Barebones Music Generation

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

Jacob Meisel
Department of Electrical Engineering
Stanford University
jameisel@stanford.edu

Abstract

This project seeks to challenge the boundaries of music generation by trying to generate new musical pieces via simple models such as standard neural networks. Through a series of different models trained on different combinations of notes in several classical piano songs, we were able to build such a generator that yielded limited success in generating pleasant-sounding music.

1 Introduction

Many complicated music generation algorithms exist such as RNNs that can generate music which most human listeners find enjoyable. However, such algorithms can be difficult to implement and understand by people who are not experts in both music theory and computer science, as they rely on highly complex and expensive models that take a long time to train along with large feature sets that are heavily reliant on music theory experience. Consequently, we decided to challenge these complex models by trying to take a simple machine learning model such as a standard neural net and combine it with basic sets of notes as features to generate pleasant-sounding music. We hoped to achieve performance nearly as high as others achieved via RNN/LSTM, CNN, or GAN models, which tend to be the norm; doing so would imply that these heavyweight models may actually be overkill in many situations such as these, and others may be better served trying simpler models first.

Therefore, we summarize the function of our model as follows: the input to our algorithm is a set of songs, which are then converted into our features, a series of notes preceding the note we want to generate. The number of prior notes before each note we wish to generate is given by a hyperparameter for the problem we labelled m. Finally, we pass these features into both a linear/logistic regression hybrid model as a baseline and a neural network as our main model. Our models have three outputs: delay, duration, and pitch. The significance of each of these will be described in greater detail in Section 3. After pitting these values against the proper values of the training notes as described in section 4, we begin predicting new songs with our trained model, where our input is the previous notes we've generated already and the output are the delay, duration, and pitch of the note that we would like to add to our generated song.

2 Related Work

Current research on music generation can be largely differentiated by the models used to generate new works of art and the techniques used to encode the desirable aspects of the music. The relevant works we explored employed CNNs [1], GANs [2, 3], and RNNs [4, 5]—including the LSTM variety—and encoded the music at the single-note level [4, 5], multiple-note level [2, 3], and audio-waveform level [1].

WaveNet [1] uses a CNN with causal, dilated convolutions to generate raw audio waveforms for text-to-speech or creative music composition applications. However, significant weaknesses are the

large computational and storage costs associated with manipulating time-domain waveforms—which need to be sampled fast enough to represent high frequency sounds at an acceptable level of quality. Additionally, since WaveNet composes a raw waveform rather than score of notes, it is rather difficult for a human wishing to tweak a song or use it as creative inspiration to modify the song.

MuseGAN [2] and MidiNet [3] use GANs with CNNs as the generator networks to compose music one bar at a time. MuseGAN can generate multi-track, polyphonic music while evaluating musical traits such as the ratio of empty bars, the number of unused pitch classes per bar, the ratio of unqualified notes, the drum pattern, and the tonal distance.

Maia [4, 5]—a new music generation project out of UC Berkeley—attempts to compose classical piano pieces in the style of Mozart, Chopin, and Bach using a two-layer stacked LSTM. It encodes midi-files into a sequence of tokens, which can correspond to the beginning of a note, the end of a note, or the passage of time. In this sense, Maia is similar to our model. However, while we attempt to directly predict each note's durations, pitches, and times until the subsequent note, Maia attempts to predict tokens, which do not have a one-to-one correspondence with notes. Most notably, it represents the durations of notes and times between notes using the "wait" tokens present between "note-start" tokens and "note-end" tokens.

Due to the high complexity of existing music generation techniques—including computational complexity, encoding schemes, and complicated models—we decided to push the limits of music generation and attempted to create the simplest model possible that could still successfully compose pleasant music.

3 Dataset

Our dataset consists of 327 piano MIDI files made by a variety of classical music composers and sourced from Krueger [6]. Recall that our parameter m determined the number of prior notes we take as features for each training example. As a consequence, the number of total notes we can predict is contingent on m, as there must be at least m notes played in the past before the note we try to predict. The two principal values we used were $m = 3$ and $m = 5$. We did an 85/15 split between training and test data, and this training data was further split into a training and validation data implicitly by the libraries we used for training. For our original train/test split, we have 643598/113577 for $m = 3$ and 642412/113367 for $m = 5$. While we did not explicitly perform a validation split as well, the libraries we used for training neural nets graphed performance on a cross-validation set that it used in training, so we used the output of these to determine which were our best hyperparameters for the net. The training graphs for each model that we used are freely available at the repository [11].

To create our labels from the dataset, we iterated over each note in the song with at least m prior notes existing before it (so if $m = 3$, all the notes after the third). We designate each of these notes the "label note" for each training example. We decided that every note could be described by a combination of three traits: its delay (how long until the next note in the song is played after the given note), its duration (how long the current note in the song is played for), and its pitch (how high the note is). The first two are continuous integer values, so they were predicted via linear regression for the baseline and ReLU activations for the neural network. Pitch was slightly more complicated: we considered it a multi-class classification problem and thus stored it as a one-hot vector. All units for time in this dataset are given in "MIDI ticks" so that our model can be time/tempo-invariant. This vector is of length 12 if we did not consider octave and of length 128 if we did, as there are 12 unique pitches (A flat through G) and 128 unique possible MIDI notes that can be played (A flat through G repeated over roughly 11 octaves). Whether or not we considered octave was another hyperparameter we toggled in our search for a good model, but we usually left it off for the sake of runtime. See section 5 for the results of turning this on.

For each note, we created a feature vector from the m prior notes. An illustration of this vector can be seen in Figure 1. Similar to the labelled notes, each of the prior notes, has three characteristics we use to describe them; they are the relative time offset (how long before the label note the prior note

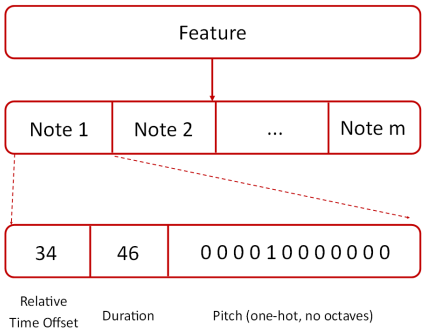


Figure 1: Feature vector formatting

was played, somewhat similar to delay), duration (as before, how long the prior note was played), and pitch (as before, what pitch the note is either as a length 12 or length 128 one-hot vector).

Finally, we provided the option to augment our dataset by using a simple moduation: we shifted each pitch up by one through eleven pitches. This augmented our dataset to be 12x the original size. Unfortunately, this bloated our run times and dataset sizes, so this option was left off by default at the cost of performance. See section 5 for results of using this.

4 Methods

We used three learning algorithms to train our models: linear regression, logistic regression, and a neural network (fully-connected, multi-layer perceptron).

We chose linear and logistic regression models for a baseline since these are the simplest models that can be implemented and used for our application. We trained both models via regularized full-batch gradient descent due to our small training set size. That is, we used the following algorithm to learn our parameters θ:

$$\theta_j = \theta_j - \alpha(\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} + \frac{\lambda}{m}\theta_j)$$

for all parameters j and repeats until convergence. Note that the gradient for each with respect to θ is the same, so both models follow the same algorithm for training. For the bias term, we excluded regularization, so we had that $\theta_0 = \theta_0 - \alpha(\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}))$. At a very high level, for each training example, the above algorithm takes our prediction $h_\theta(x^{(i)})$ and true label $y^{(i)}$ and computes the difference, or error. It then multiplies by $x_j^{(i)}$ yielding the gradient of the loss with respect to parameter θ_j. For non-bias terms, we add in a hyperparameter λ and multiply it by the parameter we are seeking to optimize θ_j in order to increase the size of the gradient term (the term in the parenthesis, or the sum). This gradient term is then subtracted from θ_j, resulting in θ_j changing such that the loss is minimized with respect to itself. If we consider the regularization term $\frac{\lambda}{m}\theta_j$, this has the effect of decreasing θ_j further and keeping θ_j low to try and minimize loss, which helps reduce variance. We do not regularize the bias term since it harms our model in practice and would not lead to any significant overfitting that the other parameters wouldn't cause, as it is nothing more than an intercept term.

For our linear regressions, we form predictions both during training and during post-training by setting $h_\theta(x^{(i)}) = \theta^T x^{(i)}$. The prediction may be thought of as a weighted sum of each feature $x_j^{(i)}$ in training example i with the model's associated weight θ_j. For pitch, we train one separate logistic regression model for each individual class via one-vs-all classification and predict the class k with the highest score $h_{\theta_k}(x) = \frac{1}{1+\exp(-\theta_k^T x)}$.

Our third model is a neural network. This network in particular has many layers: one input layer (simply the feature vector $x^{(i)}$ which we want to predict), several hidden layers of *neurons*, and one output layer with one output neuron that provides our prediction for the ground-truth label. Each neuron in the hidden layer can be found by applying an *activation function* to modify the result of some multiplication between a set of weights and all the nodes in the prior layer. For example, assume there is some set of weights θ associated with the first neuron in the hidden layer. Then the value at this neuron will be computed as $f(\theta^T x^{(i)})$ for a given training example $x^{(i)}$ and activation function f. Usually, these weights are all stored in a matrix W rather than a single vector θ.

For our activation function in our hidden layers (the middle layers) and non-pitch output layers, we used ReLU, defined as $f(x) = max(0, x)$. This activation function is useful because all positive output is linear (meaning that the output will be identical to the input for positive values) but negative values will result in the neuron being 0 and thus "not activated." Realistically, we used this activation since it worked well in practice for our model, but it is generally the most used activation function for a number of reasons. One is that the gradient will always be constant (1 if the neuron activation is positive, 0 otherwise) so the gradient neither explodes to huge values nor vanishes. Another reason is that since many neurons may be 0 with this activation, backpropagation (the act of propagating error loss back to all weights in order to minimize loss overall similar to gradient descent for logistic regression) can be performed far more quickly. Also, our non-pitch note labels are all values greater than or equal to 0, so ReLU fits this output well by restricting output to be non-zero.

Our pitch output layer activation is the softmax function given by $f(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$. This function is commonly used in multiclass prediction problems because of its output range. 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.

Once we have trained all the weights in this neural network or the baseline, we begin the process of music generation by randomly choosing m notes from our dataset. From here, we use our models to generate notes one after the other. We discard the first few notes to purge any dissonance caused by initializing our song randomly. We then convert the list of notes into a MIDI-compatible format and write out each note to a MIDI file.

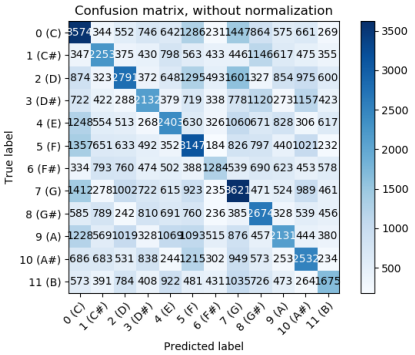
5 Experiments

For our logistic regression model, the library we used automatically computed the ideal learning rate and regularization term using an internal cross-validation process, so we did not need to toy with these parameters. We also chose to use full-batch gradient descent as mentioned above due to try to maximize our accuracy, as it didn't take too long to run our baseline on any model. However, for our neural networks, we graphed the loss over a validation set versus each epoch to tune the number of hidden layers and neurons in each hidden layer for our neural network, settling on four layers and 100 neurons per layer. In evaluating our project, we considered several metrics: RMSE for delay and duration, and accuracy and a visual confusion matrix for the pitch. Additionally, for evaluating the final generated songs, we simply used a subjective metric and input from colleagues by judging if the songs sounded good. Unfortunately, we ran out of time to formally compile a report of responses due to this deadline. The general consensus was that while our models were not great at producing full original music pieces, some neural net models (especially those with m=5) produced interesting and original snippets; however, they had issues generating full songs. On the other hand, our baseline tended to just pick one or two notes and keep playing them infinitely until the song ended. Example output MIDI files can be found under the generatedSongs folder in the repository [11]. Shown below in Table 1 are the objective performances of our models. In our dataset, the standard deviation for the delay label was 183, and the standard deviation for the duration label was 293.

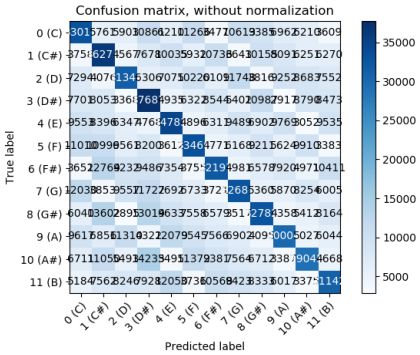
As we see, our neural net with m=5 far outperformed all of our other models. Delay and duration were moderately successful with RMSEs in the 100s (meaing each was off by 100-250 ticks or so on average), but these numbers could certainly be better. On the other hand, the "low" accuracy is quite deceiving. Take a look at the corresponding confusion matrix for the neural net with m=5 evaluated over pitch in the testing set below in Figure 2a.

Table 1: Relative Model Test Set Performances

Model	Delay RMSE	Duration RMSE	Pitch Accuracy
Baseline, m=3	151.4	269.2	19.87%
Baseline, m=5	149.4	272.5	21.87%
Neural Net, m=3	131.7	251.7	25.05%
Neural Net, m=5	122.4	248.7	26.65%
Neural Net, Dataset Augmented, m=5	115.2	227.4	28.99%
Neural Net, Octave Considered, m=5	120.3	385.3	14.00%



(a) NN Confusion Matrix, m=5



(b) NN Confusion Matrix, Dataset Augmented, m=5

As shown above, our model picks the correct pitch far more often than the incorrect one for every given note. The accuracy is additionally driven down by the fact that for any given number of notes c played in a chord during training, our features will be identical, so the maximum possible accuracy in the notes for this chord in our model is given by $\frac{1}{c}$ as the same label will be predicted for each. This manifested in our generated music as well, where the pitch progressions between notes were actually quite pleasant as rated by others, but the timing and rhythm sounds incredibly off and random. It seems that, in general, we suffered high bias on duration and delay prediction (and it may not be possible to do this well with the problem we posed at all), while our models for pitch prediction turned out to be okay.

Furthermore, we added parameters for augmenting the dataset and factoring octave into note pitches as discussed in section 3. Augmenting the dataset yielded moderately improved performance metrics (Table 1), and improved our confusion matrix for pitch prediction (Figure 2b). Taking the octave into consideration only reduced our pitch accuracy by a factor of two even though it increased the number of pitches by a factor of 10.67.

6 Conclusions

While our models certainly had some issues, our model for pitch turned out surprisingly well, even predicting chords automatically that sounded good together for the neural network with $m = 5$. If more time were available, putting resources into improving our delay/duration models or perhaps even ditching them altogether for a more expensive one or different approach altogether would be the next step in the right direction. Additionally, we merged the notes that the right and left hand played on the piano for the purposes of both training and generation, so it might be interesting to see if training two models separately on each hand would result in a better outcome; this might even be worth exploring before trying to fix duration and delay, as the left hand typically plays longer bass notes. Finally, since the music was from a mashup of many different composers over an incredibly long period, it might be worth trying to train it on just one composer's work or even a different instrumental song genre like electronic music to see if this produced more reliable results.

References

All non-library citations are given in IEEE. Other citations are given in the format requested by the source.

[1] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "WaveNet: A Generative Model for Raw Audio," *arXiv preprint*, September 12, 2016, [Online]. Available: https://arxiv.org/abs/1609.03499. [Accessed December 12, 2019]

[2] H. Dong, W. Hsiao, L. Yang, and Y. Yang, "MuseGAN: Multi-track Sequential Generative Adversarial Networks for Symbolic Music Generation and Accompaniment," in *The Thirty-Second AAAI Conference on Artificial Intelligence, AAAI-18, New Orleans, LA, USA, February 2–7, 2018*, [Online]. Available: https://arxiv.org/abs/1709.06298. [Accessed December 12, 2019].

[3] L. Yang, S. Chou, and Y. Yang, "MidiNet: A Convolutional Generative Adversarial Network for Symbolic-Domain Music Generation," in *The 18th International Society of Music Information Retrieval Conference, ISMIR 2017, Suzhou, China, October 23-27, 2017* [Online]. Available: https://arxiv.org/abs/1703.10847. [Accessed December 12, 2019].

[4] A. Spezzatti, "Neural Networks for Music Generation," *Towards Data Science*, June 24, 2019. [Online]. Available: https://towardsdatascience.com/neural-networks-for-music-generation-97c983b50204. [Accessed December 12, 2019].

[5] T. Edward, *Maia*. [Online]. Available: https://edwardtky.wixsite.com/maia. [Accessed December 12, 2019].

[6] B. Krueger, *Classical Piano Midi Page*. [Online]. Available: http://www.piano-midi.de/. [Accessed: December 12, 2019].

Libraries:

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

[8] Tensorflow (https://www.tensorflow.org/)

[9] Numpy (https://numpy.org/)

[10] Mido (https://mido.readthedocs.io/en/latest/index.html)

Code Repository:

[11] Git Repository (https://github.com/BenRocklin/CS229Project)

Contributions

Jacob helped set up the machine learning framework and perform parameter tuning on the models. Ben handled MIDI import, feature extraction, and writing to new MIDI files. Jacob and Ben contributed towards the music generation process as well.