CS230

Advanced Wind Music Classification and Generation

Benjamin I. Rocklin
Department of Computer Science
Stanford University
brocklin@stanford.edu

William J. Dunlop
Department of Linguistics
Stanford University
wjdunlop@stanford.edu

Abstract

While music generation is a relatively well-explored field of deep learning, most generators are limited to generating notes of static length and timing for the piano strictly. In order to build a more compelling and robust music generation model, we divide our task into two main segments: music classification and music generation. Despite limited time to accomplish the latter, we reached interesting results from a musical standpoint, and in doing so managed to build a LSTM-based RNN that is quite effective at predicting the next note in sequences of trumpet soloist music.

1 Introduction

Music generation has been done to death by now, with many successful models in existence that can build a simple melody that is pleasant to the ear given relatively simple parameters. Even more complicated types of music such as canons can be consistently generated. Despite these advances, generating a song that has non-uniform note patterns remains difficult, as the pitches, times between notes, and lengths of each note in a sequence can influence those that come later. Many approaches to melody generation have utilized music arranged for the piano due to its high availability. As different instruments typically play different roles in an ensemble, whether as the melody or the accompaniment, so too do certain musical instruments' physical limitations obligatorily pose restrictions on certain aspects of composing music for that instrument. – for example, while a piano can easily play notes in different octaves in quick succession, a wind player might struggle greatly to accomplish the same feat. One defining characteristic of brass music in particular is their implicit range and interval-jump restriction, as it is physically taxing for a brass player to play higher pitches for an extended period of time or play notes that are a large distance apart.

Given these successes and challenges, we separated our task into a division between music classification and generation, with the goal of capturing natural musical rhythm, accompaniment techniques, and instrument-specific restrictions. The input to our classifier algorithm takes a series of notes encoded as tokens (more on this in the Dataset and Features section) and, given this, attempts to output the next token in a given sequence via a deep network that combines LSTM and dropout layers with a softmax output. Each token represents either a note beginning, a note ending, or a period of time elapsing. Our generator takes as input a random sequence of tokens from the music dataset (identical to the classifier's input) and outputs a MIDI file comprised of the sequence of notes that is generated. This sequence is generated by running our classifier model over the generated sequence of notes.

2 Related work

Music generation is an incredibly well-researched topic, with many well-known and successful implementations already existing and widely used. Nevertheless, most of these are plagued by the

CS230: Deep Learning, Winter 2018, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

fact that they either are piano-specific, which enables music to be generated that may not be possible on other instruments; are applications specific, such as generators that can only produce canons or certain genres and styles of music; or abstract away too many important musical details, such as note length, timing, and rhythm.

Our project was initially inspired by the work done by Skúli [1]. Skúli's work is incredibly well known amongst other music generation practitioners and is considered to be the golden standard of basic music generation with RNNs; anecdotally, several other students' projects in other classes that the authors of this project are familiar with were inspired by this same work. While certainly a well-executed project, this project is rather basic, as it simply focuses on generating sequences of chords and notes for the piano of static lengths. That is, no variation exists between the different notes' lengths and rests between each note. Additionally, Skúli notes that the music generated by the algorithm is hardly playable by humans, as the generated music contains many very strange artifacts. That said, it is successful insofar as creating note sequences that sound pleasurable to human ears. Due to the similarity of our problems, we built our initial Keras model to be identical to that Skúli used and performed an architecture and hyperparameter search from there, which turned out quite well. Additionally, we used some of Skúli's code for converting sequences of notes into a dataset and for basic music generation. Given more time, we had hoped to explore additional methods for generation.

One project which laid much of the groundwork for our explorations was Clara, a MIDI generation project by Christine Payne. Payne's work [2] differed from Skúli's due to its ability to support multiple instruments. Additionally, it had support for violin MIDI generation, which, although it has physical limitations, doesn't have as many as trumpet. Rather than Skúli's model, Payne used FastAI's presupplied AWD-LSTM network. While this yielded good results, we decided to avoid this approach in order to allow ourselves more flexibility in altering our model and searching for good hyperparameters. Nevertheless, her work yielded generated music that sounded far more natural. Due to her use of encoding notes into tokens (the same format that we decided to use, more on this in the next section), she was able to produce notes of varying lengths and with different offsets in between note beginnings and endings. Consequentially, we borrowed this format in the hopes of reproducing the same behavior with our different model, and we based much of our code for building note sequences from MIDI files and converting sequences to new MIDI files around her work.

Finally, perhaps the most impressive generation was done by Fox [3]. Fox was able to produce different kinds of canons using AI; of particular note is Fox's crab canon generator, as crab canons are particularly difficult to generate due to the fact that they must be played from beginning to end and end to beginning. While Fox's work is rather unrelated to that of Skúli and Payne, her work played a significant role in helping shape our goals. We had initially hoped to generate crab canons using bidirectional RNNs but, finding this was already done successfully and took nearly a year to do so, we decided it was best to look into other techniques. This led us to find Skúli and Payne's generators and helped us form goals as a consequence.

3 Dataset and Features

We curated our own dataset from `8notes.com` [9], gathering their freely available MIDI files for trumpet, as well as for other instruments we hope to continue our work with in the near future. To separate the gathered MIDI data into tracks by instrument, we used the `mido` module by Bjørndalen [8] that provides objects and methods for interacting with MIDI. We have obtained just over 2,500 total MIDI files from the website, and selected only the 1,798 trumpet tracks for training.

Once we have our dataset of MIDI files ready, we convert them into another representation for our model. Because we use an RNN, our data takes the form of a series of sequences. For our inspiration of how to build these sequences, we took inspiration from Payne's approach and borrowed elements of her code to build our own sequences. Namely, we begin by converting the notes in the MIDI file into lists of where notes begin and end along with symbols representing periods of wait between these notes. For example, consider an example where a D is played on a trumpet for two "ticks" and a C is played after this. Note that all MIDI files are encoded in a time system of MIDI "ticks", so we use this as opposed to seconds and milliseconds as our main unit of time, also allowing our model to become tempo-invariant. Our sequence would then look like "tD wait2 endtD tC", where the "tD" token specifies that a D should be played on the trumpet (hence the t prefix), the "wait2"

token specifies that two ticks should occur before the next action, the "endtD" token specifies that the trumpet should stop playing the D at this time, and "tC" specifies that a C should then be played on the trumpet at the same timestamp in the MIDI. Although we do not make use of the prefixes currently, we intend to continue this project after the class and make use of multiple instruments in one stream of tokens.

Once we have the above tokens, we assign each start note, end note, and wait period (up to a maximum of length 24 ticks, whereupon a new additional wait period is added) a unique integer identifier. We then encode the list of the above tokens into these integer identifiers. For instance, if "tD" maps to integer 30, "wait2" to 205, and "endtD" to 68, "tD wait2 endtD" would instead be represented as $[30, 205, 68]$. From here, it is trivial to generate a list of sequences with a given length along with the next note in the sequence from these encoded lists to feed into the model, which will be discussed in greater detail farther down.

Due to the different lengths of the sequences, we have a different number of training examples for each value of the sequence length hyperparameter. For example, if a song had five notes and we chose sequences of length four as input, we would have one example comprised of the first four notes as input and the last note as the output note. On the other hand, if we chose to use sequences of length three, we would have two input sequences comprised of the first three notes and then the middle three notes. After hyperparameter tuning, we decided to report the results for sequences of length 10 with augmented data and 10 and 50 with no data augmentation. We use a 98/2 train test split and a 98/2 train dev split on the prior 98 portion of the split due to the large nature of our dataset,. With sequences of length 10 before augmentation, we have a total of 971577 sequences. This yields a training set of size 933102, a validation set of size 19043, and a test set of size 19432. With sequences of length 50, we have a total of 920197 sequences. This yields a training set of size 883237, a validation set of size 18026, and a test set of size 18394.

Additionally, we do have an optional data augmentation operation that can be used while compiling the dataset to increase its size by a factor of 12. We do not use this by default, as it makes the dataset far too large to train efficiently. We augment our data by shifting the pitch of each note up by one six times and down five times and append each of the additional eleven examples to our dataset for each sequence and output. We do not shift up or down 11 times in order to try and preserve realistic notes that a given instrument could play. For example, shifting the pitch up 11 times could bias our model towards always predicting very high notes for each instrument, and shifting the pitch down could yield a low note bias, both of which we hoped to avoid by centering the augmentation around the true note pattern.

4 Methods

After our architecture search, we decided to use an LSTM based model based on that of Skúli. For our hyperparameters, we ended up using a model with 3 LSTM-dropout layers, a dense layer, and an output softmax layer. The output layer can predict any label out of all the possible tokens: either an instrument start note, an instrument end note, or one of the 24 possible "wait" tokens specifying a duration either of a note playing or silence.

Using a softmax layer is useful because it provides the probability of one of several classes being true. The softmax function is defined as

$$f(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

It operates by calculating the prediction score for each class z_i and then picking the class with the highest score by assigning a probability to each equal to the equation above. Because each class's value is divided by the sum of all class scores, the softmax effectively yields the probability of predicting one class, as all these scores must sum to one, and the higher each class score is, the more likely that class is the true label for the example.

Our model's loss function is categorical crossentropy. The loss function is written

$$L(y, \hat{y}) = -\sum_{i=0}^{M}\sum_{k=0}^{K}(y_k^{(i)} * log(\hat{y}_k^{(i)}))$$

, where there are K classes and M training examples. In other words, it is the negated sum over all the examples of the sum over the true value for a given class times the log of the predicted score for that class (given by the softmax output described earlier). The inner sum will amount to 0 for every k besides the true k. Thus, it is effectively the negated sum over all of the examples of the log of the true class for each prediction. Thus, when the prediction is near 1 for the true class, the log will be 0, and the loss will be zero. On the other hand, if the prediction is near 0 for the true class, the log will be highly negative, and, owing to the negated loss function, the subsequent loss will be high. Consequently, we see that this works well for our multiclass prediction problem.

Finally, our model uses a series of LSTM and Dropout layers in the middle to achieve this. LSTM (Long Short-Term Memory) layers are a form of RNN (recurrent neural network). RNNs are neural networks where some of each neuron's state is fed into the next neuron along with an input. Consequently, this allows each neuron to gain a sense of time and sequence by accepting information from the prior neuron, which works well for sequential data such as audio waves, speech, and, in this case, music. One problem is that due to the fact that gradients (the way in which weights are updated) are backpropagated through neurons ahead to neurons behind, the gradients for neurons further back can "vanish" due to very small gradient magnitudes, resulting in these earlier layers not contributing much to later layers. LSTM layers solve this by introducing a series of gates and passing along additional information. In short, these gates function by deciding which information to keep and which to throw away, allowing some of the information from layers far earlier to make a more direct path to later neurons in the LSTM layer and have a greater impact, and other data that might not be useful to be ignored. Dropout serves as a safeguard against overfitting. Each neuron is given a probability to "drop out" or not be activated during a single pass. Because neurons drop out, the network during each pass is less complex, forcing each neuron to do more computatation to avoid bias and avoiding overfitting on the training set due to an overly complex model. As a result, while the model is still highly complex and can conform to complicated decision boundaries, the model will not overfit to the training set as often and should perform better on the dev and test sets.

5 Experiments/Results/Discussion

For choosing the optimal architecture, we performed a brief search on a few hyperparameters of the model, but were constrained by resources to running them for 5 or less epochs. For all of our evaluations, we consider as our primary metrics the categorical cross-entropy loss of a classification and as the accuracy. To yield an optimal sizing of the hidden layer of the LSTM, we compared two models' performance after 5 epochs.

# neurons		CCE Loss	Accuracy
128	Train	1.2267	0.6091
	Dev	1.2265	0.6087
	Test	1.2100	0.6130
256	Train	1.4420	0.5399
	Dev	1.4513	0.5362
	Test	1.4227	0.5484

While the LSTM model with hidden layers of size 128 had lower losses than the model with hidden layers of size 256, it seemed to have too high of a bias during training, and any initial edge it had in accuracy would later be surpassed by a more robust network. Due to constraints on time and computational power, we did not try any larger networks.

We started with two LSTM-dropout layers following the example of previous literature in this field. However, we evaluated our model with three layers as well as with four, in order to explore the possibility of having a deeper model.

Adding another LSTM-dropout layer seemed to make the model take longer to train per iteration, and, holding epochs of training constant, reduced the accuracy, presumably since a deeper model would take longer to train to reach equal performance. We decided to stay with 3 layers of LSTM-dropout as a result.

We also tuned the input sample sequence length. We took the network architecture we developed over the other tests and tested it on 10 tokens and 50 tokens. We used mini-batch gradient descent with

		CCE Loss	Accuracy
3x LSTM-dropout	Train	1.4420	0.5399
	Dev	1.4513	0.5362
	Test	1.4227	0.5484
4x LSTM-dropout	Train	1.4709	0.5296
	Dev	1.4861	0.5257
	Test	1.4653	0.5305

batch size 64. We selected categorical cross-entropy loss as our loss function, as in the hyperparameter searches, and trained each model for 50 epochs. In this below table, we have the accuracy and loss for different example sequence lengths, trained to 50 epochs each.

		CCE Loss	Accuracy
sequence size 10 w/ augment	Train	0.9880	0.6781
	Dev	1.0985	0.6576
	Test	1.0962	0.6587
sequence size 10 w/o augment	Train	1.0065	0.6762
	Dev	1.1150	0.6548
	Test	1.1031	0.6562
sequence size 50 w/o augment	Train	0.7321	0.7654
	Dev	0.9176	0.7149
	Test	0.9145	0.7152

Our models overfit the training set, to a degree. The effects of our data augmentation can be seen in the marginal increase in overall accuracy between the sequence size 10 with augment and that without.

We also managed to reach interesting results for our generation portion. In initial epochs of the training, outputs were just a single repeating note. See `midi_example_01` (epoch 10 of 50) in the repository folder `examples`. Generating examples from later epochs' weights, we were able to see that the model had learned musical scales, and was producing often-repetitive runs of notes in a scale. See `midi_example_02` (epoch 25). This is, qualitatively speaking, a very important part of the model's learning, as the model should first be able to pick up which notes are "in-key". The model was able to produce music with interval jumps, rests, and variation in rhythmic pattern. The first two of these are particularly significant to our motivation for choosing a wind instrument for this project, as the interval jumps were all playable, and the rests were spaced in a way that appeared to end musical phrases. See `midi_example_03`, which is from epoch 25, for an example of interval jumps, and see `midi_example_04` for an example of rests.

Later, more complex melodies began to emerge. Also see `midi_example_04` and `_05`, from epoch 40, for an example of this behavior. Often, the generation would appear to become stuck in a loop, typically about halfway through a generated sequence, owing to the fact that some of the trained model's predicted outputs chain together. This could be resolved by generating the next note via a probability distribution rather than taking the highest softmax score.

`confusion-matrix.png` in the code (it was too big to include here) contains a confusion matrix for our model with sequence size 50. As shown, the model mostly predicts along the diagonal, and most mistakes are in areas near the diagonal, showing that the predictions were accurate and it wasn't overfitting to any label. Note beginnings are smaller integers, endings are in the middle, and wait periods are large. The element on the diagonal with high frequency is the "wait1" encoding (encoded as 256), which appears in between every few notes and at the highest frequency by far. The matrix is very sparse due to the fact that many 128 MIDI notes never appear on trumpet in our dataset.

6 Conclusion/Future Work

For future work, we would recommend extending this technique to multiple-instrument classification/prediction and generation. (We may actually do this ourselves, for recreational purposes in this upcoming stretch of social distancing!). We would also recommend exploring the way that this technique can be used to capture and reproduce characteristics of different styles or genres of music.

7 Contributions

Ben focused on building an initial classification model, converting MIDI files into a dataset, and dataset augmentation. Will focused on performing the hyperparameter search over AWS, outputting generated sequences to MIDI files, and building the dataset. Both team members had a key role in decision-making for every step of building a model.

References

Prior Work/References (cited in IEEE):

[1] S. Skúli, "How to Generate Music using a LSTM Neural Network in Keras," How to Generate Music using a LSTM Neural Network in Keras, 09-Dec-2017. [Online]. Available: https://towardsdatascience.com/how-to-generate-music-using-a-lstm-neural-network-in-keras-68786834d4c5. [Accessed: 15-Mar-2020].

[2] C. M. Payne, "Clara: A Neural Net Music Generator," CLARA: A NEURAL NET MUSIC GENERATOR, 23-Sep-2018. [Online]. Available: http://christinemcleavey.com/clara-a-neural-net-music-generator/. [Accessed: 15-Mar-2020].

[3] E. Fox "Automated Canon Composition," Churchill College, 10-May-2016. [Online]. Available: http://minikanren.org/fox_diss.pdf. [Accessed: 15-Mar-2020].

Data and Libraries (cited in the library's preferred style where possible, IEEE otherwise):

[4] Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

[5] Keras, Chollet et al., "https://keras.io", 2015.

[6] Stéfan van der Walt, S. Chris Colbert and Gaël Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation, Computing in Science & Engineering, 13, 22-30 (2011), DOI:10.1109/MCSE.2011.37 (publisher link)

[7] M. S. Cuthbert and C. Ariza, "Music21: A Toolkit for Computer-Aided Musicology and Symbolic Music Data.," in ISMIR, 2010, pp. 637–642.

[8] Mido, Bjørndalen et al., "https://mido.readthedocs.io/en/latest/", 2013.

[9] D. Bruce, "Free Sheet Music & Lessons," Free Sheet Music & Lessons, 2001. [Online]. Available: https://www.8notes.com/. [Accessed: 15-Mar-2020].

Code repository: https://github.com/BenRocklin/cs230proj