Timothy Clay
CS4120
Final Project Report
April 13th, 2025

Introduction

For this project, I used three different approaches to language modeling to create computer-generated melodies by generating patterns of notes and rhythms, as opposed to words and characters. The three models I chose to use – *n*-gram, RNN, and GPT-2 – varied significantly in complexity, allowing me to see how the quality of the generation fluctuated with the type of model. I trained each model on MIDI transcriptions of contemporary pop songs, and evaluated each by their ability to 1) successfully produce melodies representative of the training data, 2) produce melodies that are internally consistent in terms of key and time signatures, and 3) generate interesting and creative melodies.

I found this project interesting and worthwhile because, despite it being a simple subversion of what we would normally expect from them, adapting these models for notes and rhythms as opposed to words is an interesting litmus test for them. There are an infinite amount of factors that go into a "good" melody. For instance, on a basic level, the melody should be internally consistent with itself and each note should flow into the next. As you get more complex, you can introduce motifs or patterns that make the melody more compelling. Generally speaking, the more thought that goes into the creation of a melody should affect the quality of the melody; this project seeks to determine how much "thought" these models can devote to the task.

Data

To train each of my three models, I extracted tens of thousands of melodies from the Lakh MIDI dataset[1], a collection of over 100,000 MIDI files published by Colin Raffel, an associate professor at the University of Toronto. These MIDI files are machine-transcribed versions of each song in the Million Song Dataset[2], which is its own thorough collection of contemporary pop music songs. The full Lakh MIDI dataset is 176,581 MIDI files, though only 45,129 of them are directly matched with songs from the Million Song Dataset. For this project, I only worked with the smaller subset of files, since I found that they were consistently of higher quality than the rest.

While the MIDI files contain full transcriptions of every part/instrument in each song, I only wanted to highlight each song's melody; I didn't want the models to learn the patterns of a guitar part, for instance, which would be distinctively different from the melody of a song. To account for this, I programmatically crawled through each file in the data, checked the encoded name of each part, and extracted only those that included the name "melody." This resulted in 12,261 tracks, with an average of just over 327 notes each, giving me a dataset of just over 4 million notes with which to train my models.

To preprocess the data, I iterated through each track note-by-note and extracted the pitch (which was encoded as an integer) and length into a list. I decided to alternate between notes and rhythms in said list because I figured that any somewhat competent model would be able to detect this alternating pattern. The final list for each melody, therefore, was in the form `[60, "whole", 62, "eighth", …]`. From there, each melody was represented by its own list inside of one larger container list. When it came time to actually feed the data into each of the models, they all required the data to be prepared in a slightly different manner, and I had to tweak this approach slightly throughout the project. I will explain these nuances in more detail in the next section, but this overview properly characterizes my default preprocessing strategy that I used throughout the project.

Models

To create and train each of my models, I defined three custom classes. The implementation of each is obviously different, but they all at least contain one function to fit the model using the provided data and another to generate a melody of a certain length using each model's respective generative functionality. The majority of the data processing is done outside of these classes, using utility functions I defined in **melody_utils.py**.

N-Gram

The first and most basic model type that I implemented was an *n*-gram model. This model works by predicting each token in a sequence using the previous *n* - 1 tokens as context. For each potential new token, it calculates the probability that the new token belongs with the existing *n* - 1 tokens using frequency counts of how often each combination of tokens has appeared in the training data[3]. For this specific task, the tokens the model is predicting are note pitches and note lengths.

I started by training this model using the data processing steps outlined previously, but quickly ran into issues where the model would jump randomly between octaves, causing me to adjust my data processing approach for just this model. Rather than keeping the note pitches as integer values, I instead translated each to its letter representation (e.g. 60 turned into "C"). Doing this meant that the generated melody would stay within the same octave range, which significantly improved the observed results.

Training the *n*-gram model was relatively straightforward. Using the tokenized data, I simply iterated through each melody and calculated the *n*-grams and sub-grams (*n*-grams of size *n* - 1), which I stored in Counter objects. When it came time to predict a note, I simply called on these counts to determine the probability of each potential new token. For the final version of this model, I settled on an *n* value of 5, which allowed the model to at all times consider the last two notes in the sequence. I experimented with several different values, but found that considering any fewer notes would result in uninformed predictions by the model, but considering any more

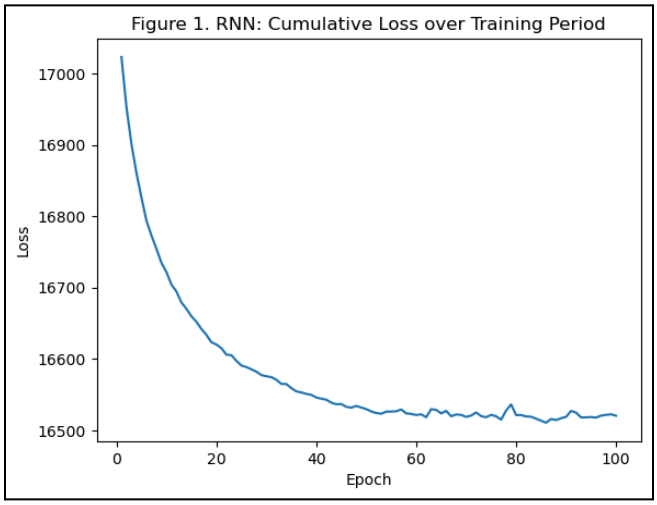
would result in too few similar instances in the training data, resulting in random tokens being generated.

RNN

The next model that I tried was a recurrent neural network (RNN), which is comparatively much more involved. The RNN is a sub-class of neural networks, which work by passing data through several processing layers to output one result. Recurrent neural networks build on this architecture by maintaining a hidden state that, for each token in a sequence, captures information from the previous tokens. At each time step, the model will update this hidden state, allowing it to retain context across the sequence[4]. The only difference between my model and a typical RNN model is that my model generated note pitches and note lengths rather than words or characters.

As was the case with the *n*-gram model, I started training this model using the aforementioned preprocessing strategy for the data in which the note pitches are represented by integers. Unlike with the *n*-gram model, however, I found that the RNN quickly picked up on the patterns of the training data as-is, and was much more natural in its predictions. For this reason, I left the data preprocessing strategy alone when training the RNN model. For the model's architecture, I experimented with different parameters to use throughout the process, and ultimately landed on using a 128-dimension embedding layer and 128 hidden units.

As for the training itself, I used the same approach to training as I had previously for feed-forward neural networks (repeatedly generating the loss, back-propagating the loss, and updating the gradients), but with an RNN architecture instead of a feed-forward one. I trained this model for 100 total epochs, which, on my local machine (Mac M2), took approximately 45 minutes. As seen in Figure 1, by the end of the training period, the total training loss appears to have converged, so I am skeptical that any further training would have significantly improved the model's performance.



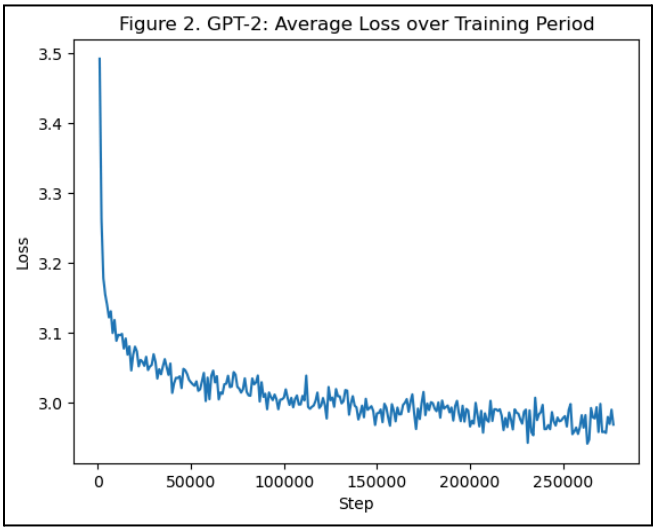
GPT-2

The third and final model that I used for this project was a pre-trained transformer model. The specific model that I fine-tuned was GPT-2, which is a 2019 pretrained model released by OpenAI built around transformers. This model works by reading an entire input all at once

(unlike RNNs, which read one token at a time) and uses attention mechanisms to figure out which tokens in the input are most important for predicting the next token. GPT-2 is originally trained on a large amount of text, but can be fine-tuned using a custom dataset, which is what I sought to do for this project[5].

To make said custom dataset, I again started training this model using integer representations of notes and alternating between notes and rhythms in the training data. Of the three models I tested, this model performed the worst when provided the data in that format. Not only did the model predict random notes, but it was at first not easily able to detect the alternating pattern between notes and rhythms, so the generated outputs were entirely unintelligible. To address this problem, I changed the preprocessing approach to both use letter representations of notes and glue together the pitch name and the length name (e.g. "C-half"). Preprocessing the training data in this way (and training the model for longer) resulted in much better results than previously found.

To train this model, the only heavy lifting I had to do was create a custom dataset that would pair windows of 10 input notes with the 10 corresponding "label" notes that would come immediately after them (for instance, if the input notes were the first 10 notes of a melody, the label notes would be notes 2-11). After that, all I had to do was use the TrainingArguments and Trainer classes from the transformers library, which allowed me to specify exactly how I wanted the model to be trained. For the final model version, I trained it for one full epoch. On my local machine (Mac M2), this took around 2 hours start-to-finish. While I would have liked to have trained the model for more than a single epoch, with the amount of training data I was using, that was not computationally viable. Still, I found that the average loss appeared close to converging by the end of the training period, as seen in Figure 2, so I don't think that the performance of the model suffered significantly by this decision.



Results

To evaluate the performance of each of these models and determine which architecture is the best at melody generation, I compared the generations of each using both objective and subjective criteria.

To objectively evaluate the performance of each model, I generated 100 different melodies of 100 notes each from the three models to see if the percentage of the types of notes and rhythms are consistent with those seen in the training data. For the sake of space efficiency in this report, the results of this analysis can be found in the **melody_generation.ipynb** notebook under the *Results* section, but, at a high level, the melodies outputted by each model were very consistent with the seen training data. For rhythms, eighth and sixteenth notes dominated the training data, but they were also the most prevalent in the generations. For the notes themselves, the most common pitches in the training data were A, D, and E, which was similarly reflected in the language models. These trends remained consistent across all three model types.

For this reason, from an objective standpoint, I believe that all the models were successfully able to produce a melody that is consistent with the data they were trained on. However, so much more goes into making a successful melody than the proportion of notes and rhythms, which is why I also evaluated the models subjectively. To do this, I created several melodies from each model and showed them to a few friends, asking them to rank them by two factors: cohesiveness and engagement. For cohesiveness, I was looking for how much the melody stays within a consistent key signature and a consistent time signature. Do any of the notes sound out of place within the context of the rest of the melody? For engagement, on the other hand, I was interested in how creative the melody is. Are the notes and rhythms varied in some sort of interesting way?

Using these parameters, I found that public opinion suggested that the RNN and GPT-2 models both significantly outperformed the *n*-gram model. The *n*-gram-generated melodies mostly sounded like a random collection of notes, and they all ranked quite poorly in terms of both cohesiveness and engagement. For the RNN and GPT-2 models, however, the feedback was much more positive. Of the three models, those who I talked to agreed that the RNN model is the most engaging and creative, while the GPT-2 model was the most cohesive and accurate. Based on these results, the "best" model out of those two simply comes down to personal preference. For me, I found the RNN melodies the most impressive, especially since I didn't have to handicap the model by limiting it to a single octave. For this reason, I concluded that the RNN is the best of the three.

Conclusions / Future Work

Overall, I was pleasantly surprised by the outcome of this project. While they would not be considered "good" if written by a human composer, the computer-generated melodies were still somewhat impressive, and it showed that the model architectures that I experimented with are capable of picking up tons of information from the training data. Going into this project, I expected the outputs to be entirely nonsensical – and some of them still were – but the results of this project thoroughly surpassed my expectations.

If I were to continue this work, there are a few improvements that I might incorporate into any next steps. First, I might want to refine the source data to more accurately match the goals of this

project. While my strategy of extracting all tracks with “melody” somewhere in the name was likely a good enough proxy for finding each song’s respective melody, I’m sure that I missed some titled “lead” or “vocals.” Refining this approach would give the models more data to learn from, which could never hurt. Another improvement I might try would be to approach the data processing differently to help the models better understand the concepts of key and time signatures. This might include adding tags in the data to represent the key or the time signature, which the models could then learn off of to improve consistency. While I could also always experiment with other model types, the promise I saw from these models thus far indicates to me that, given the refinements I mentioned, they may be able to significantly improve their generations, and perhaps even produce something comparable to a human’s own work.

Works Cited

1. Colin Raffel. "Learning-Based Methods for Comparing Sequences, with Applications to Audio-to-MIDI Alignment and Matching". PhD Thesis, 2016, https://colinraffel.com/projects/lmd/#get

2. Thierry Bertin-Mahieux, Daniel P. W. Ellis, Brian Whitman, and Paul Lamere. "The Million Song Dataset". In Proceedings of the 12th International Society for Music Information Retrieval Conference, pages 591–596, 2011, http://millionsongdataset.com/

3. Daniel Jurafsky, and James H. Martin. “N-gram Language Models.” *Speech and Language Processing*, 3rd ed., draft version, 2023, Chapter 3, https://web.stanford.edu/~jurafsky/slp3/3.pdf

4. Afshine Amidi, and Shervine Amidi. “Recurrent Neural Networks cheatsheet.” https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks

5. Alec Radford, et al. *Language Models are Unsupervised Multitask Learners*. OpenAI, 2019, https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.