Deep Vibes

Sergio Herreros[1*] **Daniel Sánchez**[1*] **Mario Kroll**[1] **Nicolas Villagrán**[1]
[1]Universidad Pontificia Comillas

Abstract

The rate of evolution of deep learning makes it very difficult to be up to date in all of its applications. Every year, thousands of researchers keep updating current state-of-the-art models and developing new algorithms. These models can now achieve precise image captioning, answers to complex problems to which they are presented, image generation and music composition. We will focus on this last topic in this paper. We are going to explore the current state-of-the-art models and also propose new models to achieve this task. The source code is available at: `https://github.com/winoo19/deep-vibes.git`

1 Introduction

Music presents itself as one of the most complex art of the human history. Its variety makes it one of the most special forms of art of the human species. It can be expressed in infinite ways: one or more instruments, slow or fast, with or without lyrics... Choosing a path to follow was not an easy task, due to the vast genres there are in the music field. We had to decide whether our domain would be audio or symbolic. Audio-domain means representing music as raw audio, encoded using the discrete Fourier transform. With it, we can represent all possible sounds (if choosing a sampling frequency higher than human threshold, i.e 20KHz), but also require huge amounts of data and computation.

Symbolic-domain means representing music using notes, events, or text tokens to represent melodies. These representations introduce a high inductive bias, because the model can focus directly on how melodies are created. The advantage of this representation is that it is much more computationally efficient, and smaller models can generate reasonably good music. This representation also has downsides. It introduces a constraint on the possible sounds it can generate, like vibratos or hammer ons, and it also forces us to choose a fixed number of instruments.

Since the complexity of the data was not our goal and our limited resources, we decided to go through the path of piano-only music, using a symbolic domain representation. However, this was not the only drastic decision we had to make. The symbolic music domain allows us, mainly, to train models more efficiently. But another question comes to mind when generating music. Do we want the models to continue songs or to create music from scratch? To answer that question, we first needed to decide which models we would use, because some models are not capable of generating music from scratch due to their nature.

On one hand, conditional music generation allows us to generate music based on a chord progression, a specific music genre, a sentiment, a prompt, etc.

On the other hand, nevertheless, unconditional music generation does not give us any control, and we can only generate random music that will resemble our dataset. In auto-regressive models like RNN or transformers, we can slightly condition music by giving it a starting melody, and asking it to continue it, but it is harder to do this with other models like Auto-encoders or GANs.

The current state-of-the-art involves either models that generate conditional music as well as models that can generate music from scratch. Therefore, we will implement different models so that both options are covered, and then make a comparison of their performance.

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

2 Data and pre-processing

In the previous section we decided how we are going to approach this problem. We are going to train different models to generate symbolic, unconditioned music.

There are two more aspects to decide. First, which and how many instruments to generate. We are going to generate single-instrument piano tracks.

Secondly, which type of symbolic representation to use. We decided to use a piano-roll representation, since it is the most common representation used in the literature. It consists on a 2D matrix, where one dimension represents discretized time and the other one the pitch of each note from to A0 to C8, exactly 88 notes representing the 88 keys of a piano. Each value at each time step is a float between 0 and 1 representing the velocity of that note.

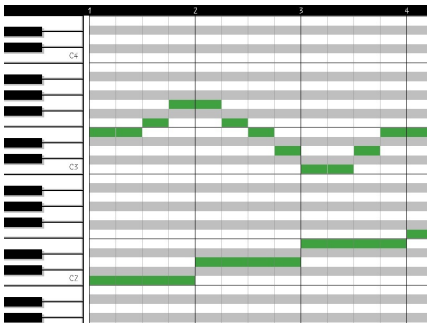


Figure 1: Pianoroll

With this information, we chose the GiantMIDI-Piano dataset, which contains 7000 songs of 1000 composers. Music is represented in MIDI, so we need to first preprocess the dataset to convert it to piano-roll by discretizing the time dimension with a sampling frequency of 16 Hz. We also reduced the size of the dataset by picking only the top 10 composers by number of tracks.

We created a complete pipeline inside data.py, where we go from midi to piano-roll, from piano-roll back to midi and from midi to audio in .wav format.

3 Models

3.1 CNN+GAN

There are many ways the generator and discriminator can process and interpret music. One of them is by using convolutional neural networks (CNNs). The model we will be implementing in this section is a version of MidiNet [1], in which both the discriminator and the generator are CNNs.

However, CNNs applied to music generation (or anything sequential, really) pose a clear problem. That is, that they have a limited (and fixed) context size. And although this is also the case for attention based models, attention relates every data point (note) in their context size with every other, while CNNs only usually relate notes close to each other. As proposed in the MidiNet paper, we will fix this by conditioning the generator on the previous bar.

For that, we will use another model: the conditioner. This model has as input the previous bar to the one we want to predict, and its outputs are concatenated in the different layers of the generator. So, while the generator is made of a few transposed convolutional layers, the conditioner is its mirror. The first transposed convolutional layer of the generator will be the last (regular) convolutional layer of the conditioner, the second of the generator will be the second to last of the conditioner, etc. And the output of each of these layers is concatenated to the respective generator's layer.

The discriminator is also a normal CNN with some convolutional layers and some fully connected layers. But in order not to allow it to overpower the generator, it only has two convolutional layers, and one fully connected layer. The generator has two fully connected layers and four transposed convolutional layers, and the conditioner has four convolutional layers. After every layer, we perform

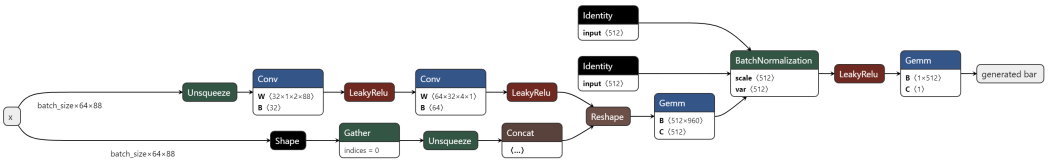


Figure 2: Discriminator's architecture

batch normalization, and the regularization is leaky ReLU for the discriminator and the conditioner and normal ReLU for the generator.

3.1.1 Training stability techniques

GANs face several challenges concerning convergence throughout the training process [2]. In general, these problems can be categorized into two main groups: mode collapse and failure to improve. Mode collapse refers to the inability of the generator to capture the diversity of the data distribution resulting in the production of identical outputs whose discrepancies are mainly noise. Failure to improve occurs when the model oscillates due to the dynamic nature of gradient descent or because of the problem of Vanishing gradients. Many techniques have been tried to aid in the training of the GAN. The first one is feature matching. This is a form of "inverse" regularization, where the higher the lambda, the more similar the output will be to real data (albeit with loss of creativity and risk of overfitting). This is done by adding the term:

$$\lambda \|\mathbb{E}(x) - \mathbb{E}(G(z))\|_2^2$$

to the loss of the generator. We also tried to add a second feature matching term, which compares the output of the first layer of the discriminator for the real data and output for the fake data. It would look like this:

$$\lambda \|\mathbb{E}(f_D(x)) - \mathbb{E}(f_D(G(z)))\|_2^2$$

with f_D being the first discriminator's layer.

The second is one-sided label smoothing. It works by, instead of setting the label of the real data as one, we set it to a slightly lower number, such as 0.9. It has two benefits: first, it makes the loss function smoother. Second, it penalizes the discriminator for being overconfident when predicting real data [3].

The next technique is to, in every training iteration, train the discriminator once and the generator and conditioner twice. This was something we varied along the training process. If the discriminator is performing too well (if it rapidly converged to 0 loss), it is a good way to allow the generator to catch up.

However, after all of this, the discriminator was still easily overpowering the generator. When the discriminator's loss is 0, the gradients don't flow well towards the generator. Thus, if the discriminator predicts perfectly at any point of the training process, the models suffer from vanishing gradients, and so the training is over [4]. To fix this problem, we added dropout to the discriminator. A dropout of 0.2 allows the generator to learn without the discriminator overpowering it.

Another technique we tried was changing the loss function. Instead of the standard (3), we tried using the Wasserstein distance A.2. However, it did not work well. The generator learnt to directly predict something very similar to the previous bar, inputted through the conditioner, which meant it did not generate original music. This fooled the discriminator, since it does not see the previous bar, but it is not what we are looking for. Hence, we kept the GAN loss.

3.1.2 Network specifications

The discriminator is made up of two convolutional layers of 32 and 64 layers respectively. The first one has a kernel of 2 by 88 (the full pitch dimension) and a stride of 2, and the second's kernel is 4 by 1 with also a stride of 2. Then, it has a fully connected layer with 512 neurons. After both convolutions, we applied a dropout of 0.5 (2).

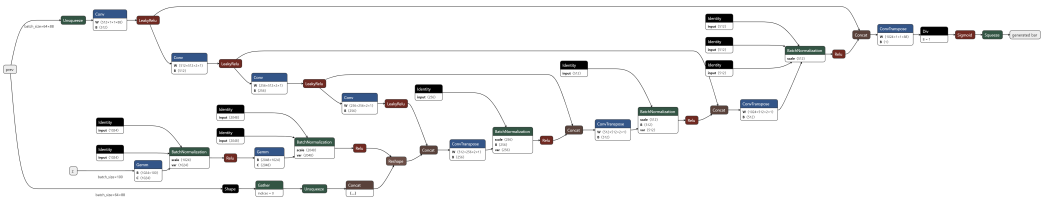


Figure 3: Generator and conditioner's architecture

The generator is made up of a fully connected layer of 1024 neurons and four transposed convolutions of 256, 512, 512 and 1 layers respectively. The first three convolutions have a kernel of 2 by 1 and stride 2 and the last has a kernel of 1 by 88 and a stride of 1 by 2. As said, the generator is the same but mirrored. So the first (regular) convolutional layer has 512 layers (the last transposed convolutional layer's inputs have 512 layers), 1 by 88 kernel and 1 by 2 stride; the second has 512 layers, 2 by 1 kernel and 2 stride; the third has 256 layers, 2 by 1 kernel and 2 stride, and the last has 256 layers, 2 by 1 kernel and 2 stride. Then, before every generator's transposed convolution, we concatenate the output of the previous transposed convolution and the output of the respective conditioner's convolution. Lastly, the activation of the generator is a sigmoid, so that the outputs stay between 0 and 1 (3).

We tried to change the discriminator's structure by adding the previous bar as an input. That is, concatenating it before the first layer to the generated (or real) bar. The aim was to prevent the generator from predicting something too similar to the previous bar. However, the discriminator became too strong, and not even setting dropout to 0.5 helped. Very quickly it converged to close to perfect accuracy. So, dropout was not sufficient as a regulator. We then used instance noise, i.e. adding some noise to the discriminator's inputs so that the distributions are smoother (and more similar) [5].

A couple more fine-tuning things were tried. First, we penalized activating notes over a small value (activation penalty). The objective of this was for the model to either have a note fully activated, or to keep it close to zero, like in a real track. That is because if it is penalized by having notes activated over, say, 0.05, it will only activate the notes it really needs in order to fool the discriminator, and the rest will stay lower than 0.05.

The last improvement we tried was smoothing over each note. The model, especially on earlier iterations (but it also did on later ones, albeit not as significantly), is prone to activating a note on a time step, deactivating it in the next one, then activating it, and so on. We have not figured out why that is the case, so follow up research on this would be interesting. Perhaps it is due to the 2 by one kernels used in the generator.

3.1.3 Results

We were not able to get any substantial result with feeding the previous bar to the discriminator. The bars generated did not look real at all. Figure 10 is an example of a model with noise with 0.3 variance, 0.5 dropout, an activation penalty of 0.5 with threshold 0.05, feature matching with lambdas 0.1 and 1.0 respectively, learning rate 0.00025 for the generator and 0.0002 for the discriminator and batch size of 128.

The output from a bar of all silences (which was the bar used as a start token for all tracks) is shown in figure 11.

The model we chose has a normal discriminator (without the previous bar). It has no noise, 0.2 dropout, an activation penalty of 0.5 with threshold 0.02, feature matching with lambdas 0.1 and 1.0 respectively, learning rate 0.0002 for both the generator and the discriminator and batch size of 128. The output of the model is shown in figure 12.

The output from a bar of all silences is shown in figure 13. It shows the output for two different noise vectors.

3.2 LSTM+GAN

Following the Occam‘s razor principle we start off by generating music with a standard LSTM model to explore the base model capabilities to ensure a safe and sound increase in complexity towards the GANs family [6]. For our generation task using LSTM we cannot impose a start-token as in language models because the output would then be deterministic. Among many other possible strategies, we have decided to initially feed the model with a random selection of contiguous steps taken from the dataset (15).

The results are unsatisfactory. The LSTM fails to learn the underlying structure of data and is biased towards a basic model that seems to be a poor summary of data. We conjecture that this is not a matter of hyperparameter searching but rather of the difficulty of standard LSTM learning piano rolls, which seem to hold pure locally spatial information. As a matter of this, we propose a 2D-LSTM model that aims to learn patterns horizontally, vertically and bidirectionally with increased ease while maintaining the recurrent and long-term capabilities of LSTMs. The 2D-LSTM is composed by two independent LSTMs networks that process piano rolls both horizontally and vertically and are fused together with the aid of a dense layer into a unique image (15).

We can appreciate a significant improvement w.r.t to the standard LSTM, but results are still far from expected. The next step is to introduce a GAN with a generator based on the aforementioned 2D-LSTM model [7]. The training instability has forced us to use different techniques to address it. To prevent overconfidence in the discriminator's predictions we propose feeding it with random noise as well as generated samples to force it learn the underlying structure of real data and not just the generator one. Even though this results in a slightly better training phase, the model constantly ends up committing either model collapse or fails to improve to a decent result (16).

As we have not yet seen promising results, we combine all the previously described with a WGAN model, which is known in the field for providing more stable train without a need to do a deep hyperparameter searching (A.2). In our subject of study this has fixed the model collapse problem and improve model´s convergence (16).

Even though we start to see some patterns that recall piano rolls, this still far from desired. Increasing the training process time results in noisier images and does not have a meaningful impact on the loss function. We conclude that 2D-LSTM model can partially learn patterns from pianorolls but the difficulties of the sensibility in GAN's training make it unfeasible in a reasonable amount of trial-and-error time to polish results.

3.3 Decoder-Only Transformer

We wanted to implement as well context-based models. Transformers [8] seemed as our best option, because of their lack of recurrent units and higher performance. It is not the original transformer, but a decoder-only transformer (A.3), which is widely used for generating tasks. In this case we designed the model with one positional encoding layer, decoder blocks and a sigmoid function at the end to clip the values between 0 and 1. We tried to use the sigmoid with temperature because we hypothesized that the decoder blocks would produce values in a wider range than expected. This turned out to produce output values far either from 0 or 1, resulting in noisy outputs.

To generate music with the transformer, we tried to generate it from scratch and conditional as well. We found out that to generate it from scratch, we needed the sequence length to be greater than one, because if the softmax received a matrix only with negative infinitives, it would return nan values. Therefore, we had to condition the output with at least one vector (17).

The output of the model does not seem real and is badly generated. We tried to condition it by giving it the start of a song, and the output did not improve at all (17). The performance of the model is not as pleasant as we wanted it to be. The model sticks to some notes and does not generate more notes. We eliminated the noise and made it more "real" by reducing the amount of notes per time step to 10 notes at the same time. To achieve this, we kept the 10 greatest values and set the rest of them to 0. As some noise values would still appear, we also set the values lesser than 0.05 to 0.

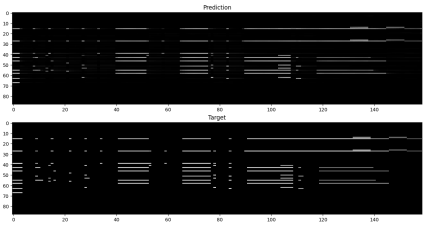


Figure 4: Comparison between input and output of the model

We suspected the model constantly overfitted the data because when an input data is passed to the transformer, the prediction looked too similar (4). We tried to reduce the learning rate, modify the structure of the model and change the loss function, but the outcome would stay the same and the training process followed the same pattern: the first epoch, the loss would suffer a dramatic decrease and then it would be very slowly reduced.

As appreciated, the model had a very poor performance in the music generating task, inferring songs with noise and with lack of rhythm and note variety.

3.4 CNN+VAE

We started training an LSTM-VAE, since our data has a temporal dimension, and we wanted to capture the time dependence with a recurrent model. This model turned out to be impossible to train, at least with the hardware we had at our disposal because training was to slow and the model didn't converge.

We ended up implementing a CNN-VAE. This model had a faster training since it was completely paralellizable by our GPUs, but still we faced numerous problems.

The loss function converged too quickly at a local minimum, and it turned out to be the weigh initialization. Since we used Leaky Relu activations, Kaiming Initialization is the proper way to make sure latent vectors didn't collapse to zero at the beginning of training.

Another issue we encountered was the unbounded KL term as mentioned in [9].Typically, we estimate the $log(\sigma^2)$ directly instead of estimating σ because $\sigma > 0$. This value is used directly in the closed form of the KL divergence (1). But this is problematic because when we then apply $e^{log(\sigma^2)}$ to undo the transformation we end up with values close to ∞. The solution to this was estimating σ directly but applying $ReLU(\sigma_x)$ to make sure we only got positive values. More complex methods to avoid them are explained in [10].

$$D_{KL}(G(\mu_\phi(x), \sigma_\phi(x)), G(0, I)) = \frac{1}{2}\sum_{i=1}^{k} \mu_\phi(x)_i^2 + \sigma_\phi^2(x)_i - log(\sigma_\phi^2(x)_i) - 1 \tag{1}$$

The most difficult part of training the VAE was balancing KL divergence and reconstruction loss. Training the VAE with the standard loss (2) quickly converged to a kl loss near zero, meaning the latent distribution is a multivariate normal, but the reconstructions were practically noise.

$$\mathcal{L}_{\phi,\theta}(x) = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x)] + D_{KL}(q_\phi(z|x)||p(z)) \tag{2}$$

To account for this, we added a parameter γ, to weight down the kl divergence, but we could only achieve one of this two results: A high value of gamma resulted again in a noisy representation of the data, and with low value of gamma we got a model with a really good reconstruction loss, but the latent distribution wasn't even close to a normal, so generating new songs was impossible as seen in Figure 5.

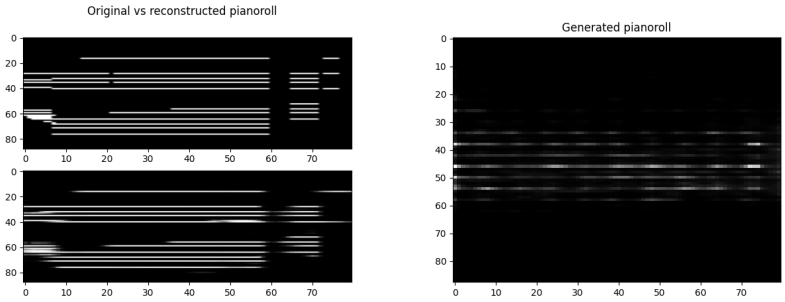


Figure 5: Trained model with $\gamma = 10^{-4}$

The last trick we tried was a variation of Cyclical Annealing of the γ parameter, proposed in [11]. We start the parameter at 0 for an entire epoch, then exponentially increase the value of γ until a defined maximum, 10^{-1} in this case and then restart the process. This technique is the one that allowed us to achieve a good balance between the reconstruction and the KL divergence loss, and eventually got to our best model. This is an improvement of the annealing schedules proposed in [11] since we use an exponential function, which was not explored in the paper, and we got the best results with it.

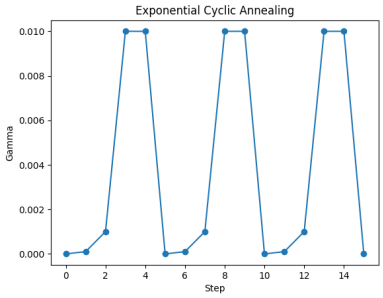


Figure 6: Exponential Cyclic Annealing

Apart from this, we applied some transformations both to input and output data, to make it easier for the model to learn the dataset. We binarized velocity of inputs using a threshold $t = 0.3$, and also normalized and binarized the model generated outputs.

After applying all previously mentioned techniques, we got our best results, with a visually good reconstruction and a generation that has musical structure and contains fragments with recognizable melodies 7.

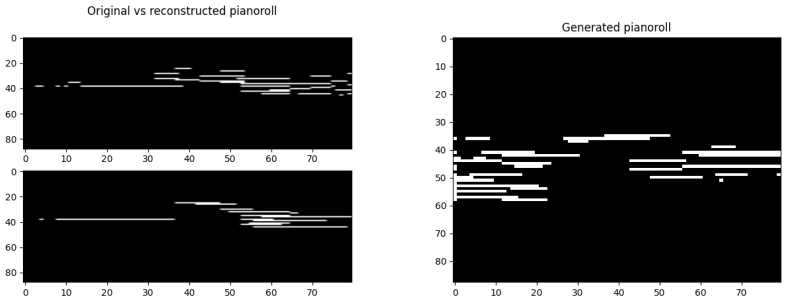


Figure 7: Best model results

4 Workload Distribution

4.1 Sergio Herreros

- `/vae/autoencoder.py`.
- `/vae/main.py`.
- `/vae/models.py`.
- `/vae/train_functions.py`.
- `/vae/train_functions_ae.py`.

Also contributed to:

- `data.py`.
- `datasets.py`.
- `midi.py`.
- `utils.py`.

4.2 Daniel Sánchez

- `eval_gan_cnn.py`.
- `gan_cnn_model.py`.
- `train_gan_cnn.py`.

4.3 Nicolás Villagrán

- `models_lstm.py`.
- `train_gan_lstm.py`.

4.4 Mario Kroll

- `transformer.py`.
- `transformer_train.py`.
- `train_functions_transformer.py`.

References

[1] Li-Chia Yang, Szu-Yu Chou, and Yi-Hsuan Yang. *MidiNet: A Convolutional Generative Adversarial Network for Symbolic-domain Music Generation*. 2017. arXiv: 1703.10847 [cs.SD].

[2] Samuel A. Barnett. "Convergence Problems with Generative Adversarial Networks (GANs)". In: (2018). DOI: arXiv:1806.11382.

[3] Tim Salimans et al. *Improved Techniques for Training GANs*. 2016. arXiv: 1606.03498 [cs.LG].

[4] Martin Arjovsky and Léon Bottou. *Towards Principled Methods for Training Generative Adversarial Networks*. 2017. arXiv: 1701.04862 [stat.ML].

[5] Simon Jenni and Paolo Favaro. *On Stabilizing Generative Adversarial Training with Noise*. 2019. arXiv: 1906.04612 [cs.CV].

[6] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. arXiv: 1406.2661 [stat.ML].

[7] Yuki Tatsunami and Masato Taki. *Sequencer: Deep LSTM for Image Classification*. 2022. DOI: arXiv:2205.01972.

[8] Ashish Vaswani et al. "Attention Is All You Need". In: (2017). DOI: arXiv:1706.03762.

[9] Arash Vahdat and Jan Kautz. *NVAE: A Deep Hierarchical Variational Autoencoder*. 2021. arXiv: 2007.03898 [stat.ML].

[10] David Dehaene and Rémy Brossard. *Re-parameterizing VAEs for stability*. 2021. arXiv: 2106.13739 [cs.LG].

[11] Hao Fu et al. *Cyclical Annealing Schedule: A Simple Approach to Mitigating KL Vanishing*. 2019. arXiv: 1903.10145 [cs.LG].

[12] Martin Arjovsky, Soumith Chintala, and Léon Bottou. *Wasserstein GAN*. 2017. arXiv: 1701.07875 [stat.ML].

[13] PH.D. CAMERON R. WOLFE. *Decoder-Only Transformers: The Workhorse of Generative LLMs*. 2024. DOI: https://cameronrwolfe.substack.com/p/decoder-only-transformers-the-workhorse.

A Appendix

A.1 GAN models

Generative Adversarial Networks (GANs for short) are a type of model used for generating data out of random noise that is indistinguishable from other (called real) data. In our case, as it is the aim of this project, we will use this model to generate music as similar as possible (and, at the same time, original) to real music scores.

GANs work by facing two models against each other. One of them is called the generator, and the other is called the discriminator. The generator's objective is to *generate* data as similar as possible as the original, while the discriminator's is to *discriminate* between real and fake data, i.e. label fake data as 0 and real data as 1. So the input of the generator is random noise (every random noise vector is associated by the generator to an output; that way, it can generate different scores), and the input of the discriminator is a music score, be it real or fake.

The equilibrium of the game is when the generator outputs music scores perfectly resembling real music, and so the discriminator cannot distinguish between real and fake data. It will then always guess with 50% confidence.

Thus, the loss function for the model must be something that takes into account both how close the discriminator's labels for the real data are to 1 and how close the labels for the fake data are to 0. A good way to do this is to use binary cross entropy (in fact, it is the standard for these models). Let's call the discriminator D, the generator G, the distribution underlying the real data p_{data} and the distribution underlying the generator's outputs p_z. Then, the loss function would look like this:

$$\min_D \max_G L(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1-D(G(x))] \tag{3}$$

The solution to this minimax game would be when $p_{data} = p_g$, and so the discriminator would always guess randomly. However, this will not always be the case, and in fact it is very easy for the discriminator to "overpower" the generator, rendering the model useless. There are many techniques used and actively researched to make both the generator better and the discriminator worse, in order to give the generator a fighting chance [6].

The training of GANs is done in two parts. First, the discriminator is trained on real data, and the loss is computed. Then, fake data is generated and the discriminator is trained on it, and the loss is computed again. Then, the parameters of the discriminator are updated on the sum of both of these losses. Lastly, the discriminator predicts again on the same fake track previously computed, and using this prediction, the generator's loss is calculated and its parameters updated (9).

A.2 WGAN model formulation

Traditional GANs are formulated under the Jensen-Shannon divergence to measure the discrepancy between real and generated probability distributions. The improvement introduced by WGANs w.r.t original ones is the use of the Wasserstein distance [12]. This metric aims to find an optimum for the transportation problem: Given two mass functions find the best strategy to transform one into another. This gives a notion of disparity between our two probability distributions (Real data and Generated data).

$$W_p(\mu,\nu) = \inf_{\gamma\in\Gamma(\mu,\nu)} \left(\mathbf{E}_{(x,y)\sim\gamma} d(x,y)^p\right)^{1/p} \tag{4}$$

This equation can be transformed into (Kantorovich-Rubinstein duality) (include proof):

$$W_1(P,Q) = \sup_{\|f\|\text{Lip}\leq 1} (\mathbb{E}x \sim P[f(x)] - \mathbb{E}_{y\sim Q}[f(y)]) \tag{5}$$

We note that the discriminator must ensure 1-Lipschitz continuity. The authors use weight clipping to restrict the minimum to a compact space to meet this requirement. As a matter of this, the problem is continuous and almost differentiable everywhere. In comparison with JS divergence, this metric approximates better to the meaning and nature of our problem, as the metric converges to 0 when the two distributions are similar and is great otherwise, dealing with vanishing gradient problem.

A.3 Decoder-Only Transformer Structure

The structure of this model is based on the decoder-only transformer [13]. In this structure, the input data goes through a positional encoding layer at the beginning and then travels through a number of decoder blocks. This blocks receive an input matrix of dimensions [batch size, context size, pitch dimensions] and return a matrix of the same dimensions (8).

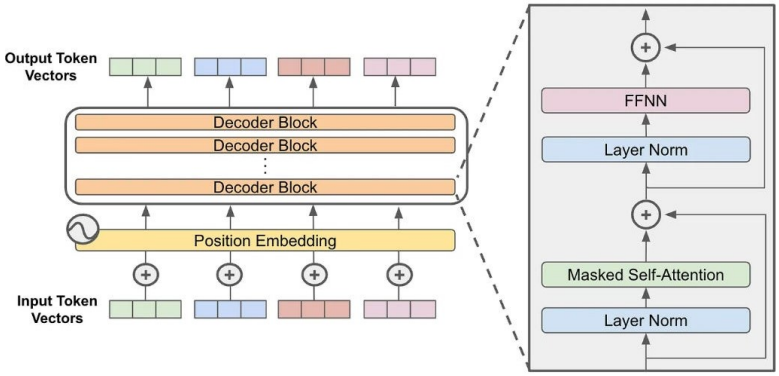


Figure 8: Decoder-Only transformer

Since we want the model to predict vectors from previous vectors, the output of the model should be the input matrix shifted left. Therefore, the loss of the model will be calculated using binary-cross entropy with logits, comparing the input data shifted left one position (without the first context vector) with the output data shifted right one position (without the last context vector). In this case, the loss function is in charge of clipping the output values between 0 and 1. When inferring, however, a sigmoid layer is added at the end to ensure the clipping of the output values.

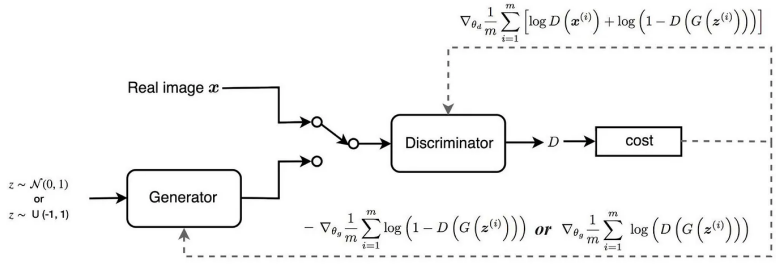


Figure 9: GAN training loop

A.4 More results

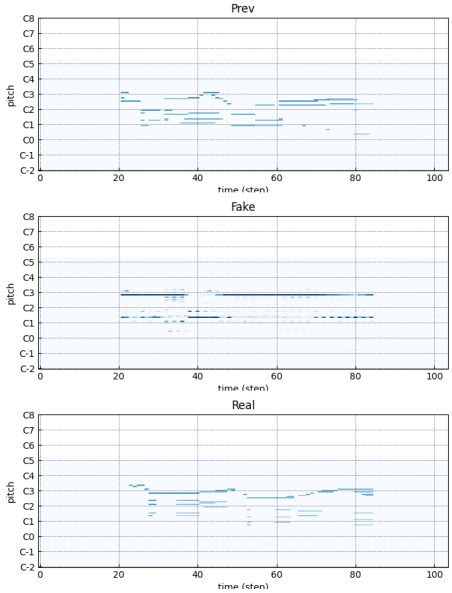


Figure 10: Output of model

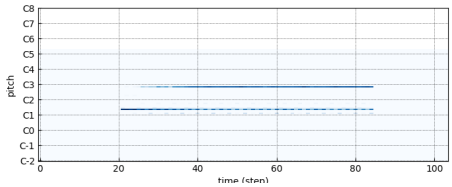


Figure 11: Output of model from silences

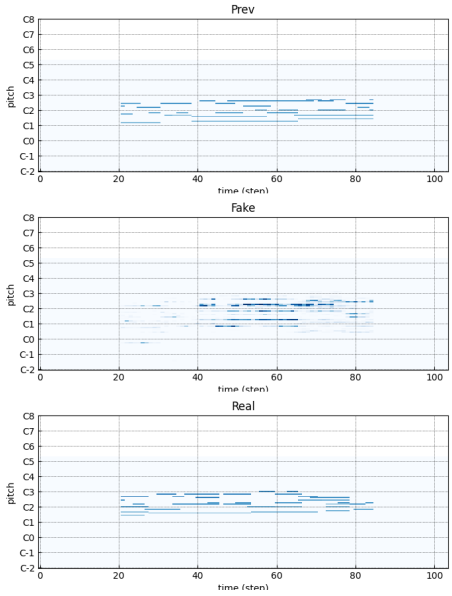


Figure 12: Output of final model

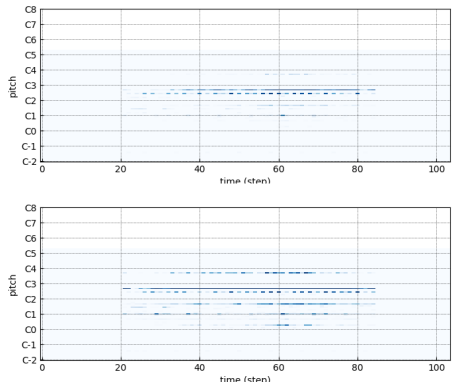


Figure 13: Output of final model from silences

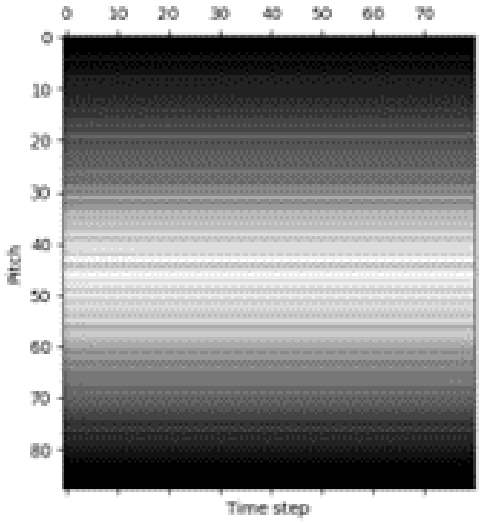


Figure 14: Basic LSTM output

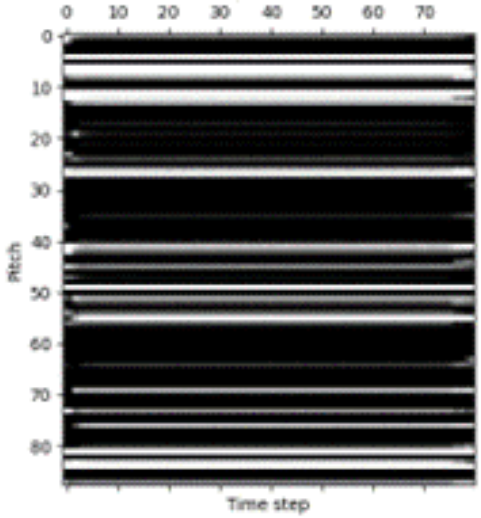


Figure 15: Basic LSTM and 2D-LSTM outputs

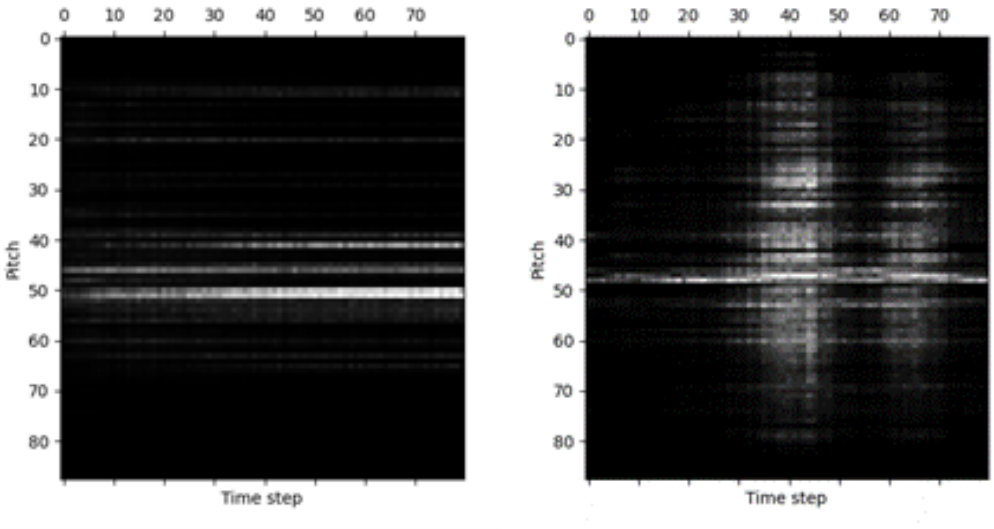


Figure 16: GAN and WGAN generated images.

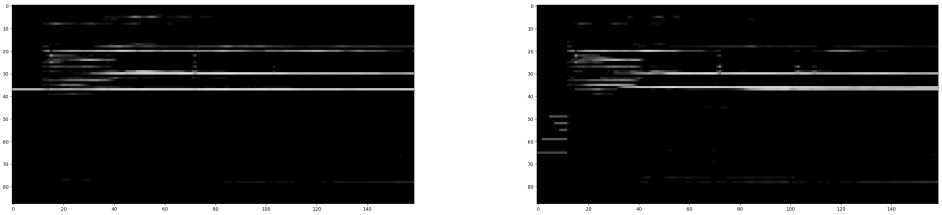
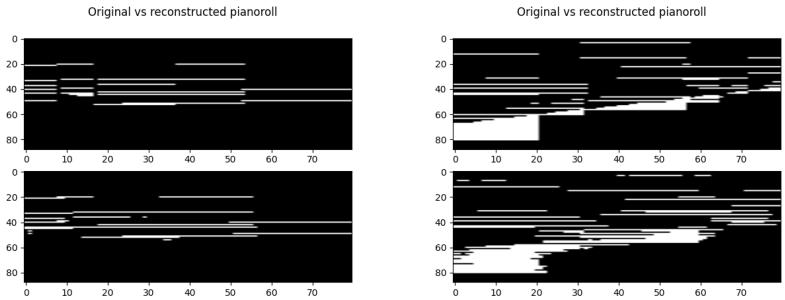
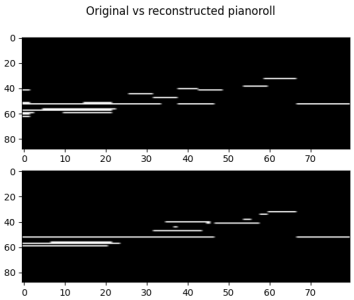


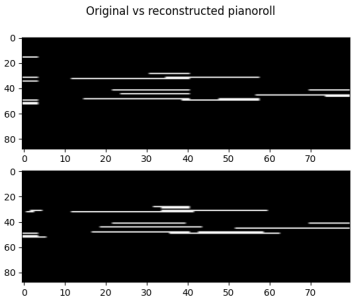
Figure 17: Transformer output with unconditional and conditional generation

A.5 CNN+VAE Results

A.5.1 Reconstructions







A.5.2 Generation