Image-Based Music Generation using Invertible MIDI-to-Image Conversions

Diana Voevodsky
Mathematics
dinushin@stanford.edu

Andrew Hojel
Computer Science
ahojel@stanford.edu

Oscar O'Rahilly
Computer Science
oscarfco@stanford.edu

Abstract

There have been many attempts at generating music using various deep generative models. Although some of these attempts have been relatively successful, image generation models significantly outperform music generation models in their respective tasks. We leverage these impressive advances in image generation to achieve higher quality music generation. Using a dataset consisting of classical musical instrument digital interface (MIDI) files, we map each file to a visual representation, and feed the resulting images to StyleGAN2 [7], WassersteinGAN [1], and PixelCNN++ [12]. The given model then generates new images based on the learned distribution of the MIDI image representations. Finally, we convert the generated images back to their musical MIDI form. We find that PixelCNN++ significantly outperforms the other models in the Melody, Harmony, and Rhythm of the resulting MIDI files. We also find that this technique outperforms traditional music generation models such as Music Transformer [4].

1 Introduction

The goal of our project is to leverage high-performing unconditional image generation models to generate music from image representations of musical instrument digital interface (MIDI) files. Conventionally, music generation models take audio files as inputs and use their contents to generate new, original audio files. This approach, however, has not proven to be as successful as one might hope. The human ear is still often able to distinguish a computer-generated song from a human-written one, even with the best performing models. On the other hand, image GANs have recently become powerful enough to deceive the human eye. We decided to capitalize on these advances and apply them to music generation tasks.

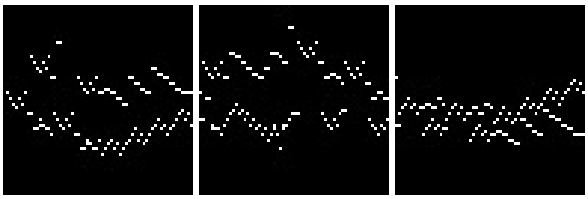

Figure 1: First three images depicting the beginning of *Bach's Invention No. 1*

We take a MIDI file as input and map to its image representation. We then feed this representation into an image generation model (see technical approach for details), which generates new images that hold the general format of a MIDI file's visual representation. Finally, we convert this image back into its original MIDI format, and play the generated sound.

35th Conference on Neural Information Processing Systems (NeurIPS 2021).

2 Related Works

Numerous machine learning tasks have leveraged visual representations of audio files in the training of their audio classification models. In a recent paper published in Nature, Gupta et al were able to use recurrent convolutional neural networks, trained on visual spectrograms of bird calls, to classify particular species [3]. The task of music generation, on the other hand, has not leveraged visual representations of audio. Instead they rely on raw audio or other forms of audio files like MIDI as seen in Deepmind's Wavenet [11] or Open AI's Jukebox [10]. Although Yang et al have attempted using a General Adversarial Network (GAN) [15], rather than the conventionally used Recurrent Neural Network (RNN), on raw MIDI files, this technique is still quite different from our approach. We use state of the art image generation models – in particular, StyleGAN2 [7], WassersteinGAN [1], and PixelCNN++ [12] – to generate new visual representations of MIDIs and produce results that can rival the state of the art generative models.

3 Approach

In order to convert MIDI files to visual representation (and visa versa), we wrote a script which takes a MIDI file and outputs a 64x64 image representation as shown in Figure 1. We based our script off the code in [9].

3.1 Dataset

There were a number of different consideration we had to take into account when deciding upon a dataset for this specific project. Music can vary greatly in rhythm, harmony and complexity depending on a number of factors such as genre or instrumentation. As an example, more contemporary musical genres like Jazz are often far less structurally consistent in harmony and rhythm than classical music. Furthermore, there is no one instrument that music is written for, and music written for one instrument can sound very different to music written for another (e.g. Piano vs Drums). In order to fix a number of these variables, we focused our dataset to only include western classical Piano music. There are a number of reasons we focused on this specific choice of genre and instrumentation. Firstly, early classical/Baroque music has very rigid rules regarding harmony and rhythm, which provide a good benchmark to measure our models' representational abilities. Secondly, there is a wide availability of western classical piano music online, which helps in amassing larger numbers of MIDI files. Finally, a lot of the literature surrounding generative music models also use western classical piano music within their training sets, and so performing comparisons between other model's outputs and ours is more fair.

Aside from the type of music present in our dataset, the other aspect we had to decide upon was how we would represent our audio visually. Work conducted by Gupta et al. [3] and Fakhry et al. [2] perform audio classification on images of mel spectrograms, a particularly rich visual representation of audio (shown in Figure 2). Unfortunately, however, the transformation from audio to mel spectogram is not invertible and so in converting a generated mel spectograms back to audio we would lose information. Visual representations of MIDI images, however, whilst not as visually rich as mel spectograms, are fully invertible.

For our baseline dataset, we collected over 200 different MIDI files of piano music written by four composers: Bach, Mozart, Haydn and Beethoven. These 200 MIDI files were then converted to 4040 64x64 MIDI images. In order to increase our dataset, instead of continuing to manually download MIDI files that fit our criteria, we used a pre-existing constructed dataset titled GiantMIDI-Piano from [8], containing 10,854 unique piano solo pieces composed by 2,786 composers. We pulled 60,000 images from GiantMIDI-Piano to create our larger dataset.

An important trade-off in using visual representations of audio over the raw audio itself is that in mapping the audio to a visual format, we are discretizing our data. Audio, of course, is in a continuous space and so by converting it to a visual format we inherently lose information present in the original audio. As an example, if we take each column of our MIDI image to be a 8th note, there are certain rhythmic structures that are impossible for our images to represent (i.e. triplets). On the other hand, if we make the discretization even finer, say 128th notes, the amount of structural information we are able to encode in a dataset of the same number of images decreases as each image depicts a smaller amount of music.

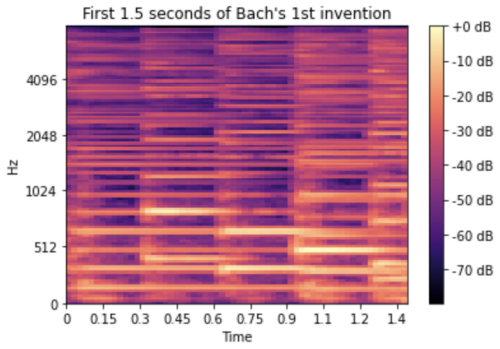

Figure 2: Mel Spectogram depicting the first 1.5 seconds of *Bach's Invention No. 1*

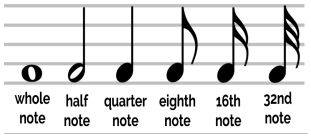

Figure 3: Note values to demonstrate different levels of audio discretization

To balance this trade off, we settled on using a column width of 16th notes. While this does put limitations on certain, more complex rhythms, these notes are small enough where close approximation to these structures are still possible. It also allows us to a substantial amount of audio songs in a more compact number of images, which we thought was very important.

3.2 Technical Approach

In order to generate new images, we chose three generative models and compared their performance.

StyleGAN2 (Baseline). StyleGAN2 [7] improves upon StyleGAN [6], which is built based on the GAN structure – the push and pull between a generator and a discriminator. They compete as the generator tries to produce as realistic data as possible while the discriminator works on recognizing synthesized data from real data. In particular, a generator $G(.)$ learns to map a latent variable z to a realistic image, while a discriminator $D(.)$ aims at separating real images x from synthesized ones $G(z)$. These two networks compete with each other and are jointly optimized with

$$\mathcal{L}_D = -\mathbb{E}_{x \in X}\left[\log(D(x))\right] - \mathbb{E}_{z \in Z}[\log(1 - D(G(z)))].$$

$$\mathcal{L}_G = -\mathbb{E}_{z \in Z}[log(D(G(z)))]$$

where Z and X denote the pre-defined latent distribution and real data distribution respectively. StyleGAN constructs an alternative generator architecture which leads to an automatically learned, unsupervised separation of high-level attributes. StyleGAN2 improves on StyleGAN by redesigning the generator's normalization as well as regularizing the generator to encourage good conditioning in the mapping from the latent codes to the images.

The StyleGAN architecture includes a style block which consists of modulation, convolution, and normalization. The modulation scales each input feature map on the convolution based on the incoming style. In StyleGAN2, this is alternatively implemented by scaling the convolution weights:

$$w'_{ijk} = s_i \cdot w_{ijk},$$

where w and w' are the original and modulated weights, respectively, s_i is the scale corresponding to the ith input feature map, and j adn k enumerate the output feature maps and spacial footprint of the convolution, respectively. The purpose of instance normalization is to essentially remove the effect of s from the statistics of the convolution's output feature maps. This can be achieved by scaling each output feature map j by $1/\sigma_j$, where σ_j is the modulation and convolution output activations'

standard deviation. Alternatively, in StyleGAN2, this is baked into the convoltuion weights:

$$w''_{ijk} = \frac{w'_{ijk}}{\sqrt{\sum_{i,k} w'^2_{ijk} + \epsilon}}.$$

In addition to this redesign of the generator's normalization, StyleGAN2 also regularizes the generator. The goal here is to encourage that a fixed-size step in $\mathcal{W}$ results in a non-zero, fixed magnitude change in the image. This can be measured empirically by stepping into random directions in the image space and observing corresponding $\mathbf{w}$ gradients. At a single $\mathbf{w} \in \mathcal{W}$, the local metric scaling properties of the generator mapping $g(\mathbf{w}) : \mathcal{W} \rightarrow \mathcal{Y}$ are captured by the Jacobian matrix $\mathbf{J}_\mathbf{w} = \partial g(\mathbf{w}) / \partial \mathbf{w}$. In order to preserve the expected lengths of vectors regardless of the direction, StyleGAN2 therefore formulates the regularizer as

$$\mathbb{E}_{\mathbf{w},\mathbf{y}\sim\mathcal{N}(0,\mathbf{I})} \left(||\mathbf{J}^T_\mathbf{w}\mathbf{y}||_2 - a \right)^2,$$

where $\mathbf{y}$ are random images with normally distributed pixel intensities and $\mathbf{w} \sim f(z)$, where $\mathbf{z}$ are normally distributed.

WassersteinGAN. WassersteinGAN [1], like StyleGAN2, is based on the conventional GAN structure. Its main contribution is proposing the use of the Earth Mover (EM) distance, rather than the conventionally used Kullback-Leibler (KL) divergence in order to measure the similarity between the model distribution and the real distribution. The purpose of this change is to mitigate mode collapse, which is a problem commonly experienced by GANs including StyleGAN2, and one that we experienced in our baseline results, which is why we chose WassersteinGAN as our next experiment.

The EM distance is defined as

$$\mathrm{W}(\mathbb{P}_r, \mathbb{P}_g) = \inf_{\gamma \in \Pi(\mathbb{P}_r, \mathbb{P}_g)} \mathbb{E}_{(x,y)\sim\gamma} \left[||x - y|| \right],$$

where $\Pi(\mathbb{P}_r, \mathbb{P}_g)$ denotes the set of all join distributions $\gamma(x, y)$ whose marginals are $\mathbb{P}_r$ and $\mathbb{P}_g$ respectively. Intuitively, $\gamma(x, y)$ indicates how much "mass" must be transported from x to y in order to transform the distribution $\mathbb{P}_r$ into the distribution $\mathbb{P}_g$. The EM distance is then the "cost" of the optimal transport plan.

In order to train a model using the EM distance in practice, the WassersteinGAN uses the following approximation:

$$\mathrm{W}(\mathbb{P}_r, \mathbb{P}_\theta) = \sup_{||f||_L \leq 1} \mathbb{E}_{x\sim\mathbb{P}_r}[f(x)] - \mathbb{E}_{x\sim\mathbb{P}_\theta}[f(x)]$$

where the supremum is over all the 1-Lipschitz functions $f : \mathcal{X} \rightarrow \mathbb{R}$. In other words, the model attempts to solve the problem

$$\max_{||f||_L \leq 1} \mathbb{E}_{x\sim\mathbb{P}_r}[f(x)] - \mathbb{E}_{x\sim\mathbb{P}_\theta}[f(x)]$$

using

$$\Delta_\theta \mathrm{W}(\mathbb{P}_r, \mathbb{P}_\theta) = -\mathbb{E}_{z\sim p(z)}[\Delta_\theta f(g_\theta(z))].$$

PixelCNN++. The original PixelCNN developed by van den Oord et al [13], proposed a masked convolution architecture that allows a convolutional neural network (CNN) to generate images, while maintaining auto-regressive properties to ensure that each pixel is only conditioned on pixels before it (all pixels above and to the left of the current pixel).

We decided to use PixelCNN++ instead of PixelCNN for various reasons that directly relate to our problem formulation. PixelCNN++, proposed by Salimans et al [12], provides two primary benefits when applied to this problem: the use of discretized logistic mixture likelihood and using downsampling instead of dilated convolutions.

The use of discretized logistic mixture likelihood replaces the 256-way softmax used to model the conditional distribution of a sub-pixel or color channel of a pixel in PixelCNN. This change was made to signficantly decrease the amount of memory usage and to account for sparse gradients early in training with respect to the network parameters. The discretized logistic mixture model assumes there is a latent color intensity v with a continuous representation that is measured to the nearest 8-bit representation for an observed pixel. The continuous univariate distribution is modeled as a mixture of logistic distributions, which allows for easy calculation of the probability of an observed pixel

value x. This strategy applies particularly well to our problem formulation because pixels only take on a binary value of 0 or 255 (either a note or not a note). Therefore, it would be very redundant to model the conditional distribution of a sub-pixel using a 256-softmax. In addition, this applies particularly well to our porblem because for the edge values of 0 and 255 naturally are given higher probability by the discretized logistic mixture because any probability mass outside of the [0,255] range is clipped. Therefore the probability distribution for all sub-pixel values x except 0 and 255 using K mixture components is as follows:

$$v \sim \sum_{i=1}^{K} \pi_i \text{logistic}(\mu_i, s_i) \tag{1}$$

$$P(x|\pi, \mu, s) = \sum_{i=1}^{K} \pi_i [\sigma((x + 0.5 - \mu_i)/s_i) - \sigma(x - 0.5 - \mu_i)/s_i)] \tag{2}$$

For the edge values, which receive more probability density, the distribution is modeled as follows (using equation (1) to model v):

$$P(x = 0|\pi, \mu, s) = \sum_{i=1}^{K} \pi_i [\sigma((x + 0.5 - \mu_i)/s_i) - \sigma(-\infty - \mu_i)/s_i)] \tag{3}$$

$$P(x = 255|\pi, \mu, s) = \sum_{i=1}^{K} \pi_i [\sigma((\infty - \mu_i)/s_i) - \sigma(x - 0.5 - \mu_i)/s_i)] \tag{4}$$

The second significant improvement over PixelCNN is the use of downsampling with short-cut connections instead of dilated convolutions. This reduces computation complexity because dilated convolutions use an input that increases as a result of zero-padding, whereas downsampling decreases the size of the input at each downsampling (the decrease is dependent on the stride). The downsampling can improve the modeling of long range structures because the original PixelCNN was forced to use convolutions with a small receptive field. The major downside of downsampling is that it loses information, but this is addressed by adding short-cut connections between the subsampling and upsampling layers as follows:

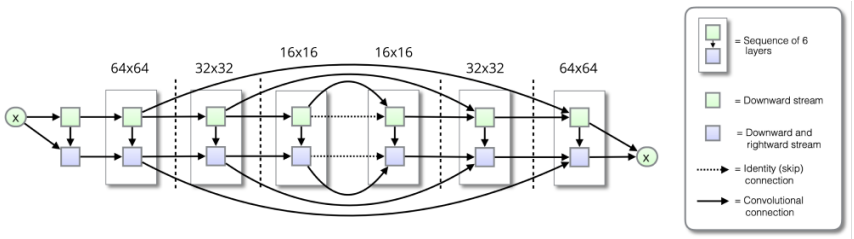

Figure 4: Demonstrates the short-cut connections in our model (takes an input of 64×64 and consists of 6 blocks and 5 ResNet layers)

4 Results

4.1 Baseline

We used StyleGAN2 as our baseline model. The model did not generate the desired results. Figure 5 depicts the series of images generated by StyleGAN2 after training for 0, 5, 10, and 50 ticks. The likely reason for this outcome is model's handling of the sparsity of the training images. The average percentage of white pixels across the 4040 training images is 3.7%. This is significantly less than other similar datasets such as MNIST which for reference, on average, has almost 4 times the number of white pixels (12.4%). This level of sparsity in our dataset causes the model to optimize its objective by generating 100% black pixels, a phenomenon commonly referred to as mode collapse.

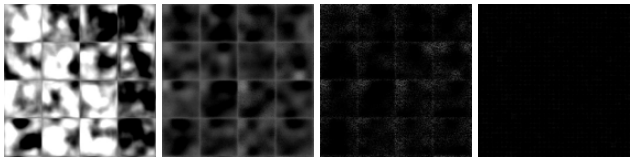

Figure 5: Images generated by StyleGAN2 after training for 0, 5, 10, and 50 ticks

4.2 Results

4.2.1 Metrics

Unfortunately, there is no objective way of evaluating audio samples produced by these generative models. This is because judging audio quality and determining how well a given audio file represents the underlying data distribution is a subjective task. When evaluating our audio samples, however, we proposed three metric to look at, namely: melody, harmony, and rhythm.

In order to have some way of quantitatively comparing samples, we derived score metrics based off of the three evaluation criteria discussed above.

1. **Melody:** The simplest test of whether a music has a clear melody is if a listener can succesfully identify a melodic line and hum along. The melody score is therefore a binary score with a value TRUE indicating a hummable melody and FALSE indicating an un-hummable melody
2. **Harmony:** Harmonic analysis is less clear cut. For this metric we score the harmony out of 5, with 5 indicating the piece contained mostly consonant chords and 1 indicating the piece contained mostly dissonant chords.
3. **Rhythm:** Our test for rhythm is whether or not the listener can clap along to the underlying rhytmic pattern of the piece. The rhythm score is also binary, with TRUE indicating a clappable rhythm and FALSE indicating an un-clappable rhythm

4.2.2 WassersteinGAN

In its original implementation, WassersteinGAN uses 3 pixel channels. Therefore, our first attempt at training WassersteinGAN with our small dataset resulted in generated images such as the ones shown in Figure 6.

Figure 6: Image generated by WassersteinGAN using 3 pixel channels

In order to eliminate the colored pixels, we altered the WassersteinGAN architecture slightly to allow it to operate with only 1 pixel channel. We then trained this version of the model on our large dataset. This resulted in the correct pixel coloring, but sub-optimal pixel patterns persisted, as shown in Figure 7.

4.2.3 PixelCNN++

The results from PixelCNN++ significantly outperformed our other models. We trained PixelCNN++ for 120 epochs on the small dataset and found that the later epoch samples both looked and sounded like piano samples that could have come from our underlying data distribution. Something we noticed in the training of our PixelCNN++ model is a steady increase in test loss after around 20 epochs, indicating that we were running into overfitting. We were, however, not too concerned about this due to the fact that we used a very high dropout rate ($p = 0.5$) and also because our dataset was quite

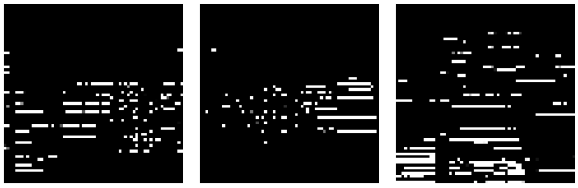

Figure 7: Image generated by WassersteinGAN using 1 pixel channel and large dataset

small (4040 images). Furthermore, in examining the outputs, we saw no indication of chunks of audio from the train set appearing in any of the generated samples. We discuss some of the generated samples at length in the analysis section.

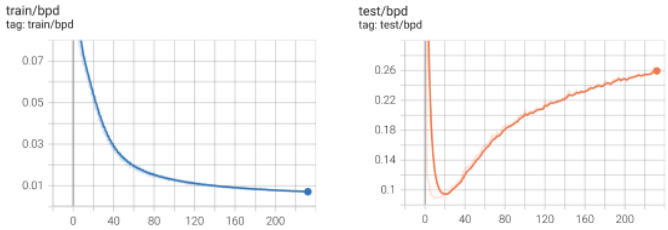

Figure 8: The training and test NLL expressed in bits-per-dimension

4.2.4 Big PixelCNN++

In addition to training PixelCNN++ on our small (4040 image) dataset, we also attempted to train it on our large (60,000 image) dataset. Unfortunately, each training epoch took roughly 24 hours to complete, and so we were only able to train the model up to 3 epochs. The results show that the model was nevertheless able to learn, and show that it has promising potential if given the time to train for more time.

4.3 Rotated PixelCNN++

The original implementation of PixelCNN++ uses a raster scan generation order, which means that every sub-pixel is generated conditioned on all the pixels above it and to the left of it. A visualization of the raster scan order can be seen in Figure 9:

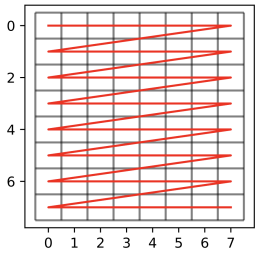

Figure 9: A visualization of raster scan generation order.

We realized that this may not be the most intuitive means of generating music given that it generates notes at a given pitch level then varies time (equivalent to writing all the notes at a certain pitch then decreasing pitch). Music is typically composed at each timestep, building chordal progressions in chronological order. This was the rationale behind rotating the images 90 degrees in the clockwise direction before training so that the raster order would generate the notes at each time step in the order of increasing pitch. Another rational for flipping the images was to be able to stick multiple MIDI images together by conditioning the generation of the next image given the bottom rows of the previous image. We were unable to train a model long enough to get high fidelity results, but

the images shown in Figure 10 demonstrate that the model was training successfuly on the curated / small dataset.

Figure 10: Samples generated by Rotated PixelCNN++ on the curated dataset at epoch 25.

Given the fact that the model did not train for enough epochs, we were unable to adequately evaluate the performance of this model and were unable to attempt generating multiple stitched images in succession. In addition, we would be excited to attempt generating music using different generation orders such as S-Curve or Hilbert Curve as seen in Jain et al. [5]

4.4 Analysis

4.4.1 WassersteinGAN

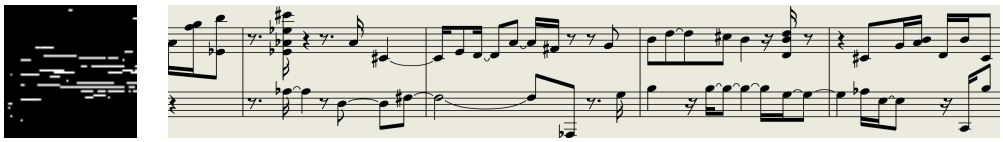

Figure 11: WGAN Example 1

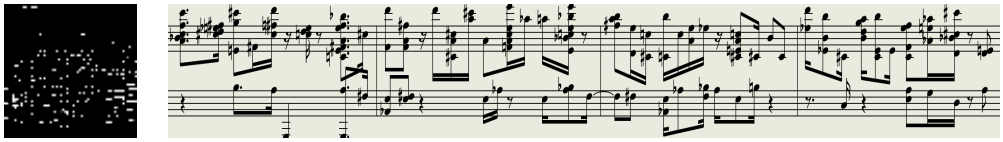

Figure 12: WGAN Example 2

	Melody	Harmony	Rhythm
WGAN (Example 1)	FALSE	2	FALSE
WGAN (Example 2)	FALSE	1	FALSE

Table 1: WGAN Examples Metrics

Unfortunately, the images generated by WassersteinGAN seem to not consistently resemble those from the training set. Both Examples 1 and 2 lack a melody that is easy to follow, do not have strong harmony (the chords are dissonant), and do not have a consistent rhythm. The most likely reason for this is that unlike models like PixelCNN++, the generative procedure in WGAN is not auto-regressive, meaning when generating samples it is not looking at previous notes or structures, rather it is simply creating an image which might fool the discriminator. This method may work for larger training sets; however, in this instance it did not produce passable results. Visually, though, the MIDIs do somewhat resemble the training set and so while they do not sound acceptable we believe that there is potential with this model architecture.

Link: `https://www.youtube.com/watch?v=J-_G1u2iWug`

4.4.2 PixelCNN++

Visually, these MIDI images looks very similar to examples shown above form the training set. Furthermore we can see clear descending dots (indicating a descending melodic line) and structured

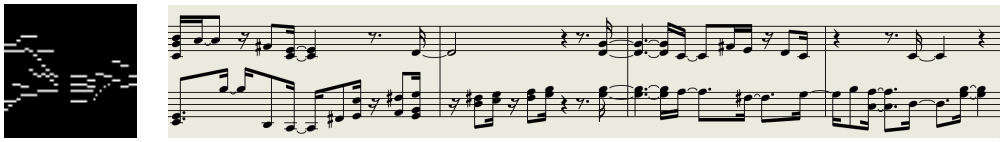

Figure 13: PixelCNN++ Example 1

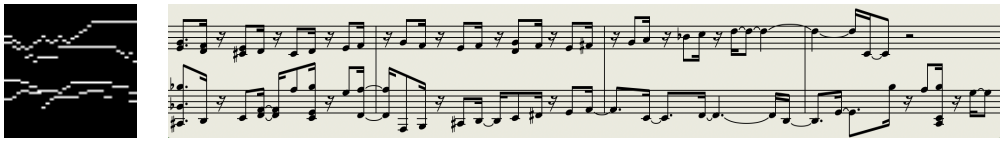

Figure 14: PixelCNN++ Example 2

	Melody	Harmony	Rhythm
PixelCNN++ (Example 1)	TRUE	5	TRUE
PixelCNN++ (Example 2)	TRUE	4	TRUE

Table 2: PixelCNN++ Examples Metrics

breaks between notes. Transforming these images back to MIDI files we can listen to the audio (follow the link below). The audio from these samples are extremely impressive. There is a clear and distinct melodic line: after listening to the samples once it is possible to roughly hum the melody. There is also a very clear harmonic structure. The melody is supported harmonically by consonant (not clashy) chords that have a clear progression throughout the clip. Finally, there is also a clear rhythmic structure within the clips. The notes do not appear randomly and even where there are breaks in audio (like in the middle of Example 1) they do not sound out of place. Towards the end of Example 1 we even here a syncopated rhythmic line, an extremely impressive rhythmic structure for our model to generate. Overall, these samples demonstrate the impressive ability of PixelCNN++ to generate visual MIDI images.

Link: `https://www.youtube.com/watch?v=Fqolqd4JM74`

4.4.3 Big PixelCNN++

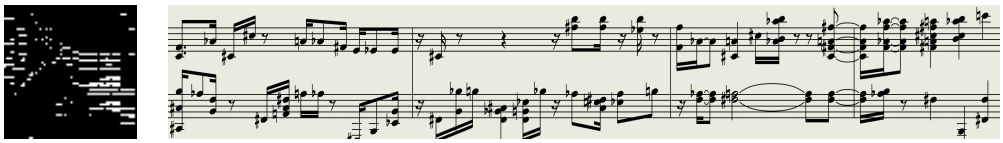

Figure 15: Big PixelCNN++ Example 1

	Melody	Harmony	Rhythm
Music Transformer (Example 1)	False	2.5	False

Table 3: Big PixelCNN++ Examples Metrics

Given the size of the large dataset (60,000) images Big PixelCNN++ takes a very long time to train. Due to time constraints, we were only able to obtain results sampled after the third epoch. Despite this short amount of training time, we can see that the model is learning the visual structure of the MIDIs within our dataset as is evident from the ascending/descending structures (which we can interpret as melodic lines) and stacked white lines (representing chords). Moreover, listening to the outputs certainly demonstrates that the model is learning. While there are no clear melodic lines when listening, there is certainly evidence of vertical harmonic structure (certain chords sound very consonant even though the transition between chords don't sound natural). Furthermore, there is

a semblance of rhythm (all be it a fairly weak one). Another interesting aspect of the examples is that the results are far more complex and sounds more virtuosic. Given that the underlying dataset (GIANT-MIDI piano) contains a far wider variety of piano music, a lot of which is more virtuosic than our curated set, the fact that then samples from Big PixelCNN++ reflect this virtuosity demonstrates that it is on its way to learning a representative distribution of the training data.

Link: `https://www.youtube.com/watch?v=J6BoruoANuE`

4.4.4 Competing Model: Music Transformer

We decided to compare our music generation models to Music Transformer, a music generation model that uses a Transformer-based architecture, developed by Huang [4]. The MIDI files are converted into a data representation of discrete token values that are then used used to train the Transformer, a sequence model based on self-attention developed by Vaswani et al. [14]. The model was able to generate high-fidelity piano music.

We decided to use this model as a point of comparison for our models because it uses a symbolic representation of music (MIDI files) and is based on converting the original MIDI file to a new representation in order to take advantage of a class of models, in their case Transformer-based models. We are doing something similar in order to leverage image generation architectures on MIDI files. A key difference is that we make use of a significantly more sparse representation of MIDI files as images. This is with the hope that the visual representation helps the model learn the very visual structure of music / MIDI files.

Particularly, we were curious about how Music Transformer would compare to our top performing model, PixelCNN++, so we trained it for 150 epochs (compared the 120 epochs of PixelCNN++) on the small dataset

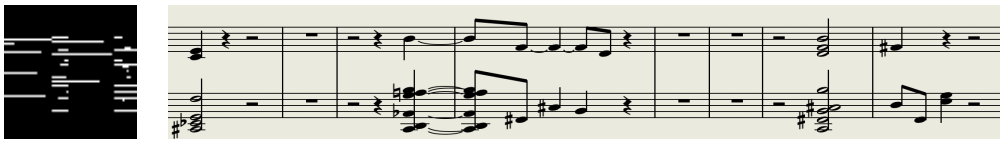

Figure 16: Music Transformer Example 1

	Melody	Harmony	Rhythm
Music Transformer (Example 1)	False	1	False

Table 4: Music Transformer Examples Metrics

As the table metrics indicate, the audio results from the music transformer were not good. There was absolutely no sense of melody or rhythm and the harmony, while some chords were consonant, on the whole was very poor. These were our worst audio results. We also only showed one example due to the model's poor performance.

Link: `https://www.youtube.com/watch?v=ypq1G4aHhgs`

5 Conclusion

Although StyleGAN2 and WassersteinGAN proved not to be valuable tools for music generation, PixelCNN++ surpassed our expectations of success. We were very impressed with the results we were able to achieve using PixelCNN++ as our image generator. We believe that we are the first to attempt solving the music generation task through visual representations of MIDI files. Given the novelty of our approach and our promising results, there are many extensions that we are excited to try. We are excited to continue building on Binary PixelCNN++ to make a model more tailored to our application, train on significantly larger datasets, attempt to generate music for different genres using a conditional PixelCNN++, explore different image representations (ex. change dimensions), and attempt to generate longer clips of music and model longer term dependencies by passing in a latent representation of the earlier parts of the song.

References

[1] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan, 2017.

[2] Ahmed Fakhry, Xinyi Jiang, Jaclyn Xiao, Gunvant Chaudhari, Asriel Han, and Amil Khanzada. Virufy: A multi-branch deep learning network for automated detection of covid-19, 2021.

[3] Kshirsagar M. Zhong M. et al. Gupta, G. Comparing recurrent convolutional neural networks for large scale bird species classification. *Nature*, 11(17085), 2021.

[4] Cheng-Zhi Anna Huang, Ashish Vaswani, Jakob Uszkoreit, Noam Shazeer, Curtis Hawthorne, Andrew M Dai, Matthew D Hoffman, and Douglas Eck. Music transformer: Generating music with long-term structure. *arXiv preprint arXiv:1809.04281*, 2018.

[5] Ajay Jain, Pieter Abbeel, and Deepak Pathak. Locally masked convolution for autoregressive models, 2020.

[6] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks, 2019.

[7] Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Analyzing and improving the image quality of stylegan, 2020.

[8] Qiuqiang Kong, Bochen Li, Jitong Chen, and Yuxuan Wang. Giantmidi-piano: A large-scale midi dataset for classical piano music, 2020.

[9] mathigatti. midi2img. `https://github.com/mathigatti/midi2img`, 2021.

[10] Dhariwal H. Jun et al. P. Jukebox: A generative model for music, 2020.

[11] H.Zen et al. S. Dieleman. Wavenet: A generative model for raw audio, 2016.

[12] Tim Salimans, Andrej Karpathy, Xi Chen, and Diederik P. Kingma. Pixelcnn++: Improving the pixelcnn with discretized logistic mixture likelihood and other modifications, 2017.

[13] Aaron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. Conditional image generation with pixelcnn decoders, 2016.

[14] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[15] Li-Chia Yang, Szu-Yu Chou, and Yi-Hsuan Yang. Midinet: A convolutional generative adversarial network for symbolic-domain music generation, 2017.