Lo-Fi Music Generation with Sequential Models

Jason Xu and Ashwin Swar
Tufts University
jason.xu@tufts.edu
ashwin.swar@tufts.edu

Abstract

Music generation has been a getting attention in the Deep Learning literature. Especially since the introduction of Wavenet, waveform generation, in general, has been getting popular. In this paper, we talk about our attempts at generating waveform music from a small initial sample. We talk about tow different models, one temporal convolution based and other Fourier Transform LSTM based, and describe the pros and cons of each and the challenges we faced in the process. We hope that this paper will be a good starting point for anyone interested in this topic.

1 Introduction

Generating new samples from a probability distribution has been a very popular use of deep learning methods. Generative Adversarial Model(GAN), Variational Auto-Encoder(VAE) are some popular examples of it. These are probabilistic models where we try to model the conditional distribution $P(X|c)$, where X is the sample and c is some latent information vector. Once we have a model of this distribution function we can then sample from this distribution to generate new data. This idea is interesting because it proposes a solution to the issue of privacy and data gathering for training. Such models could pave a way to generate training data without an invasion on privacy. These models have become very popular in image generation[7][3][20]. A further look into this research area shows that these models have evolved past simple generation of new data. Image to image translation[10], human pose generation[14], face aging generation[2], super resolution[13] are some new applications of these models.

Music generation is a growing area in this field of generative models. Many attempts at music generation have been done in symbolic domain, such as MIDI sequences[6][4][19][11]. MIDI files are a sequence of notes that are to be played in the order they appear. In converting a raw music to MIDI we have abstracted away a lot of information about the physical process that produces the sound. For example, in a piano, note C can be played in infinitely many ways depending on how hard or fast the key is struck but in a MIDI sequence it just appears as C. This sort of abstraction makes the modelling problem more tractable and allows us to use simpler models to tackle the problem. But this method is restrictive in the sense that we lose diversity in the outputs we can represent by the model because we have confined the outputs of the model to a small set.

A raw waveform, on the other hand, contains rich information about the processes that produced the music. This allows the models trained on these data to produce a wide variety of outputs. It allows them to capture the character in the music which is unique to people who played it. But this also makes the modelling problem much more difficult. This is most of the work done in music generation have been in the symbolic domain and far fewer successful works have been published in the waveform domain. Some of the works that have had success using waveform are[15] which use generative models, likelihood based models or adversarial models. So, this area is ripe for exploration and has a lot of opportunities to learn about cutting edge models on generative problems and time series analysis and this is what got us excited to pursue this area for our project.

Submitted to 34th Conference on Neural Information Processing Systems (NeurIPS 2020). Do not distribute.

For our COMP137 Deep Neural Networks Final Project, we decided to look into the Lo-Fi music generation. We were initially drawn to this specific genre because it reduced the complexity of our problem, is very rhythmic and repetitive in nature, and is a popular genre of study music for students.

2 Our Approach

Our approach to tackling this problem differs slightly from the ones in the papers cited above. Our approach is similar language model trained on Shakespeare's text. When given a few sentences of input, the model produces paragraphs that sound like Shakespeare. We train our model just like a time series prediction task on a data set containing several songs but of same genre. In other words, during training we give the model a fixed length sequence and have it predict the next few time steps. Then we compare the predictions against actual values to make the gradient updates. After training, to generate music, we pass in a small sample (or noise) to the model which produces the next sequence as its prediction. We would then feed back the predictions to produce next predictions and continue this process to get arbitrarily long output music samples. This idea is clearly illustrated in figure (1).

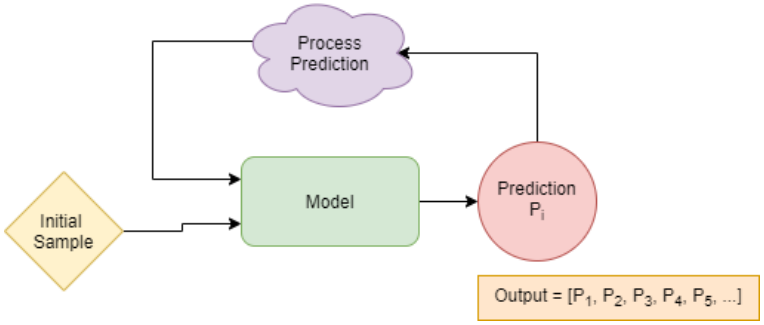

Figure 1: Our approach to music generation

We investigated three different ways to tackle the task of music generation. One was using baseline LSTM for time series prediction. The second was an LSTM Encoder Decoder with the spectrogram data. And lastly, a Temporal Convolutional Network.

3 Data

3.1 MIDI vs Waveform

As discussed earlier, many of the attempts at Lo-Fi music generation found online used MIDI files as their representation for the songs, a file that stores the specific notes and instruments of a song, similar to sheet music. However, the approach we decided to go with was using waveform data, which measures the displacement of air molecules over time caused by the playing of music.

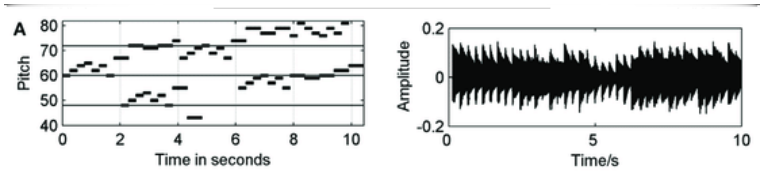

Figure 2: MIDI vs Waveform

Modelling waveform data was a significantly more complex task due to the number of samples per second, and the range of values at each sample. In MIDI data, songs typically play in the range of $4 - 32$ notes per a second with around 100 different values for each note. However, in waveform data, a sample rate is typically $44,100$ samples/second, with each sample having a value in the range $[-32768, 32767]$.

This added complexity, however, comes with greater potential. MIDI files represent the notes and instruments, not the actual audio data. Thus, MIDI files must be passed through software in which the computer will estimate what the song sounds like. Thus, converting music into MIDI files and then back into music is not a lossless process and results in a lot of lost information due to the very simplified, but powerful, MIDI file.

3.2 Data Collection

We collected our own data since we wanted more control over where we were sampling from and how we would process our data. Our data was collected from select Lo-Fi albums on Spotify using the Spotipy Python library [17]. Using this library, we were able to get 30 second samples from each of the songs in our selected albums, These 30 second samples were downloaded as mp3 files, and then converted to wav files using ffmpeg-python.

3.3 Data Pre-processing

We first split the songs into fixed length input and output sequences which we used for training/testing. We wanted to split our task of modelling entire Lo-Fi music into multiple sub problems, which prompted us to try to isolate different instruments in our music. A key characteristic of Lo-Fi music is the rhythmic drum beat in the background in addition to a melody. Using Spleeter by Deezer, we are able to separate our music into 5 different channels: bass, drums, piano, vocals, other [5]. We noticed that almost all of the music was split into the drums, piano, and other. This gave us two types of data sets of split sequences: one with all instruments and the other with individual instruments isolated. We experimented with both data sets to train the different models we investigated.

In addition to splitting up our music into different channels, we down sampled from 44.1k to 4.41k. When using waveform data to represent our audio data, we have to store our continuous waveform function as discrete data-points. The industry standard, 44.1k is a great way to approximate the continuous waveform function due to its high-frequency sampling rate.

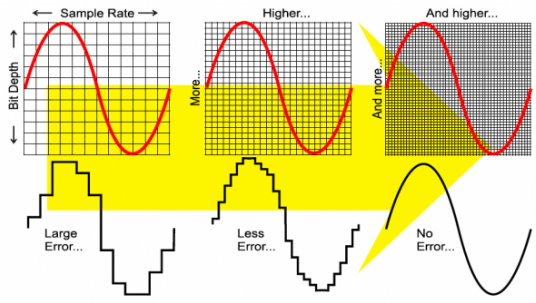

Figure 3: Effect of Increasing Sample Rate

The above figure shows the effect of down-sampling our music to 4.41k. We approximate our waveform function with slightly more loss, however we are able to pass in 10 times less data into our models, significantly increasing processing speed. This down-sampling has its biggest impact on high frequency (high pitch noise) since it requires higher sample rates to accurately capture the shape of the waveform.

4 Baseline Model

Shown below, is one of the initial architectures that we attempted to use. It is straightforward LSTM that takes in 500 different time-steps, each of which is 1 sample long. The reason why we decided to go with a many to one architecture was because we believed that if we were able to train such a model with a high enough degree of accuracy, we would be able to convert our predictions into input, allowing our prediction to continue through time.

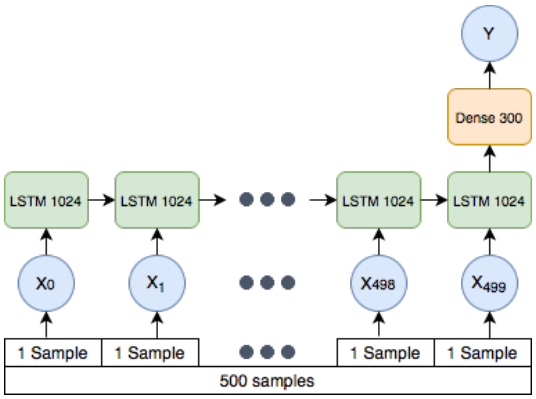

Figure 4: Baseline LSTM Architecture

A few issues arose with this architecture. The first is that back-propagation through time takes significantly longer than we would like. With limited computation power and time, we were not able to get our loss low enough during training.

The second issue that arose was that the model didn't generalize well. The model tended to predict either a constant value, leading to a horizontal line in our waveform data. Int he scenarios in which the model did predict non-linear wave-forms, they always would produce static sound. The two models that follow have added complexity in their architecture/pre-processing to correct for these mistakes.

5 ST-FT Encoder Decoder Model

5.1 Encoder Decoder Overview

The first of our two models is a Short Time Fourier-Transform Encoder Decoder sequential model. The model uses 10,000 samples of input (2.25 seconds) to predict 10,000 samples of output.

The encoder is a many-to-one LSTM with a hidden layer size of 2048 and input shape of (20, 1000), where each of the 20 time-steps represents 500 samples (0.11 seconds). Our decoder is a one-to-many LSTM with a hidden layer size of 2048, and output shape of (20,1000).

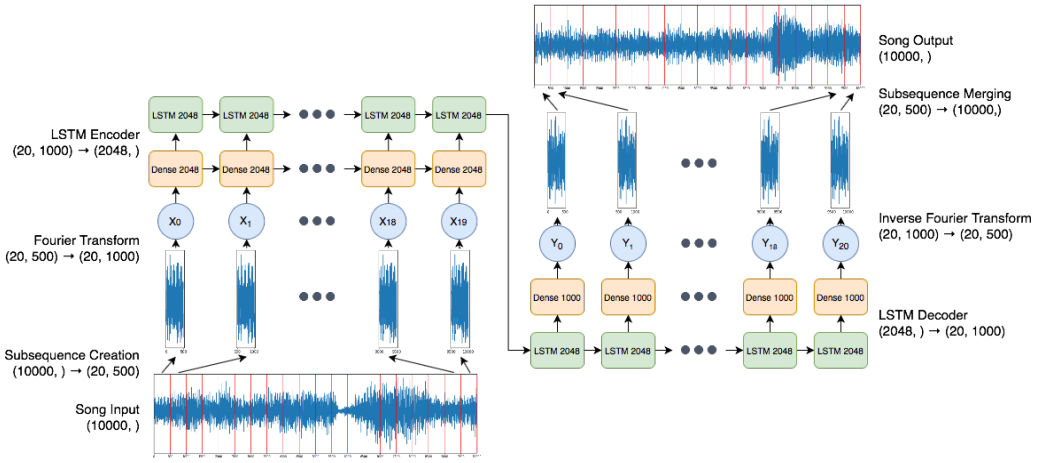

Figure 5: ST-FT Encoder Decoder Model Architecture

5.2 Time Domain to Frequency Domain

This section will describe how we go from 10,000 samples of input and obtain an input shape of (20, 1000) for our encoder LSTM. We first split up our 10,000 samples into 20 non-overlapping sub-sequences, each with 500 samples.

Given one of the sequences of 500 samples, our discrete waveform function can be described as $A_j(k)$ where $j \in [0, 20)$ and $k \in [0, 500)$. It can be thought of as j being our sub-sequence index, and k being the index within that sub-sequence.

Given one of our sub-sequences, $A_j(k)$, of input of length N = 500 samples, we can calculate our Discrete Fourier Transform[9], $X_j(t)$

$$X_j(t) = \sum_{k=0}^{N-1} A_j(k) \cdot W^{tk} \quad t = 0, 1, \cdots N-1$$

where W is the principal Nth root of unity.

$$W = e^{\frac{2\pi i}{N}}$$

Thus, each of our Discrete Fourier Transforms, $X_j(t)$, is represented as a sequence of 500 complex numbers.

$$X_j(t) = a_j(t) + b_j(t) \times i$$

Each of our sub-sequences, $A_j(k)$, are represented as two sequences of length 500 (a sequence of real coefficients and a sequence of imaginary coefficients). For simplicity, these two sequences are concatenated, giving us a sequence of 1000 values for each of the 20 sub-sequences, and a final shape of $(20, 1000)$ after our ST-FT. [16]

5.3 Encoder Decoder Architecture

Our LSTM encoder encodes the input of $(20, 1000)$ into a single 2048 long vector. This 2048 long vector is then used to generate the $(20, 1000)$ output vector, which can be transformed and reshaped into a 10000 long sample (2.25 seconds).

6 Temporal Convolutional Network Model(TCN)

6.1 TCN Motivation

Modern RNNs work great for time series analysis. But in our case, we have long input sequences and we found that RNNs are very slow to train. This is because of the sequential calculation of gradient for each input series. Intuitively, RNNs can be thought of as fully connected layers that have as many hidden layers as there are time steps in the input sequence of the RNNs. So, a 512 time step input to a RNN is similar to a fully connected network with 512 hidden layers in terms of gradient computation. This is going to make the gradient calculation very slow because for each input we have to forward propagate and back-propagate through 512 layers. An elegant solution to this problem is to use Temporal Convolutional Networks(TCN) [12].

TCNs use 1D convolutional kernels to extract features from series data. We know, from Vison models, that convolutional kernels are great feature extractors. They also work great for series data in terms of feature extraction. Because of inherent parallel structure in convolutions these networks can be made significantly faster to run/train. Since we slide the kernel across the sequence from left to right, we do not lose the sequential information in the time series. Literature shows that TCNs consistently match or outperform regular sequence models[18] like LSTM, GRU while using less number of parameters and without much tuning. This is why we decided to include TCN model for our times series modelling.

6.2 TCN Architecture Overview

A simple TCN layer consists of stacked 1D convolution layers just like regular Convolutional Neural Networks(CNNs). Figure [ref figure] shows the basic details about the TCN architecture. At the botton we have a simple 1D convolutional layer with a associated kernel. In the figure the kernel size is two. The hidden layer above the base/input layer is called the first dilation layer. In this layer the the original kernel of size two is dilated by a factor of two and then applied to the same input sequence. Similarly, the second hidden layer the kernel is further dilated by a factor of two and then applied to the input sequence and so on. At the end we have an output layer which can then be connected to other layers like Dense layers. The dilation factor is usually chosen to be powers to two. Due to dilation, the kernel is sparse and has the same number of parameters as the kernel in the input layer. This property is clearly illustrated in figure (6) which shows the dilation for a 2D convolutional kernel.

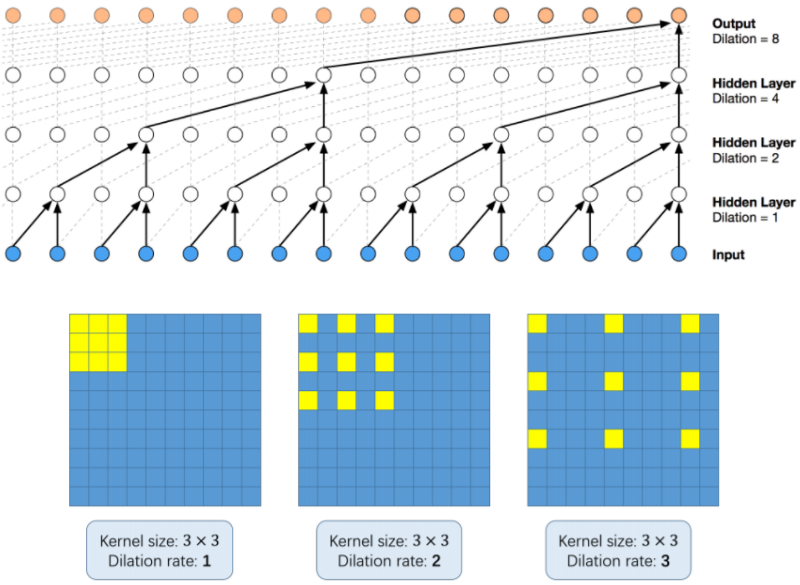

Figure 6: (Top)[18] TCN with dilation=[1,2,4,8], kernel size of 2 and only one stack. (Bottom) Visualization of dilated convolution in case of 2D convolution

6.3 TCN Properties: Pros and Cons

TCNs have many of the desirable properties of RNNs. They have memory. Memory in temporal convolutional network is generalized as its receptive field. Receptive field measures from how far back the information can affect the result at current time. TCNs memory directly depends on the length of the kernel. But increasing the length of kernel increases the number of parameters linearly with the length and if we want long term memory then our model will become very big. To get around this fact we can use dilation layers. Dilation layer helps increase effective memory of the network but has the same number of parameters as the un-dialated kernel. This means by adding suitable number of dilation layers with appropriate dilation factors we can get a network with the desired amount of memory without causing the network to be too big. We can also stack TCN layers as shown in figure (7). Stacking increases the receptive field greatly. In fact the increase is by a factor of the number of stacks we have. The relation between kernel size(K_{size}), number of stacks(N_{stack}) and the dilation factors at i^{th} layer (d_i) is captured in a nice mathematical expression as follows:

$$R_{field} = 1 + 2(K_{size} - 1)N_{stack} \sum_i d_i \tag{1}$$

TCN do not have the problem of exploding or vanishing gradients as long as the number of dilation layers are kept within a reasonable size. Just like ReNet, we can add skip connections between dilation layers to mitigate this issue greatly. Another, advantage of TCN is that it lends itself to a nice

parallel structure due to the convolution operation which means that training with these networks is much faster compared to LSTM/GRU. One big drawback of TCN is that, unlike LSTM/GRU, it cannot handle variable length inputs.

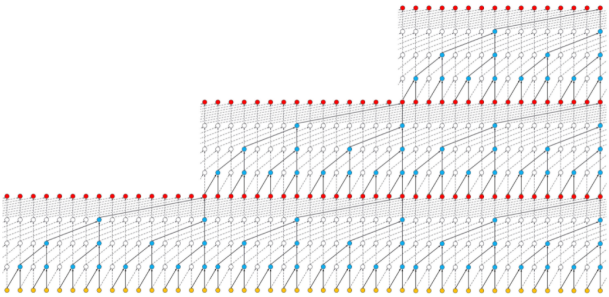

Figure 7: Temporal convolution with 3 stacks and kernel size of 2

6.4 TCN Specific Training Remarks

This speed up in training and improved performance led us to consider using TCN for our song generation task. Thankfully for us, there is a package (by Phillippe Remy[1]) which we can install in Keras that allows us to use TCN as a keras layer. We still had to tune several parameters like kernel size, number of dilations, input length to get the model to train well. Here are some of the things we learned while training a TCN model for the song prediction task. Using longer kernels did not really help as it made the model very big especially when we used many kernels per layer. The training would become very slow due to this which we wanted to avoid. Using many kernels was generally beneficial but we had to be careful about severe over fitting and bigger model size. Dilation was set so that the receptive field was about as long as the input sequence. Dilation was given as a list of numbers where each number corresponded to the dilation factor in that layer. It is better to use dilation to increase the receptive field than to use bigger kernels. We wanted to use long input sequences as the more information we can give the model the better the prediction is going to be. But increasing the length of input sequences also increases the memory requirement of the computer and we run the risk of running out of memory. From our tuning experience we found that for an input size of 512, kernel size=2 to 8, and an appropriate dilation=[1,2,4,8,16,32] worked the best. The general architecture used is given in figure (8)

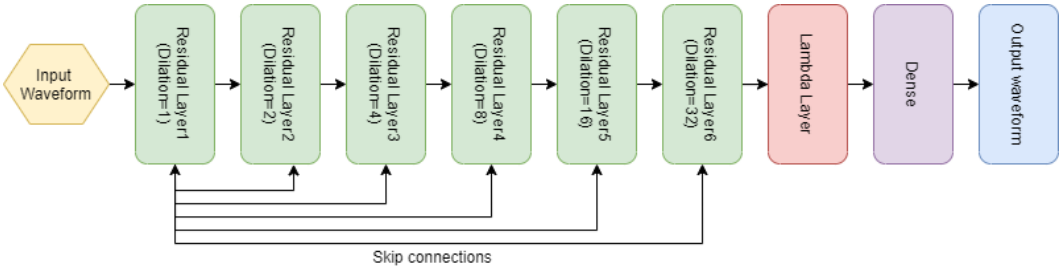

Figure 8: TCN architecture used for training

7 General Training

For TCN model, the TCN specific training parameters were chosen according to the discussion in the TCN training remarks section. The optimizer used was SGD with learning rate of 0.01. The error metric used was $Huber$ or $LogCosh$ because these were resistant to occasional wrong prediction as opposed to Mean Squared Error which is very sensitive to wrong predictions. In training, we chose the scheme where we train with small batch size(for e.g 32) at the beginning for a few hundred epochs and then gradually increase the batch size and train further. This approach has been studied and there is evidence that it is effective at finding stable and wide local optimums which lead to better

generalization [8]. We found this to be true in our case. We also found that $tanh$ activation in TCN layer worked better for our data set.

For ST-FT model, an ADAM optimizer was used instead. Additionally, rather than varying batch size, learning rate was varied instead. Both the drums and melody models were trained for 200 epochs at learning rates of 0.001 and 0.0005.

8 Results

As discussed earlier, the output music was generated by feeding back the predictions into the model. It is hard to quantify how good the predictions are without hearing the samples. We've included some of the predicted wave-forms and the audio samples are in our Github repository. Some of the outputs of the different models are shown below.

Here is an example of the TCN model output for an initial random noise input. The model outputs noise for the first 65000 iterations and then starts to produce melodies.

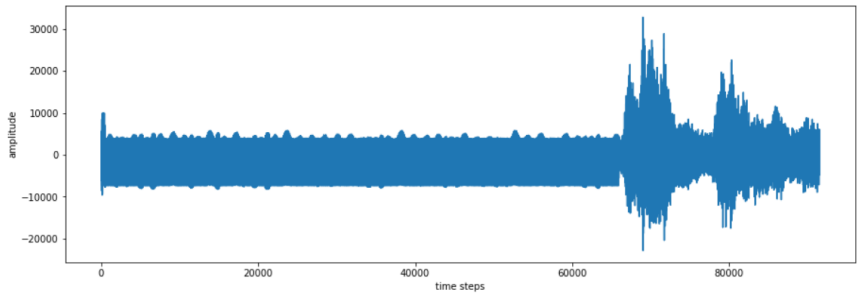

Figure 9: TCN output to initial random noise input

In the STFT model, the predictions tend to be very repetitive. This is likely because not much

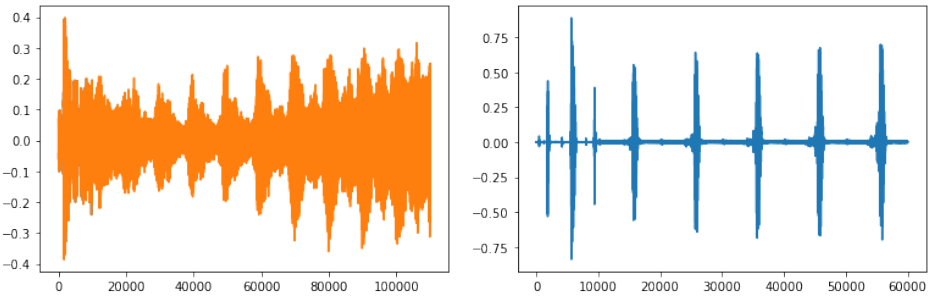

Figure 10: (Left)Melody Prediction. (Right)Drum Prediction

information about long term structure can be encoded in the 2048 dimensional vector. In the above figure, it is clear where the predictions begin with the large spikes.

When predicting drum wave-forms, there is the same issue with repetitiveness. The generated spikes are evenly distributed. Additionally, the spikes are slightly wider than the original spikes shown in the $[0, 1000]$ range. This results in a more muffled drum sounding beat.

9 Discussions

Because we were using raw waveform, the outputs of the models were not, in general, clean. They usually had high frequency hissing which was expected as the models do not know anything about music and are just generating data based on what happened before. Another things we noticed was that, the output music did not have sustained melody. There would be moments where we could hear

some patterns and then a completely different sounding patterns would emerge. Because of this the output did not sound good as a whole but it had small chunks that were good.

The TCN model had good performance in the full instrument data but had a poor performance in the isolated instrument data. For full instrument data, TCN model could produce musical patterns from just a noise input. But as was the case with all the models, we could not get it to produce sustained melodies. In the case of isolated instruments, we have an idea why this model did not perform so well. This is because in the isolated instrument data, we have hundreds of samples of silence(which is basically background noise) in between the instruments(for instance drums). This means that we would need the model to have very long memory to keep track of the instruments playing and the noise in between. To get such long memory, we had to make our TCN model very big which we could not do because of computer memory and training time limitations. But we believe that given the appropriate resources to make the memory long enough, this model will work just fine in the isolated instrument data set.

The ST-FT model did well at generating continuous and sinusoidal waveforms, an issue that we ran into in our baseline model. In our baseline model, there would be jumps at different time-steps that would not otherwise be possible in real audio. To limit these jump discontinuities, we realized that predicting the frequencies of the output rather than the amplitude at different times would give us a much more sinusoidal and continuous shape.

Throughout this project, we unfortunately realized we did not have enough computation power or resources. We could improve our model by increasing the hidden layer size from 2048 to 4096 or 8192, increase the amount of training data, and increase the input length. A big problem that this model has is that it although it sounds like music, it can get very repetitive. These improvements would increase the model's memory as well as its ability to generalize to new test data.

All in all, we hope that the insights we have in this paper about tackling waveform prediction problem will prove helpful to anyone starting out in this field and facilitate their research.

References

[1] https://github.com/philipperemy/keras-tcn.

[2] Grigory Antipov, Moez Baccouche, and Jean-Luc Dugelay. "Face aging with conditional generative adversarial networks". In: *2017 IEEE International Conference on Image Processing (ICIP)*. 2017, pp. 2089–2093. DOI: 10.1109/ICIP.2017.8296650.

[3] Andrew Brock, Jeff Donahue, and Karen Simonyan. "Large scale GAN training for high fidelity natural image synthesis". In: *arXiv preprint arXiv:1809.11096* (2018).

[4] Gino Brunner et al. "MIDI-VAE: Modeling dynamics and instrumentation of music with applications to style transfer". In: *arXiv preprint arXiv:1809.07600* (2018).

[5] Deezer. *deezer/spleeter*. URL: https://github.com/deezer/spleeter.

[6] Hao-Wen Dong et al. "Musegan: Multi-track sequential generative adversarial networks for symbolic music generation and accompaniment". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1. 2018.

[7] Ian J Goodfellow et al. "Generative adversarial networks". In: *arXiv preprint arXiv:1406.2661* (2014).

[8] Mert Gurbuzbalaban, Umut Simsekli, and Lingjiong Zhu. *The Heavy-Tail Phenomenon in SGD*. 2021. arXiv: 2006.04740 [math.OC].

[9] Paul Heckbert. *Fourier Transforms and the Fast Fourier Transform (FFT) Algorithm*. Jan. 1998. URL: http://www.cs.cmu.edu/afs/andrew/scs/cs/15-463/2001/pub/www/notes/fourier/fourier.pdf.

[10] Phillip Isola et al. *Image-to-Image Translation with Conditional Adversarial Networks*. 2018. arXiv: 1611.07004 [cs.CV].

[11] Vasanth Kalingeri and Srikanth Grandhe. "Music generation with deep learning". In: *arXiv preprint arXiv:1612.04928* (2016).

[12] Colin Lea et al. "Temporal convolutional networks: A unified approach to action segmentation". In: *European Conference on Computer Vision*. Springer. 2016, pp. 47–54.

[13] Christian Ledig et al. *Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network*. 2017. arXiv: 1609.04802 [cs.CV].

[14] Liqian Ma et al. *Pose Guided Person Image Generation*. 2018. arXiv: 1705.09368 [cs.CV].

[15] Aaron van den Oord et al. "Wavenet: A generative model for raw audio". In: *arXiv preprint arXiv:1609.03499* (2016).

[16] Unnati-Xyz. *unnati-xyz/music-generation*. URL: https://github.com/unnati-xyz/music-generation.

[17] *Welcome to Spotipy!¶*. URL: https://spotipy.readthedocs.io/en/2.18.0/.

[18] Jining Yan et al. "Temporal convolutional networks for the advance prediction of ENSO". In: *Scientific reports* 10.1 (2020), pp. 1–15.

[19] Li-Chia Yang, Szu-Yu Chou, and Yi-Hsuan Yang. "MidiNet: A convolutional generative adversarial network for symbolic-domain music generation". In: *arXiv preprint arXiv:1703.10847* (2017).

[20] Jun-Yan Zhu et al. "Generative visual manipulation on the natural image manifold". In: *European conference on computer vision*. Springer. 2016, pp. 597–613.