Musical Local Conditioning for WaveNet

Ryan Wixen
rwixen@stanford.edu

Drew Wadsworth
swads@stanford.edu

Ricky Parada
rparada@stanford.edu

Abstract

Interest in generative deep learning models has spiked over the past several years, and among the most popular applications of these models is AI generated music. While objectively harder to evaluate than more structured problems, this obstacle makes music generation a more fascinating problem in our view. While there are currently a variety of existing models that aim to sound like particular instruments, it remains a challenge for implementations to learn targeted melodic features. With this in mind, we built upon WaveNet, a state of the art text-to-speech model, by expanding its architecture to condition on these local musical features. Our model generates realistic sounding piano samples whose notes can be controlled by local conditions.

1 Introduction

Thanks to improvements in Natural Language Processing (NLP), text-to-speech (TTS) models such as WaveNet [7] have been able to both understand what a particular speaker sounds like (global condition) and arrange their sounds in a logical way to emulate human speech (local condition). Our model seeks to use the same network to enforce aspects of human composition during piece generation. (in addition to learning simply what a piano sounds like). At present, the WaveNet implementation can generate music, though it lacks the local conditioning helpful for replicating existing works. That is, the output can sound like a piano, but only through a collection of seemingly unorganized notes that swing between styles very quickly over a short time interval. This is equivalent to an NLP model outputting unorganized phonemes rather than structured word-forming syllables. To accomplish more realistic generation using local conditioning, our model will take in an audio waveform annotated with MIDI pitches of the notes being played over time, which are local conditions of the audio. We then feed this input to a 1-dimensional Convolutional Neural Network (CNN) described in our approach section, which outputs a WAV audio sample aiming to sound like the input sequence.

2 Related Work

A variety of other projects regarding piano generation using TTS methods have been conducted in recent years. This section presents the results of papers using similar models and highlight the similarities and differences with our approach.

Oord et al., 2016 (WaveNet): WaveNet is Google's deep generative model of audio waveforms, specializing on human voice inputs. It is our project's primary source of inspiration, as it is a parametric TTS model that allows conditioning on additional inputs. The open source implementation of this model we use in this paper was trained on a variety of classical raw audio samples and generates fascinating, yet discombobulated piano pieces. We plan to improve upon the open source implementation's music generation capabilities by inserting local conditioning as described in the original WaveNet paper [7].

Paine et al., 2016 (Fast WaveNet): A group of researchers from IBM and the University of Illinois, Urbana-Champaign improved the time complexity of the naive WaveNet implementation by caching

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

results of previous calculations. Our implementation utilizes this speedup during generation to output longer samples, which can take an extremely long time without caching [5].

Kotecha et al., 2017 (Generating Music using LSTM): Kotecha uses a model similar to WaveNet, but one trained on MIDI representations of 20 of Bach's fugues. The model further utilizes a two-layered LSTM architecture with separate LSTMs for the time axis and the note axis, which is also referred to as a bi-axial LSTM. We instead opted for a single memory model with the local conditioning as an additional input because this approach was more consistent with the WaveNet description. Kotecha uses negative log likelihood for a loss function (aka binary cross entropy), which is also seen in the WaveNet implementation [4].

Payne et al., 2019 (MuseNet): MuseNet is the new state of the art music deep neural network generator, famous for creating up to 4 minute musical compositions with up to 10 different instruments. It can additionally combine styles/genres from country to Mozart to the Beatles. The defining principle behind the model is that it learns everything from scratch including harmony and rhythm patterns, which is what our model aims to do albeit on a much lighter dataset. These learned features in turn help the model generate pieces that align with our human understanding of what music is [6].

Each of the aforementioned works feature different models that approach the same task: music generation using TTS methods. They each employ datasets in the form of raw audio or MIDI encodings, though some don't allow conditioning on additional inputs. Our goal is to generate a model that provides this functionality, training on more focused data that fits our predictive needs.

3 Dataset and Features

Our model is trained on two MIDI sources, a classical sample from ADL Piano MIDI [3] and another jazzy sample from a collection of live jazz by Doug McKenzie [1]. We trained the model on two songs, Debussy's "Suite Bergamasque" from ADL Piano and Brubeck's "Strange Meadow Lark" from McKenzie, which comprises a sufficient dataset size since our input is audio at sampling rate 16,000 (16kHz raw audio). Although we only have 2 songs from two genres, there is 11 minutes of audio between them, which is plenty to train the model. As a side note, most pieces in the ADL MIDI dataset extracted only tracks with instruments from the "Piano Family" on songs with multiple instruments (including "Suite Bergamasque").

The dataset was preprocessed in order to simplify the problem that the model must learn. To that end, the MIDI files were edited to be monophonic, meaning at most one note plays at a time. It might be harder for the model to learn to map polyphonic pitch inputs to audio. The audio was also reduced from stereo to only one mono channel, so the model only has to learn a single output stream. We also include a local conditioning stream of the current MIDI pitch being played at each timestep. This second input is at the same sample rate as the audio. MIDI pitch numbers range from 0 to 127, so they fit in the unsigned byte input datatype. MIDI pitch 0 represents a rest and 60 represents C4. We created this input dataset by processing the MIDI file the was used to produce the audio data.

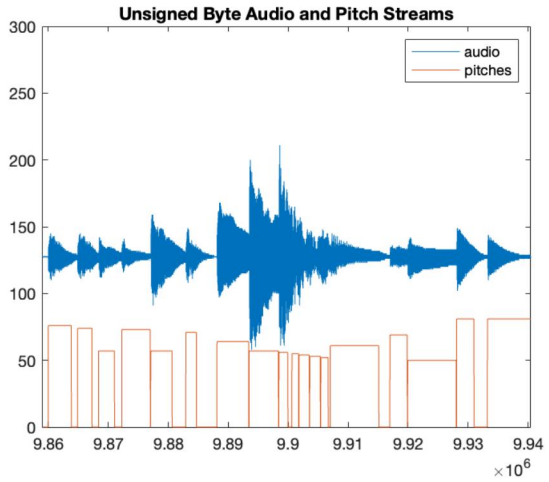

Figure 1: Segment of Training Dataset with Audio and MIDI Pitch Data

4 Approach

As a starting point for our project, we found an open source partial implementation of WaveNet, called pytorch-wavenet. Like every other open source implementation we found, this implements the standard (non-conditional) WaveNet, which encodes a single speaker identity and can generate the next sample of audio output given a time-series of previous samples. But, this implementation lacks any conditioning through which the model could be directed to speak specified words or play specified musical notes. So, we aimed to add local conditioning to the model, telling the model which note to play at a given time.

Our model is trained on an AWS instance of type p3.2xlarge with 8vCPUs and an Nvidia Tesla V100 SXM2 GPU. Even with this powerful machine, training was rather slow, taking multiple hours per epoch. However, we found that training for a just a few epochs generated acceptable results. Batches are appropriately sized to use most of the instance's memory.

We trained on our own dataset: audio generated by a MIDI piano emulator in Logic. To validate performance, we seed the model with a segment of input from the training data distribution and repeatedly sample the model's output, feeding back in each generated sample as part of the input to generate the next one.

The model uses a binary cross entropy loss function that compares the model's confidence in each one of 256 possible output sample values to the true output sample label. This method does not take into account how far off its prediction is; therefore, predictions that are close but slightly wrong are penalized equally to those that are very wrong. So, we considered changing the cost function to be a regression so that worse guesses result in higher loss than predictions that are just slightly wrong. However, we chose not to do this because a regression can be harder to learn than a classification, and existing work tends to use one-hot outputs rather than scalars.

We also considered changing the input encoding from unsigned byte-length integers to a float representation that is normalized to the range $[-1, 1]$, thinking that this would help avoid vanishing gradients produced by both the sigmoid and hyperbolic tangent activation functions. However, upon further investigation of the starter implementation, we found that the input samples are converted to one-hot vectors of size 256. Specifically, the value of each unsigned byte corresponds to the hot index of each vector via a scatter. Changing the inputs to floats would have no benefit because the actual one-hot inputs to the model are already vectors of 0s and 1s, so vanishing gradients are not a problem. Further, for the MIDI pitch inputs, a one hot representation makes intuitive sense because, unlike the audio samples, the MIDI pitches do not represent a physical scalar quantity but rather an abstract encoding of the data.

The WaveNet model uses multiple convolutional layers with identical architecture. The outputs of each layer are skipped to the end of the model, where they are summed.The hidden layers are composed of a dilation convolution, a residual connection, two non-dilated convolutions: $\mathbf{p} = W_{f,k} * \mathbf{x}$ and $\mathbf{q} = W_{g,k} * \mathbf{x}$, and a gated activation function: $\mathbf{z} = \tanh(\mathbf{p}) \odot \sigma(\mathbf{q})$ where $W_{f,k}$, the convolution kernel for layer k under the $\tanh$ activation, is called the *filter* parameters, $W_{g,k}$, the convolution kernel for layer k under the sigmoid activation, is called the $gate$ parameters, $\mathbf{x}$ is the input to the layer, and $*$ is a convolution. A dilation convolution increases the size of the convolution kernel by placing a number of zero elements between each element in the original filter. We call a dilation convolution an l-dilation convolution if the number of 0 elements placed between each existing element is $l - 1$. For example, if we start with the following kernel: $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ it would, for a 2-dilation, be applied as: $\begin{bmatrix} a & 0 & b \\ 0 & 0 & 0 \\ c & 0 & d \end{bmatrix}$

After the activation is applied on a layer, the output goes through an additional length 1 convolution, and the input to the layer is added back in to the output. This last addition is the residual connection. As a side note, the WaveNet paper describes that the residual comes from before the input reduction (dilation), but in our starter implementation, the residual comes from after the dilation but before the convolutions. When taken after the dilation, the shape of the residual signal better matches the shape of the output signal to which it will be added, so this change in the implementation is reasonable.

The input to the existing model is a time-series of one byte unsigned integer values representing samples taken of some audio waveform. The output of the model is the predicted next sample. In our model, we add an additional input and convolution to select layers, which the model can condition on. As described in section 3, we provide MIDI pitches for the model to emulate. This is analogous to providing a text-to-speech model phonemes to generate speech for. With this local conditioning, the equations for $\mathbf{p}$ and $\mathbf{q}$ above then become

$$\mathbf{p} = W_{f,k} * \mathbf{x} + V_{f,k} * \mathbf{y} \text{ and } \mathbf{q} = W_{g,k} * \mathbf{x} + V_{g,k} * \mathbf{y}$$

where $V_{f,k}$ and $V_{g,k}$ are additional convolutional filters and $\mathbf{y}$ is the additional channel of conditioning data. Note that this method of providing this extra conditioning data is described in the original WaveNet paper [7], but it is not provided in the open source implementation of WaveNet that we are building on top of for this project [2]. The structure of this conditioning input is described in section 3, and the architecture of a single layer of this model is given by figure 2. Note that the two outputs at the top of the diagram are the local condition and activation inputs to the next layer, respectively.

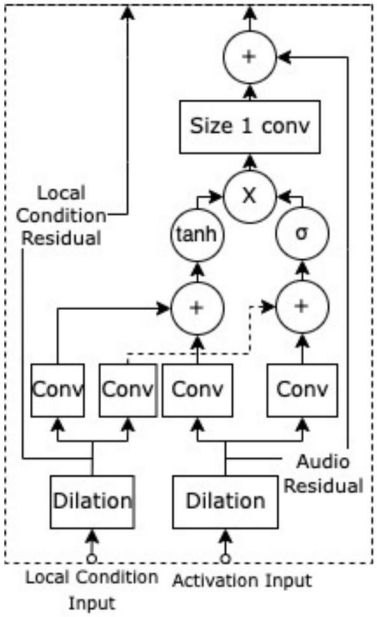

Figure 2: Architecture of One Layer

We have created a model designed to learn how to take a desired pitch as input and output a full-bodied audio waveform that sounds like the instrument the model has been trained to emulate, which involves producing the characteristic overtone frequencies and envelope for that instrument.

5 Experiments/Results/Discussion

The model learned to follow the local conditioning pitches somewhat successfully. We tested how our model had learned local conditioning by manipulating the local conditioning data that was provided to the model as a guide during generation. The greatest indication of success we saw was when we made the local condition change pitches midway through generation to a note different than the one it started on. The model correctly introduced the new pitch later in the generated audio, as we would have hoped. We chose the second pitch, Bb, to be one that had appeared multiple times in the training set following the previous pitch, A, which we believe contributed to the model's ability to achieve this transition between notes. However, the transition was not perfect. Most notably, the second note doesn't come in with a clear attack, but rather fades in. Further, the first note continues to sustain after the second note comes in. Regardless of these imperfections, we were very pleased that the model learned local conditioning well enough to play a note on demand.

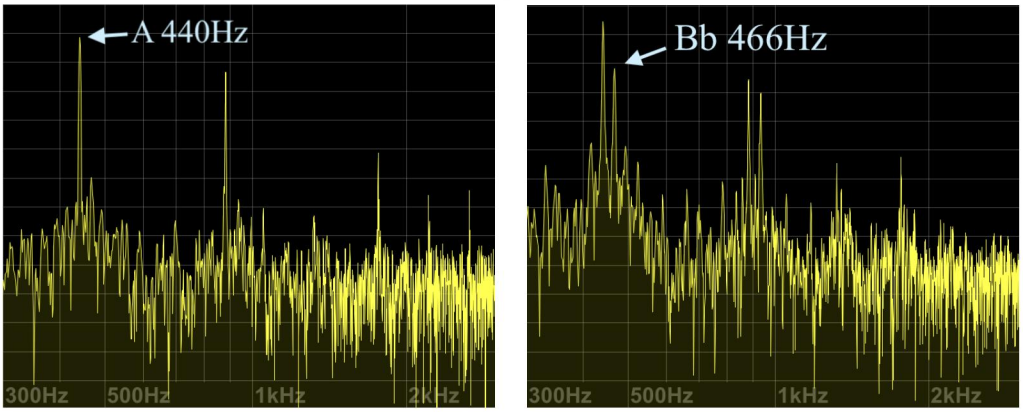

Figure 3: Spectra of generated audio at two points in time illustrating that local conditioning brings about a second pitch, Bb, while the first pitch, A, sustains

In general, the trained model did not do a good job of stopping notes. When we inserted a rest into the local conditions for generation, the current note did not stop as expected but kept playing. This

may be somewhat reasonable because piano notes do continue to decay a little after the note stops being played. Perhaps increasing the receptive field of the model would improve behavior in this area by giving the model better knowledge of how notes evolve over time.

Training on other local conditions gave mixed results. For example, when the seed local condition pitch did not match the actual audio pitch, the model correctly recognized this as an unseen part of the input space and accordingly output noise. When we local conditioned the model to play a sequence of notes it had not seen much before, it did start to change notes, but did not change to the correct note. As mentioned in the Future Work section, data augmentation could be used to make sure there are no notes that the model has not seen, as well to expose the model to more sequences of notes.

Without local conditioning, the model had 1834592 parameters. After implementing local conditioning throughout the first block, the model had 1884416 parameters. As expected, this increase indicates that the model is slightly more complex to be able to do local conditioning. We saw unpredictable results when training a model that had local conditioning on every block, like a jump in training loss in the second epoch that did not go back down.

The WaveNet paper describes local conditions being processed with a 1x1 convolution rather than a dilated size 2 convolution as is done for the audio. Because it could be beneficial to the model to have access to prior local condition, the model we built uses size 2 dilated convolutions for both the audio and the local conditions. Though it is more computationally expensive, it would be useful for the model to know not only that a certain pitch is currently playing but also when it started playing, as notes evolve and decay over time. This is less true of local conditions for speech, like phonemes or spectrograms, that do not change over time in the same way. This difference between speech and musical data justifies our choice to diverge from the model described in the paper, which focuses on speech, by using larger convolutions for our musical local conditions.

6 Conclusion/Future Work

After extensive training and testing, our model correctly learned to play a successor note on command via local conditioning, a feature that wasn't captured by the starter WaveNet implementation. However, the model still struggles when transitioning to another pitch or to a rest, as indicated by the slowness of pitch transitions or noise presence. Regardless of these shortcomings, the fact that a previously seen note appears in the generated samples suggests that even the quality of state of the art models such as WaveNet can be improved upon. Though a reputable result was achieved by adding local conditioning, there is much more to be done in creating a robust generative model. Considering the limitations of our model as presented in our results, we hope to enhance the CNN model with the following additions:

Currently, we simplified the problem by training the model on monophonic data, where at most one note is playing at a time. The model could be trained polyphonic data, which is more musically interesting, by changing the one-hot pitch input vectors to be multi-hot such that each hot index corresponds to a note currently being played. The model should be able to generalize to this more complex problem.

Our model is currently capable of roughly playing a given sequence of pitches. In order to better learn the envelope of each note as well as to become able to generate its own musical sequence of pitches, as suggested in the Wavenet paper, the model might need a longer receptive field to learn such long term musical information.

Our current dataset does not cover all notes and all transitions between notes. In order to increase the output space that the model is capable of generating, we need to increase the input space of the training data. The dataset needs to have all notes that the model could be expected to play. In order to accomplish this, we would augment the dataset with transposition, or shifting by a constant pitch, akin to translation in image augmentation.

7 Contributions

Ryan was responsible for creating datasets and contributed to the group's vision of the project. He made the audio files that went into our datasets and preprocessed them to be an appropriate sample rate, normalization, and number of channels. He wrote the script that converts a MIDI file into a local

condition stream at the same rate as the audio. Also, he developed some of the ideas that shaped the direction of the project and played a role in interpreting what we found along the way. He also made visualizations of the dataset and results and did his share of managing training and testing.

Ricky wrestled with changing the audio inputs to floats before the discovery that this switch was futile due to one-hot encoding. He was also responsible for generating sample clips on his local machine once the AWS instance gave us the trained weights. He further took the lead on writing the project overview given in the abstract and introduction sections as well as the related works section. He also wrote the conclusion section excluding future work.

Drew spearheaded the implementation effort to add local conditioning to the Pytorch code. He made changes to the model structure, the forward pass Pytorch function, the dataset structure in the Python code (which allowed us to shuffle both audio and local condition datasets together), the training code, and the generation code. Drew also discussed the implementation approach with Ryan and Ricky, whose ideas greatly assisted the implementation effort. Ryan and Ricky both, of course, also contributed code to this implementation effort. Drew ran and debugged the model during the implementation stage, and he came up with ways to incorporate the local conditioning data into the model (e.g. reducing the local condition before passing it to the next layer, to make tensor shapes compatible). Drew wrote the second half of section 4, Approach, where the model architecture and our changes to the architecture are discussed.

References

[1] `https://bushgrafts.com/midi/`.

[2] `https://github.com/vincentherrmann/pytorch-wavenet`.

[3] Lucas N Ferreira, Levi HS Lelis, and Jim Whitehead. Computer-generated music for tabletop role-playing games. 2020.

[4] Nikhil Kotecha and Paul Young. Generating music using an lstm network, 2018.

[5] Tom Le Paine, Pooya Khorrami, Shiyu Chang, Yang Zhang, Prajit Ramachandran, Mark A. Hasegawa-Johnson, and Thomas S. Huang. Fast wavenet generation algorithm. *CoRR*, abs/1611.09482, 2016.

[6] Christine Payne. Musenet, 2019.

[7] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W. Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *CoRR*, abs/1609.03499, 2016.