Conditional MIDI Generation using a Transformer-Based Variational Autoencoder

Connor Casey, Quargs Greene, Alex Melnick, Bogdan Sadikovic
ccasey@bu.edu ggreene@bu.edu melnick@bu.edu bogdans@bu.edu

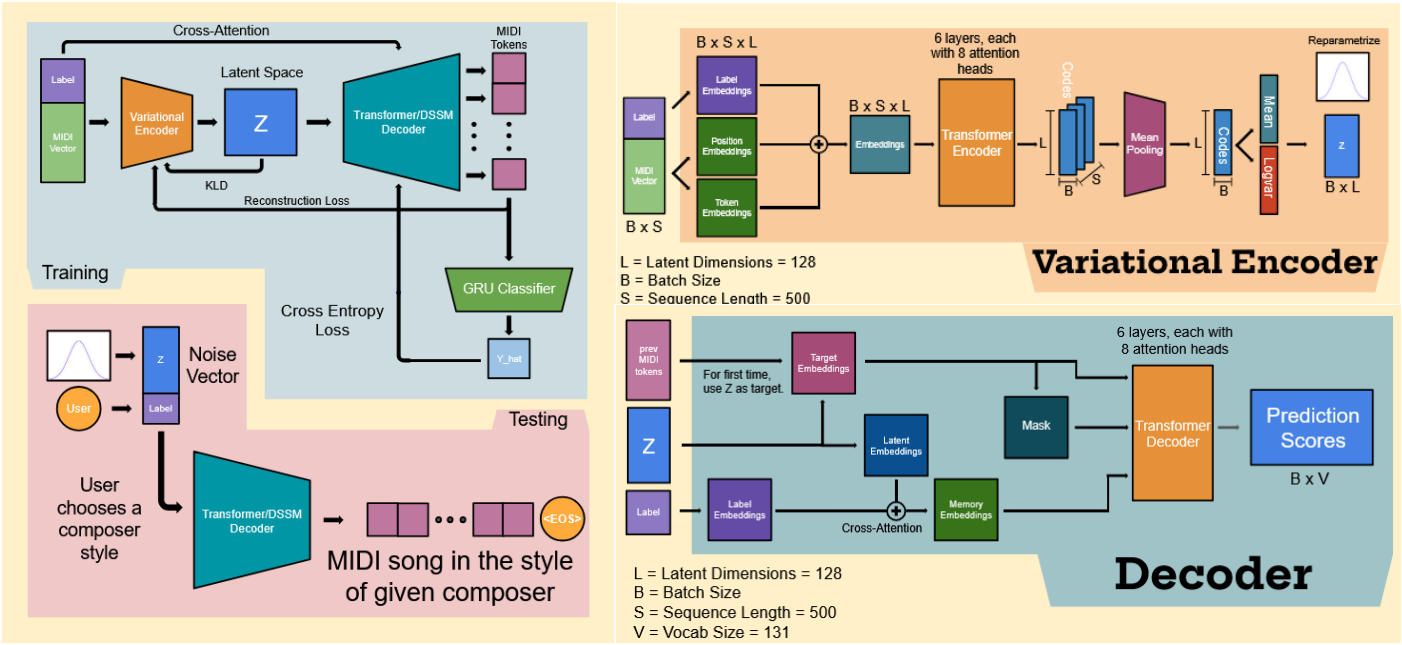

Figure 1. Model Architecture

1. Task

The focus of this project is using a transformer-based variational autoencoder (VAE) to perform conditional generation of classical music in a specific composer's style. The most difficult portion of this project has been the design and implementation of the transformer-based VAE. We've faced a ton of issues, specifically memory leaks and computation graph corruption due to the autoregressive structure of our decoder. This prevented us from utilizing cross entropy loss, which severely hampered and bottlenecked many of our results. We ended up having to axe teacher forcing and the autoregressive decoder. Furthermore, we have worked to balance space constraints with robustness of training, test, and validation data, considering the tradeoffs involved in using a less augmented dataset and smaller token vocabulary in exchange for smaller file sizes. In the future, we'd like to scale up our model to work with an expanded vocabulary, multiple channels to simulate polyphony (the playing of multiple notes at once), and use more in-depth measurements of reconstruction loss and a more accurate classifier to train a model with stronger conditional generation.

2. Related Work

With the rise of deep learning networks, MIDI generation has improved considerably. Typically, due to the nature of their architecture, previous works have mostly been encoder-decoder CNN networks or used LSTMs. Both Midipgan [9] and MIDINet [10] use generative adversarial networks (GANS) to effectively train the model by allowing the discriminator to test melody distribution. However, GANs are notoriously unstable, and as noted in Arora et al.[11] that instability can lead to issues when training sufficiently complex models, leading to the use of less-complex models that ultimately cannot generate temporally-sound music. The use of transformer networks for MIDI data generation has demonstrated some promising results [5][6]. Though transformer networks are not widely used, their ability to learn sufficiently complex data and their wide-use in other domains have pushed us to experiment with them, as opposed to using previously demonstrated methods. There also exists some indication that deep learning approaches can be used to generate instrument or

genre-specific music [6][8]. In our investigation, we hope to address the problem of producing distinct musical styles within a single genre.

Due to the requirements of the project and the feedback we received on our proposal, we wanted to look into an additional model architecture that could be used for generation. Upon discussion with Professor Kulis, we learned that traditional RNN's have been increasingly superseded by newer deep state-space + RNN models, called Neural State Space Models (NSSM). Numerous papers have utilized these for time-series forecasting and full generative models [12][13]. NSSMs are effective for handling long sequences and can work in both autoregressive and non-autoregressive settings. For our goal of conditional MIDI song generation, where our only input will be the composer/artist, NSSMs will allow us to generate MIDI songs without requiring previous tokens as inputs.

One of the first papers to introduce the idea of using a VAE for MIDI music generation was [24]. The paper introduces a model for generating and modifying symbolic music using a variational autoencoder. It takes in MIDI files and breaks them down into three components: pitch, velocity, and instrument. These are encoded into a shared latent space, which the model can then use to reconstruct the original music or generate new sequences. One of the key features is that it can do style transfer by changing certain parts of the latent code to reflect a different composer's style. The model uses softmax outputs for categorical features like pitch and instrument and trains with mean squared error loss.

One thing that immediately stood out to us is their use of MSE loss instead of cross-entropy for things like pitch and instrument, which are clearly categorical. Using MSE here doesn't make a lot of sense because it assumes a continuous output, and that could really mess with the gradients and lead to less confident predictions. Cross-entropy would be a much better fit for one-hot encoded targets and would likely lead to sharper and more accurate outputs. For this reason, we set out to use cross-entropy loss in our model, however this caused significant challenges as described below.

3. Approach

Our approach from the onset was to use a type of VAE that could have autoregressive components to fix an issue faced by autoregressive models, deterministic output given the same input. By creating a VAE, we could ensure that, though the label might be the same, the random input noise would ensure a fully unique song conditioned on that label. When looking at the domain of MIDI token sequences for music, we wanted to capture long-term structure and patterns, which transformers are quite effective at. We represent MIDI songs as token sequences using pretty-MIDI and a specific token vocabulary ranging from 0-131.

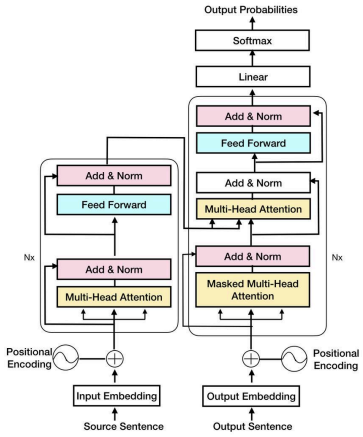

Figure 2: Transformer Architecture (Left:Encoder,Right:Decoder)

3.1 Variational Encoder

For the encoder, we embed both the sequence and label, using position embeddings on the sequence. We then feed this into a `nn.TransformerEncoder()` object to obtain a set of latent codes, one for each token in the sequence. To make the latent space express information about the entire sequence, we use a mean pooling layer, to obtain a single latent code. Then, as is traditional in VAEs, we use a linear layer connected to two identical linear layers, each representing the mean and log variance of the prior distribution present in the latent space. The prior distribution is an ideal of what we'd like the latent space to adhere to, so we can use KL divergence to measure the difference between the prior, our ideal latent space *P(i)*, and the posterior, our reality, *Q(i)*.

$$D_{KL}(P||Q) = \sum_i P(i)\log\frac{P(i)}{Q(i)}$$

This is important because we need to be able to feed random noise to our model, and have it be able to decode a song from that. The choice of our prior distribution is important to ensure we can reliably generate unique songs conditioned on the label. As standard gaussians are often used for posterior distributions of unconditional generations, we use a mixture of 34-gaussians with unit variance, each initialized with a random means sampled themselves from standard gaussian. This allows us to measure the KL-divergence of each encoded sample differently based on their label, improving the ability of the model to keep classes separate in the latent space as depending on the label, *P(i)* will change. We can detail this as follows, the current posterior is defined by the VAE's layers:

$$q(\mathbf{z} \mid \mathbf{x}, y) = \mathcal{N}(\boldsymbol{\mu}_q, \mathrm{diag}(\boldsymbol{\sigma}_q^2))$$

Our prior is a gaussian centered on a mean determined by the label:

$$p(\mathbf{z} \mid y) = \mathcal{N}(\boldsymbol{\mu}_y, \mathbf{I})$$

Thus, our KL-divergence can be expressed as:

$$\mathrm{KL}(q(\mathbf{z} \mid \mathbf{x}, y) \parallel p(\mathbf{z} \mid y)) = \frac{1}{2} \sum_{i=1}^{d} \left[\log \left(\frac{1}{\sigma_{q,i}^2} \right) + \sigma_{q,i}^2 + (\mu_{q,i} - \mu_{y,i})^2 - 1 \right]$$

To avoid the model undergoing posterior collapse, which is where VAE essentially "gives up" on bothering to learn how to align the posterior with the prior, we utilize something called *annealing*. Fu et al [25] also had to utilize annealing for their VAE with an autoregressive decoder, but used cyclical annealing. Annealing means weighting the KL divergence very low in the beginning, and over the course of training increasing it. We have the weight increase exponentially over the course of epochs, up to a maximum value of 0.5. This increasing annealing can lead to an increase in loss during the later epochs due to the scale of the KLD compared to the other losses. In the future, we'd like to implement Fu et al.'s [25] use of cyclical annealing to address the U-shaped loss we consistently would see.

3.2 Decoder

Our decoder has by far been the problem-child of this project, we've had to completely redesign it in order to get our reconstruction loss, Cross Entropy, working correctly. Similar to our encoder, our decoder uses a `nn.TransformerDecoder()` object. We will detail our original design for the decoder, and the multitude of issues associated with it, and how we addressed them. Our decoder was originally designed to be autoregressive, predicting a single token every time inputs were fed into it. The latent space, Z, was used as input to the decoder's memory embeddings and as a way to perform weight modulation on the target embeddings. We utilized the label, and all previously generated tokens (or a start token if it's the first time) to use cross-Attention. Cross-Attention in the context of conditioning on a label, is the process of embedding the label and utilizing it as part of the keys and values of the transformer, while the queries are from the generated tokens. Lerch et al. [26] use transformer-based autoencoders with cross-attention to conditionally generate skeletons based on specific actions. Using this, the model learns to pull important information from the labels during generation. In our original architecture, the decoder was called iteratively for each token until an end was predicted or the max length was reached. During this, we'd also utilize teacher forcing to avoid early errors from compounding errors, something very common in autoregressive models. To measure the reconstruction loss between the model's output, and the original data, we'd need the logits/prediction scores of each token in the generated sequence. However, after over a week of debugging, we were not able to get this decoder functioning. It suffered from posterior collapse (due to teacher forcing), PyTorch computation graph corruption, and cascading memory leaks that when used with Cross Entropy would cause it to crash repeatedly.

To address this, we rewrote the entire decoder to not function autoregressively. Instead of generating one token at a time, it generates the entire sequence all at once, with a few repeated refinement iterations of feeding the generated sequence back into the decoder as the target. On the first iteration, the latent space, *Z*, is used as the target (Figure 1). It still uses cross-attention, and combines the latent space with the labels to create memory embeddings for the model. The embeddings are then fed into the transformer decoder to output predictions for each token in the generated sequence. Following this, we just take the argmax of each token's prediction scores for the entire sequence to get a fully generated sequence. Then as a hyperparameter, we have a number of refinement iterations that feed this sequence back into the model to reduce randomness. Typical of autoencoders, we need to use a loss function that describes the difference between the input data *x*, and the output data x_hat. This is called the reconstruction loss, and can be a variety of different loss functions from MSE, to what we will be using for this: cross entropy. Let's define *x* as a series of tokens:

$$\mathbf{x} = (x_1, x_2, \ldots, x_T), \quad x_t \in \{1, \ldots, V\}$$

Let Defined $\hat{x}$ as the decoder's output following a Softmax:

$$\hat{\mathbf{x}} = (\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_T), \quad \hat{x}_t \in [0,1]^V, \quad \sum_{j=1}^{V} \hat{x}_t^j = 1$$

Then the cross-entropy reconstruction loss is:

$$\mathcal{L}_{\text{recon}} = -\sum_{t=1}^{T} \sum_{j=1}^{V} \delta_{x_t=j} \log \hat{x}_t^j$$

The decoder outputs logits for each token, which after fed through a softmax function become probabilities of which is the likeliest token in the vocabulary. Using cross entropy, we

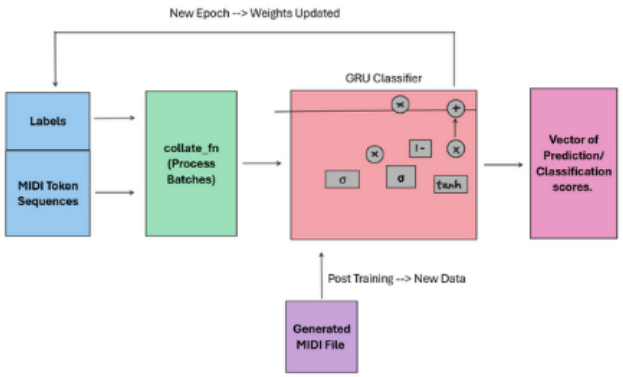

Figure 10: GRU Classifier Training Pipeline

can quantify the difference between these two sequences of categorical tokens.

3.3. GRU Classifier

In order to evaluate the different styles of music that our model generated and verified that there was a consistency in the style of music the model was generating, we used a Gated Recurrent Unit (GRU) architecture in order to perform the composer based style classification. This classifier uses the tokenized MIDI sequences and predicts the composer's label from which the sequence originated from (or is most similar to). The architecture accepts a vocabulary size of 131 tokens, and turns them into a 128 dimensional embedding space. The embedded sequence is passed through a multi-layer GRU with two layers, which has 128 hidden units per layer. This enables the model to better capture long-range dependencies from all contexts in the given sequence. To further improve the model's ability to learn and contextualize data, we added a self-attention mechanism to the hidden GRU states. This allowed the classifier to focus on the most relevant parts of the sequence by computing the attention weights via a learned linear projection, followed by a softmax normalization across time steps. After the weighted sum is passed through a layer normalization step, the training gets stabilized. This representation is subsequently fed into a two-layer connected network, ending in a softmax output over the composer classes. We used Cross-Entropy loss and the ADAM optimizer, with early stopping based on validation loss.

In order to avoid mixing in data being used by the classifier and the generation model (to avoid bias), we only used the validation and test splits from our dataset in order to train the model and evaluate it. For better accuracy, we also had

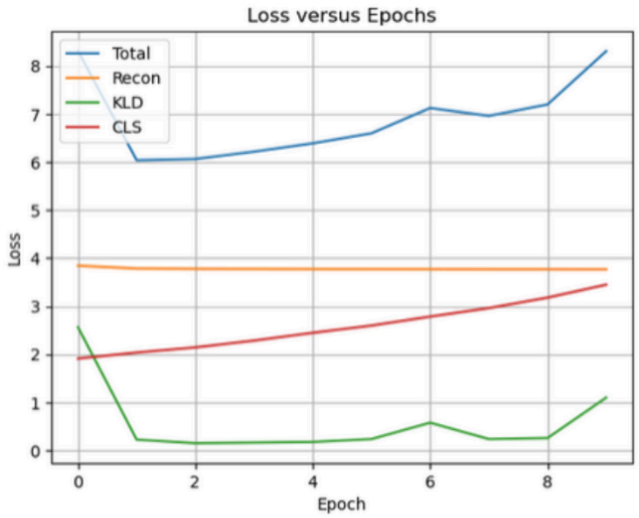

Figure 3: Loss v.s Epochs

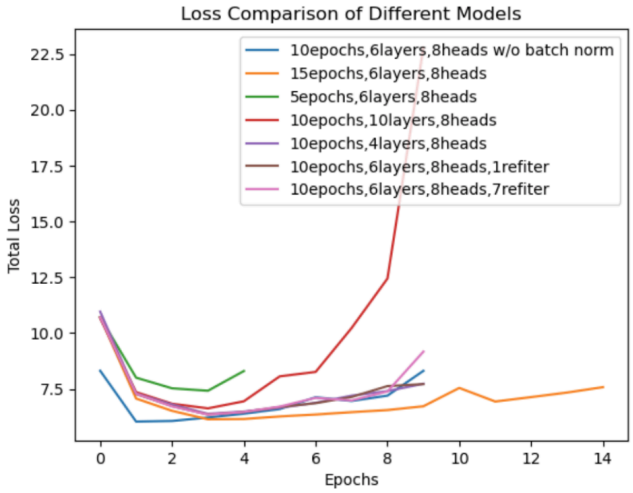

Figure 4: Loss v.s Epochs for Different Models

the model check accuracy based on top-3 most likely composers, as some songs would sound similar. The reason we chose a GRU model architecture over a LSTM architecture was mainly due to the GRU's greater efficiency in training, as well as requiring a smaller dataset for training. As the classifier was being trained on a smaller portion of the dataset, 30% of the dataset, a GRU felt more useful for the purpose of classification of MIDI files. We trained the model with different parameters, such as 256 hidden layer size and 256 embedded dimensions, and 3 layers. However, the GRU model was sensitive to many of these tunings, as it would often overfit if trained for too long.

3.4 Hyperparameter Tuning

For hyperparameter tuning of the VAE, we established a few important hyperparameters as necessary to be tuned: Number of Training Epochs, Number of Layers in Transformers, Number of Refinement Iterations. Due to time constraints, we chose not to do grid search or random grid search, and instead tune each hyperparameter in isolation.

4. Datasets

We used the MIDI and Audio Edited for Synchronous Tracks and Organization (MAESTRO) V2.0.0 dataset for this project. V3.0.0 only differs in content compared to V2.0.0 in that it removes six chamber music works originally containing strings. The original metadata files contain six fields which are: "canonical_composer", "canonical_title", "split", "year", "midi_filename", "audio_filename", and "duration". Since we were only concerned with MIDI, we ignored the "year" and "audio_filename" metadata altogether [3]. The dataset contains a total of 1276 performances and 7.04 million notes, totalling 198.7 hours of music. Before we began using the dataset, we preprocessed it to get the data to be in a suitable format for the generative models and the GRU

model during training. We tokenized all the MIDI sequences, so that the models can read the files as sequences of tokens. Due to difficulties feeding in the larger file sizes of larger token vocabulary, we ended up going with a smaller vocabulary size of 131 tokens which included pitches mapped to a range of 0-127 and three special <bos>, <eos>, and <rest>, respectively signifying the beginning and end of the song, and rests. The tokenization of the MIDI files for a larger token vocabulary was implemented as follows: 1. <bos> and <eos> tokens still signify the beginning and end of each work. 2. Time shift tokens with a temporal resolution of 10ms quantize the data into time shifting events which can be added together and total up to a 1s time shift between events, thereby eliminating the need for <rest> 3. Velocity tokens indicate the velocity of each event and are quantized into 32 velocity bins. 4. Note-on and note-off events are tracked, with the tokenization vocabulary supporting multiple simultaneous note-on events. The token vocabulary size is 390 tokens and the maximum token sequence length is 500 tokens. MIDI files with lengths exceeding this sequence length were split into multiple consecutive sequences. An inverse token dictionary allows for generated token sequences to be converted into MIDI files.

We experimented with training on multiple versions of the dataset, but ultimately used the non-augmented version with a simple 30-60-10 train/test/validation split due to file size constraints. The augmented version of the dataset was augmented via numerical shifting of the note values corresponding to frequency (i.e. pitch) shifts. Let *s* be the number of semitones each sequence was transposed by. Each token sequence was pitch-shifted up and down by an amount of semitones contained in the set $s \in [-12, 12]$, where *s* is an integer. Additionally, there was an alternative splitting approach available utilizing iteration through a variety of different train/test/validation splits for the dataset, including k-fold cross-validation with $k \in [2, 5]$, and a simple 60/30/10 split of the dataset. The authors of the MAESTRO dataset suggest a non-shuffled train/test/validation split with 962 files designated as training data, 177 files designated as test data, and 137 files designated as validation data. However, we chose to deviate from this suggestion, as there was no stratification or shuffling within each subset of the dataset using this approach.

We also ended up removing 45 invalid MIDI files. Each sequence was also labeled with a composer and work title. Due to the fact that the dataset authors fail to distinguish between composers and performers and therefore occasionally annotate the data either exclusively with a performer's name or with a composer and a performer's name, we also dropped such records, bringing the total number of composer classes from 55 down to 34.

5. Evaluation Metrics

In order to evaluate our VAE for conditional MIDI generation we have two parts to evaluate: 1. The quality of the generated MIDI sequence 2. The class separation of the VAE's latent space. Due to the presence of our GRU classifier as an important tool to ensure the model was creating similar songs to other songs of the same label, we wanted to utilize it again to evaluate the model. This time, the GRU is trained with ⅓ as many classes to maximize its accuracy rating.

We evaluated the model by obtaining the accuracy and top-3 accuracy. We did this by generating 10 songs given a specific label, and then feeding that into the GRU, and taking the prediction with the highest confidence, repeating for each of the classes the GRU was trained on. However, we also wanted to use a standard evaluation criteria for the quality of our generations, and we settled on cosine similarity. Cosine similarity is defined as the inner-product between two vectors, divided by their magnitude:

$$S_C(A, B) := \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$$

As such, similarity scores can range from -1 to 1. For each generation, we calculate this on a class-by-class basis, comparing each generation to 10 songs of its own class. We then compare this against how similar they are to songs of other classes, averaging it over all other classes, to get class relative cosine similarity, defined as:

$$\text{Intra}(c) = \frac{1}{m^2} \sum_{i=1}^{m} \sum_{j=1}^{m} \text{S}_c(s_i, s_j)$$

$$\text{Inter}(c) = \frac{1}{m \cdot |R_{\neg c}|} \sum_{i=1}^{m} \sum_{j=1}^{|R_{\neg c}|} \text{S}_c(s_i, r_j)$$

$$\text{RelCosineSimilarity}(c) = \text{Intra}(c) - \text{Inter}(c)$$

This gives a standard way of measuring our model's conditional generation, with a range of -2 to 2. In terms of measuring the separability between classes in the latent space, we first compute a PCA projection of the encodings of 10 MIDI sequences of each class. This gives a sense of how cluttered the latent space truly is. Then for a standard way to quantify the separation, we decided to utilize silhouette score. Silhouette score describes how well samples in a cluster stick to its own cluster (are clusters mixed or not). It uses the difference between the average distance between points of the same cluster a(i), and the closest cluster to any given point, b(i) to do this. It's defined as:

$$a(i) = \frac{1}{|C_I| - 1} \sum_{j \in C_I, i \neq j} d(i, j) \qquad b(i) = \min_{J \neq I} \frac{1}{|C_J|} \sum_{j \in C_J} d(i, j)$$

$$s(i) = \begin{cases} 1 - a(i)/b(i), & \text{if } a(i) < b(i) \\ 0, & \text{if } a(i) = b(i) \\ b(i)/a(i) - 1, & \text{if } a(i) > b(i) \end{cases}$$

As can be seen, the silhouette score can range from -1 to 1, similar to the cosine similarity. The closer it is to either value, the further separated and cleaner the latent space is, while the closer it is to 0, the more similar to random noise.

6. Results

← Click here to listen to a generated sample! (PDF Viewer with Multimedia support such as Adobe Acrobat required)

The GRU classifier had an overall accuracy of 45% for top 1 accuracy and a 76% accuracy for top 3 accuracy. Once we shortened the class number from 34 to 12 classes, which significantly improved the model's accuracy, going from 20% accuracy to 45% in top 1. The model showed that when given too many epochs, generally anything over 8, it would begin to overfit, and would not learn from any new training received. The model really improved once we added self attention, as it was able to contextualize data better. One issue that the model has is that the amount of data it is trained on is very low. It is trained on roughly 30% of the dataset, and as such doesn't get the chance to truly form the best patterns behind the composer and the music they create. This can hurt the overall model, as a poor classifier can take the generation model into a direction it shouldn't have gone, where it may mislabel it, and as such hurt the overall model's success.

As we chose a more efficient model architecture with the GRU, we also lacked some ability to form contextualization between longer sequences, which could have weakened its accuracy. A larger transformer classifier would have been better for contextualizing the dataset, but with the size of the

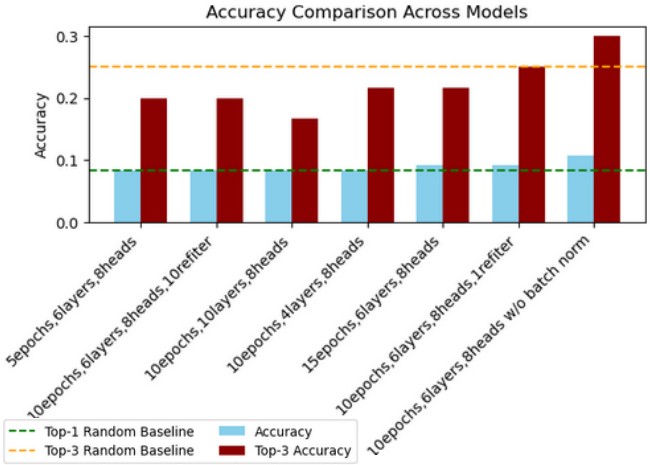

Figure 6: Accuracy for Different Models

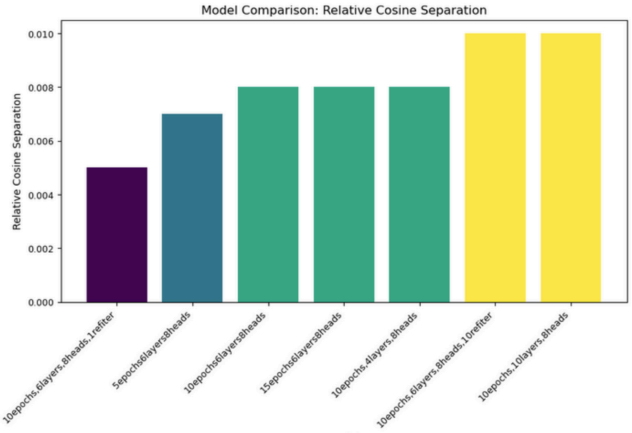

Figure 7: Relative Cosine Similarity for Different Models

current dataset allocated for the classifier, that would not be an effective or successful model. In figure 11, we can see that the GRU had a steadily decreasing loss. This indicates that the model did not overfit to the training data, but also that it did not lack any learning. The accuracy also steadily increases, which stops at around 45% for the top 1 accuracy. When accounting for the chance of a random guess, the model is 5.5x better than a standard guessing. The model would benefit from a larger dataset, but we had to cut back on that in order to allow the generation model to train on substantial enough data.

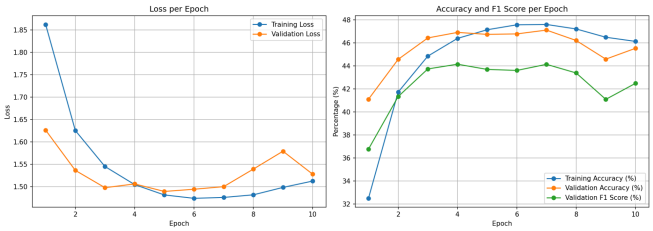

Figure 11: GRU Classifier Results and Evaluation

In Figure 3, we can see the anatomy of the U-shaped loss that is mostly influenced by a CLS increase over time and a KLD jump near the end.. The reconstruction loss remains strikingly consistent, only decreasing somewhat over the course of the model's training. It's possible that the data is quite easy to reconstruct, or more accurately, that to Cross Entropy, it seems easy to reconstruct. The quality of the generations point to the idea that we need a better

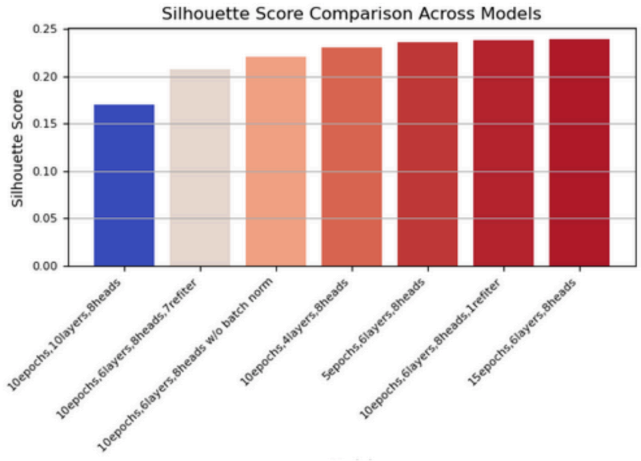

Figure 5: Silhouette Score for Different Models

reconstruction loss function that takes into account local structure and patterns, rather than token-by-token accuracy. The CLS (classifier) loss, remains consistent across epochs, with its rise in value being attributed to the use of exponential annealing that causes it to become closer to its actual influence. This illustrates that our model is not improving when it comes to making the GRU confident in its classifications.. The KLD loss starts high, and rockets down, which is good. It doesn't experience posterior collapse, and consistently decreases until around the 8th epoch where it begins to rise again. This is partly due to the exponential annealing, but even without it this pattern remains. This suggests that near the end of training, the model's latent space becomes distorted.

When analyzing the losses of Figure 4, we can see this shape is consistent across parameters. The reason behind the 10 layer models incredible jump, is that due to its higher layer count, the distortion of the KLD near the end of training was far higher than any other, rocketing the loss upwards. The rest followed the same trend, either stretched or compressed based on the amount of training epochs used.

We wanted to identify if specific characteristics highlighted improvements in either the label conditioning or song reconstruction. When analyzing the latent space separation found in Figure 8, we find a moderate amount of class separation; this is supported when looking at Figure 5, where we can see that the same model achieves a silhouette score of 0.22.

When varying the amount of training epochs, we observe that silhouette score increases regardless. This makes sense, as early in the training the KLD is very high, heavily influencing the encoder, and late in the training the KLD's weights approach higher values gaining more influence. The single iteration model also has a high silhouette score, we believe this is due to the single iteration meaning the encoder's latent space is more impactful compared to allowing the decoder to run multiple times, this is supported by the drop in silhouette score from the 10 iteration model. For context, every other model utilizes 5 refinement iterations by default.

When analyzing the quality of the reconstructions, we can look at classifier accuracy or relative cosine similarity. The extremely low scores of both the accuracy and relative cosine similarity indicates that the generations are very similar regardless of class. This makes sense when comparing the actual generations of the model, often consisting of middle pitch notes with intermittent rest tokens. For 12 classes in Figure 6, the random baselines can be seen clearly being greater than the majority of the model configurations. The only one that is significantly above both the top-1 and top-3 accuracy is the base model, which achieves a 10.8% and 30% accuracy respectively. Overall, these results point to the fact that our label conditioning is weak or in most cases not working at all. This can be doubly confirmed by Figure 7, which shows the relative cosine similarity between configurations. The highest value obtained is 0.010, with a standard deviation of 0.010. This near-zero value in each configuration, just like before, points to the fact that our conditional generation is not generating unique MIDI sequences for each label.

7. Conclusion

Getting this transformer-VAE to train without exploding felt like half the battle, but the numbers show we still have a lot to learn about making it musical. The "refinement" decoder kept the computation graph alive, yet the losses in Figure 3 tell their own story: reconstruction cross-entropy barely moves, CLS loss inches up as annealing gives it more weight, and KLD decreases and then rebounds around epoch 8, warping the latent space. That rebound shows up in every variant we tried (Figure 4), and the 10-layer model flat-out blew up because its KLD spike was huge.

The latent-space plots look alright: the PCA blobs are at least separable and the silhouette score lands at 0.22. Unfortunately, that separation isn't reflected in the music itself. When we asked our GRU to guess the composer of fresh samples, it managed only 10.8 % TOP-1 and 30 % TOP-3—barely above random for 12 classes. Relative cosine similarity is basically zero (0.010 ± 0.010), so the clips sound almost the same no matter which label we feed in. Listening tests back that up: lots of middle-register notes with occasional rests, but not much Bach-vs-Beethoven

Figure 9: Confusion Matrix of GRU magic.

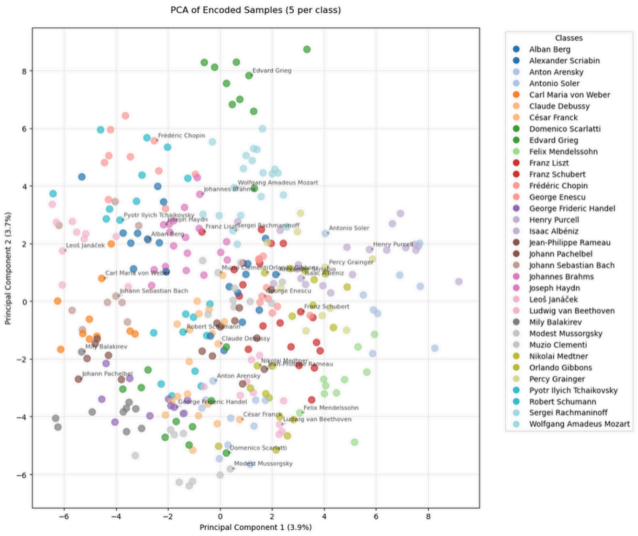

Figure 8: PCA Representation of Latent Space

Why the mismatch? The flat reconstruction loss hints that cross-entropy treats token-by-token accuracy as "good enough," ignoring local patterns and phrasing. Meanwhile, our mixture-of-Gaussians prior does nudge embeddings apart, but the decoder's five refinement passes seem to wash a lot of that signal out (silhouette drops again when we go to ten passes). The single-pass model actually clusters best—probably because the encoder's latent vector has more sway when it only gets decoded once.

So, wins and fails: Wins – model trains stably; no posterior collapse; latent clusters aren't total soup. Fails – label conditioning is weak; generated music sounds samey; deeper/longer models just amplify KLD chaos.

If we had another few weeks, first on the list would be a smarter reconstruction loss that scores motifs or n-gram structure instead of raw tokens. We'd also try fewer refinement passes, or even a lightweight autoregressive head, to keep label info from being washed out. And, yes, we'd sit down with the "middle-C orchestra" our model keeps writing and figure out why it's so afraid of high notes.

All that said, for a semester project we're satisfied: the pipeline runs, the metrics are honest, and every plot can be reproduced from the repo. Next time, we'll aim for music that sounds a bit less like a metronome and a bit more like a composer.

8. Use of Generative AI for Project

1. The following characteristics were chosen after being recommended:
    - Mixture of Gaussian's as prior distribution
    - Refinement Iterations (autoregression alternative)
    - Silhouette Score
2. Used extensively for debugging:
    - Debugging original decoders cavalcade of issues
    - Debugging MIDIVAE's forward and train loop
    - Debugging evaluation scripts.
3. Commenter for Compatible Working
    - Used for commenting on files where multiple people would work intermittently on (MIDIVAE.ipynb, Compare_Model.ipynb)
4. Used for editing parts of this report

Appendix B. Code repository

https://github.com/MIDI-Generation/midi-generation

References

1) Tudor-Constantin Pricop and Adrian Iftene, Music Generation with Machine Learning and Deep Neural Networks, ScienceDirect.com, 2024
2) B. -A. Cretu et al., "Music Generation using Neural Nets," 2022 International Conference on INnovations in Intelligent SysTems and Applications (INISTA), Biarritz, France,2022,pp.1-6,doi: 10.1109/INISTA55318.2022.9894268, ieeexplore.org
3) Hawthorne et al. "Enabling Factorized Piano Music Modeling and Generation with the MAESTRO Dataset." In International Conference on Learning Representations, 2019.
4) Huang et al. MidiFind: Fast and Effective Similarity Searching in Large MIDI databases, International Symposium on Computer Music Multidisciplinary Research, 2013
5) Tudor-Constantin Pricop, Adrian Iftene,Music Generation with Machine Learning and Deep Neural Networks, Procedia Computer Science, Volume 246, 2024, Pages 1855-1864,ISSN 1877-0509
6) A. Stais, Piano Music Generation with Deep Learning Transformer Models, Artificial Intelligence and Learning

Systems Laboratory (AILS) of the School of Electrical & Computer Engineering, National Technical University of Athens (ECE NTUA), 2023

7) https://github.com/asigalov61/Ultimate-MIDI-Classifier
8) Yadav, Prasant Singh, Khan, Shadab, Singh, Yash Veer, Garg, Puneet, Singh, Ram Sewak, A Lightweight Deep Learning-Based Approach for Jazz Music Generation in MIDI Format, Computational Intelligence and Neuroscience, 2022, 2140895, 7 pages, 2022. https://doi.org/10.1155/2022/2140895
9) S. Walter, G. Mougeot, Y. Sun, L. Jiang, K. -M. Chao and H. Cai, "MidiPGAN: A Progressive GAN Approach to MIDI Generation," 2021 IEEE 24th International Conference on Computer Supported Cooperative Work in Design (CSCWD), Dalian, China, 2021, pp. 1166-1171, doi: 10.1109/CSCWD49262.2021.9437618.
10) Yang et al. “MIDINET: A Convolutional Generative Adversarial Network for Symbolic-Domain Music Generation”. 2017 ISMIR
11) S. Arora, A. Dassler, T. Earls, M. Ferrara, N. Kopparapu and S. Mathew, "An Analysis of Implementing a GAN to Generate MIDI Music," 2022 IEEE MIT Undergraduate Research Technology Conference (URTC), Cambridge, MA, USA, 2022, pp. 1-5, doi: 10.1109/URTC56832.2022.10002181.
12) Rangapuram, Syama Sundar, et al. "Deep state space models for time series forecasting." Advances in neural information processing systems 31 (2018).
13) Goel, K., Gu, A., Donahue, C. & Re, C.. (2022). “It’s Raw! Audio Generation with State-Space Models.” 2022, International Conference on Machine Learning
14) Pasquier et al. “MIDI-GPT: A Controllable Generative Model for Computer-Assisted Multitrack Music Composition” arXiv, (2025).
15) Huang et al. “Music Transformer: Generating Music with Long-Term Structure” OpenReview, (2019).
16) Cataltepe, Zehra & Yusuf, Yaslan & Sonmez, Abdullah. (2007). Music Genre Classification Using MIDI and Audio Features. EURASIP Journal on Advances in Signal Processing. 2007. 10.1155/2007/36409.
17) Ian Simon and Sageev Oore. "Performance RNN: Generating Music with Expressive
18) Timing and Dynamics." Magenta Blog, 2017.
19) Curtis Hawthorne, Andriy Stasyuk, Adam Roberts, Ian Simon, Cheng-Zhi Anna Huang,
20) Sander Dieleman, Erich Elsen, Jesse Engel, and Douglas Eck. "Enabling
21) Factorized Piano Music Modeling and Generation with the MAESTRO Dataset."
22) iIn International Conference on Learning Representations, 2019.
23) midi2Performance GitHub Repository, 2022. https://github.com/robz/midi2performance
24) Brunner, Gino; Konrad, Andres; Wang, Yuyi; and Wattenhofer, Roger. MIDI-VAE: Modeling Dynamics and Instrumentation of Music with Applications to Style Transfer. In Proceedings of the 19th International Society for Music Information Retrieval Conference (ISMIR), Paris, France, 2018. arXiv:1809.07600.
25) Fu, et al. Cyclical Annealing Schedule: A Simple Approach to Mitigating KL Vanishing. Published in NAACL 2019. arXiv:1903.10145.
26) Lerch, et al. Unsupervised 3D Skeleton-Based Action Recognition using Cross-Attention with Conditioned Generation Capabilities. IEEE/CVF Winter Conference on Applications of Computer Vision (WACV) Workshops, 2024, pp. 211-220

Appendix A. Detailed Roles

Table 1. Team member contributions

Name	**Task**	**File names**	**No. Lines of Code**
Connor Casey	Design of model Implemented MIDIVAE class and variational encoder Wrote evaluation/analysis code Extensively debugged model Design of slides Wrote sections 1, 3.1,3.2, 3.4, 5, 6	MIDIVAE.ipynb MIDIVAE_new.ipynb Compare_Models.ipynb ...	550
Bogdan Sadikovic	Designed the GRU Classifier for music and created evaluation code for it. Worked on debugging the MIDIVAE class and model. Wrote GRU slides, and the parts of the results slides. Wrote sections 3.3 and 6 for the final report.	music_classifier.py evaluate_model.py MIDIVAE.ipynb …	493
Alex Melnick	Implemented the transformer decoder in MIDIVAE Spearheaded the end-to-end debugging of MIDIVAE generative model Wrote slides for Decoder, MIDI Background, and Motivation & Goals Wrote sections 2 and 7 and edited whole of of report	MIDIVAE.ipynb MIDIVAE_v2.pth …	550
Quargs Greene	Created tokenizer, split and annotated data Wrote datasets section of report Wrote content of MIDI Data Representation and Data Organization and Splitting slides for presentation Set up Git LFS	MIDI_preprocess.ipynb data_links.txt	485