameters such as key signature and tempo, or one could be picked at random for the deployed model.

The songs generated by the model capture some musical ideas, like staying within key, keeping consistent rhythms, and even playing some chords. However, it lacks any sense of musical motifs, repeating patterns, or structure such as a chorus and verse. Some examples of songs generated by the model can be seen in Figure 6.

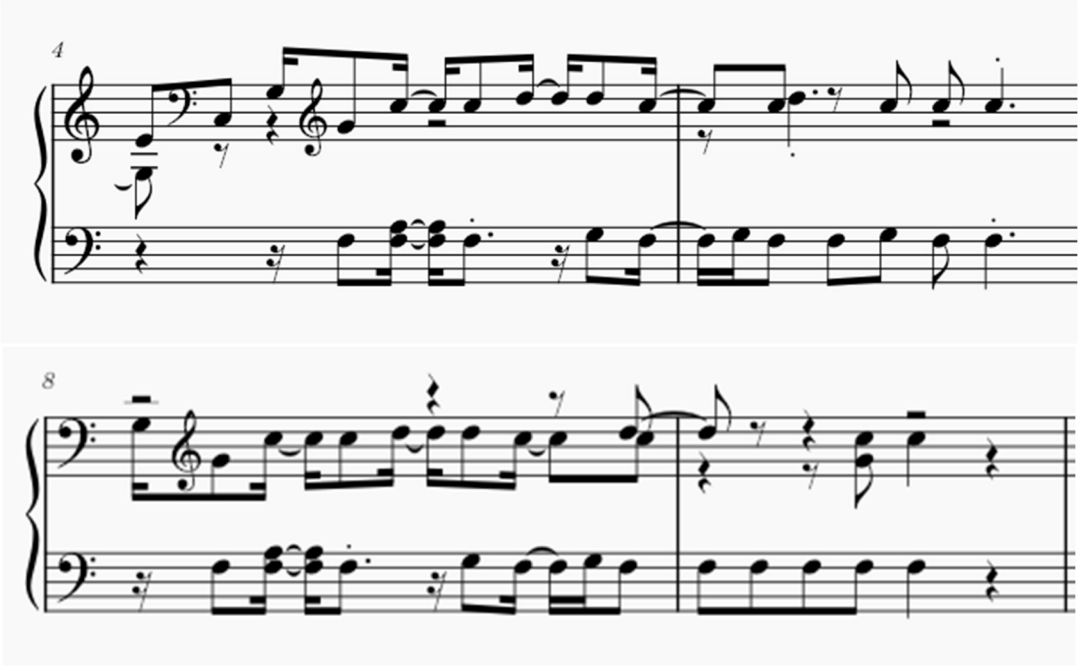

Figure 6 – Examples of music generated by the model

6 Conclusion

The model was able to generate a song on a laptop in around a minute and 20 seconds. It succeeded in generating something musical without needing a high-end graphics cards and long inference times. However, the music is simpler than models like MuseNet and Museformer. While it stays within key and has reasonable timings, it lacks different instruments and volumes, as well as other musical qualities like repeating motifs.

A possible solution would be to use a more efficient architecture. The basic transformer used has a performance of $O(n)$ and other architectures like the Spare Transformer and Mamba operate in $O(n\sqrt{n})$ and $O(n)$, respectively [8] [9]. If the performance increased, more parameters could be used and the model could learn more patterns.

Additionally, with faster performance there could be more room for features that were removed, such as the track metadata and volumes. These qualities will help the music seem more human and might help the model by providing more context.

Finally, the dataset contains very diverse music subsets, ranging from Bach to Coldplay. Fine-tuning on a specific genre or artist may allow the model to use its limited parameters more effectively.

7 References

[1] C. Payne, "MuseNet," OpenAI, 25 April 2019. [Online]. Available: openai.com/blog/musenet.

[2] P. Dhariwal, H. Jun, C. M. Payne, J. W. Kim, A. Radford and I. Sutskever, "Jukebox: A Generative Model for Music," OpenAI, 2020.

[3] P. Dhariwal, J. Heewoo, C. M. Payne, J. W. Kim, A. Radford and I. Sutskever, "Jukebox Github Repository," OpenAI, 13 November 2020. [Online]. Available: https://github.com/openai/jukebox/.

[4] "Ambrose Piano Tabs," Ambrose, 2012. [Online]. Available: http://ambrosepianotabs.com/. [Accessed 16 December 2023].

[5] C.-Z. A. Huang, A. Vaswani, J. Uszkoreit, N. Shazeer, C. Hawthorne, M. A. Dai, D. M. Hoffman and D. Eck, *Music Transformer: Generating Music with Long-Term Structure,* Google, 2018.

[6] B. Yu, P. Lu, R. Wang, W. Hu, X. Tan, W. Ye, S. Zhang, T. Qin and T.-Y. Liu, *Museformer: Transformer with Fine- and Coarse-Grained Attention for Music Generation,* Microsoft, 2022.

[7] R. B. Ole Martin Bjorndalen, "Mido," 15 December 2023. [Online]. Available: https://pypi.org/project/mido/.

[8] R. Child, S. Gray, A. Radford and I. Sutskever, "Generating Long Sequences with Sparse Transformers," 23 April 2019. [Online]. Available: https://arxiv.org/pdf/1904.10509v1.pdf.

[9] A. Gu and T. Dao, "Mamba: Linear-Time Sequence Modeling with Selective State Spaces," 1 December 2023. [Online]. Available: https://arxiv.org/ftp/arxiv/papers/2312/2312.00752.pdf.