Music Generation using Transformers with Proximal Policy Optimization (PPO)

Julia Lewandowski, Robert Carter, Noah Armsworthy, Grant Sutherland *Dalhousie University*

***Abstract*— Music is a key part of human culture, and has been with us since before recorded history. Advancements in computer generated music have seen Recurrent Neural Network models capable of polyphonic audio that is able to maintain structure and expressive timing, adding a more human quality to their songs. Where RNNs struggle is maintaining these dependencies throughout increasingly longer sequences. Another, relatively newer, sequence based model is the Transformer model. It implements an general encoder and decoder architecture, but this model stands out for our purposes by the use of its multi-headed self-attention layers. These layers allow the model to record several relative positional relationships between input tokens within the data string and build a more global song structure that can be referenced far into song generation. However, these models are most often trained from a specified corpus, and attempts to learn from a broader set of rules via reinforcement learning have been restricted to LSTM neural networks and off-policy reinforcement learning algorithms. Here, we explore the potential of using powerful transformer neural networks paired with Proximal Policy Optimization (PPO), a simpler yet powerful on-policy algorithm. We find that transformers are readily able to learn from a training corpus or from an environment through PPO, though designing a truly suitable reward function that can evaluate the quality of music is no small feat and will be the focal point of new work.**

I. INTRODUCTION

Music has captivated the human imagination since the beginning of civilization. It has persisted through time and across cultures, and as technology has continued to develop, interest in new and improved ways to create captivating music has persisted. This interest has extended into the realm of machine learning and artificial intelligence, specifically with regards to the task of artificially creating "authentic" sounding music mimicking the quality of human composers.

Because music is highly structured and frequently self-referential as it incorporates repeated segments, phrases, and motifs, music is well-suited for representation as sequential data that can be manipulated by machine learning algorithms. Multiple ways of representing musical data exist, with the most common representations being ABC Notation, wherein alphanumerical characters are used to represent notes, and MIDI, a structured format for encoding musical scores for digital instruments [1]. In this paper, we concentrate on MIDI, specifically considering the MIDI event vocabulary introduced by Oore et al. in 2018 for increased musical expressiveness [2].

Machine learning sequential approaches have been demonstrated to succeed at generating music in a style matching the corpus they are trained on [1]. One of the first attempts at using a neural network architecture to generate music was made by Peter Todd in 1989, where he applied a feed-forward neural network to implement monophonic melody generation trained on good and poor examples of melodies [3]. This model was capable of retaining pairwise correlations between notes and was implemented as a Markov model that considered only the previous melody measure during generation [3]. However, this approach suffered from limitations in terms of the lengths of sequences that could be learned and an inability to incorporate musical coherence in the form of learned "global structure" [3]. With the advent of deep learning, increased processing power and larger model architectures have allowed for significant advances in polyphonic music generation, with recurrent neural networks (RNNs) making a significant contribution to this area [1]. These have included models such as SampleRNN [4], which was capable of generating audio samples one at a time using a neural network trained on three highly varied datasets, and PerformanceRNN, which notably focused on generating music with "expressive timing and dynamics" and was trained on a corpus of classical music [5]. Both works demonstrated an impressive ability to incorporate and retain local structure and motifs, with PerformanceRNN in particular being capable of generating musical snippets with more "human-like" quality assigned to them qualitatively by a panel of listeners [5]. However, to some extent, neither were able to consistently learn and build upon long-range musical dependencies similar to previous approaches [4], [5].

In order to overcome this challenge in retaining long-term structure, we consider transformers, a deep learning model initially introduced by Vaswani et al. in 2017 capable of employing self-attention [6]. A standard transformer architecture consists of a fixed number of encoder and decoder layers, where both types of layers employ a self-attention layer and a feed-forward network [6]. In broad terms, self-attention allows for multiple representations of an input sequence to be explored by the model such that it learns the amount of attention each token should pay to other tokens of the sequence [6]. It specifically provides the benefit of incorporating long-range dependencies and global structure of the sequential data it is trained on [6]. The encoder layer focuses on learning sequence representations with attention factored in, while the decoder layer generates sequential data using these representations [6]. One significant improvement to the original transformer architecture described by Vaswani et al. was put forward by Shaw et al. in 2018 regarding an adjustment to the encoder: rather than simply relying on self-attention completely devoid of consideration of the local positioning of input sequence tokens, a transformer

model could combine both self-attention with consideration of relative positions [7]. This is a highly relevant positive adjustment for transformer based music generation applications, as incorporating such information increases the ability of such a model to benefit from both local and global structure incorporation of musical motifs, as well as relative timing of note-playing to better mimic a human performer [8]. One example of a recent transformer decoder architecture utilizing this adjustment was the Music Transformer created by Huang et al. in 2018: trained on the JSB Chorales and Piano-e-Competition datasets, it was capable of generating music building on sample inputted motifs as well as generating accompaniments in a polyphonic style [8].

Interestingly, previous work demonstrated the capacity of $\Psi-$, G-, and Q-learning in training an LSTM model to generate music [9]. The *Note RNN* was initially trained using 30,000 MIDI songs, with reinforcement learning used to fine-tune the model [9]. The fine-tuning process, with any of the RL algorithms improved the model's compositions substantially. Having seen the capacity of RL in this domain, we choose instead to use an on-policy algorithm, known as Proximal Policy Optimization [11]. Music generation stands to benefit from PPO due to its comparably superior efficiency and speed, while being a more stable algorithm than other on-policy options [11]. As an actor-critic method, it does require the training of two models, a separate actor and critic network. The former dictates the action done while the latter is needed to evaluate the actor's choices.

This project represents the first known attempt to combine a music transformer decoder architecture with reinforcement learning for more guided music generation. We describe the transformer architecture and reinforcement learning setup used in the following Methods section, demonstrate the viability of both components in our Results, and consider future work to develop this framework further in the Discussion.

II. METHODS

A. Dataset and Preprocessing

In order to teach the model how to cohesively string notes together, it needed to be provided with many examples of music scores broken down into strings of notes. For this task, we used the Maestro Dataset which contains 200 hours of piano music from 1276 songs in MIDI file format [10]. A maximum sequence size of 2050 tokens was selected for all sequences given that this length was used demonstrated by Huang et al. as sufficient for training purposes while balancing hardware limitations [8]. As such, Maestro pieces longer than this were split such that they were each 2050 tokens or less.

Rather than direct audio data, MIDI files contain messages that outline song instructions for the chosen output device/software. These messages include information about a song's tempo, time signature, instrument type, and notes. This dataset was chosen because each song contained only a single piano track which would be simple to break down, analyze, and compare the training and generated samples. Note messages for piano contain data for a note's frequency, the time a note is held for, and the velocity or 'loudness' with which it is played. The vocabulary for these messages can be broken down into individual events as described by Oore et. al, 2018 [2] and represented by the following string tokens:

- 128 "note_on" events (vocabulary indices [1 ... 128])
- 128 "note_off" events (vocabulary indices [129 ... 256])
- 125 "time_shift" events #time shift = 1: 8ms (vocabulary indices [257 ... 381])
- 32 "velocity" events (vocabulary indices [382 ... 413])

With addition of "<pad>", "<start>", and "<end>" tokens located at indices 0, 414, and 415 respectively. By using these tokens and their indices, the embedding, attention and decoding can be implemented as a language model. The training songs were then processed by their MIDI messages that contain several parameters into a corresponding sequence of individual events.

To implement the matrix multiplication required in the self attention mechanism described below in section D, we needed to ensure proper dimensionality to pair with with their Query, Key, and Value matrix. This added extra importance to the reduction of each varied length song sequences in to smaller equal sized sequences. The data was cut up into portions of 2050 events, padded with the <pad>tokens if necessary, and given encapsulating <start>and <end>tokens.

B. Masking

Two masking layers were implemented and combined for use by the transformer. First, a padding mask function was required in order to account for padding added to sequences shorter than 2050 tokens during the preprocessing phase. This prevented the model from treating padding in the affected subset of sequences as input. The generated padding mask was the same shape as the inputted tensor, capable of being broadcasted to n dimensions as needed, and was represented as a one-hot encoded vector where ones corresponded to indices consisting of padding that required masking.

The second masking function involved the creation of a look-ahead mask relevant for use during the model's musical generation phase. This was implemented with regards to the scaled dot-product attention functionality discussed in greater detail in Section D, where the model was prevented from viewing tokens past the current index it was at during its iteration using an upper triangular mask, limiting it to considering only the current token i and past sequence of tokens $[0, i-1]$ when generating the $(i+1)$th token [8].

These two masking functions were combined for a final masking functionality, wherein the maximum of the two masks formed the final mask. This simultaneously preserved the look-ahead mask in addition to retaining the padding mask indices, such that the final mask vector contained a value of 1 for each index the model was prevented from viewing.

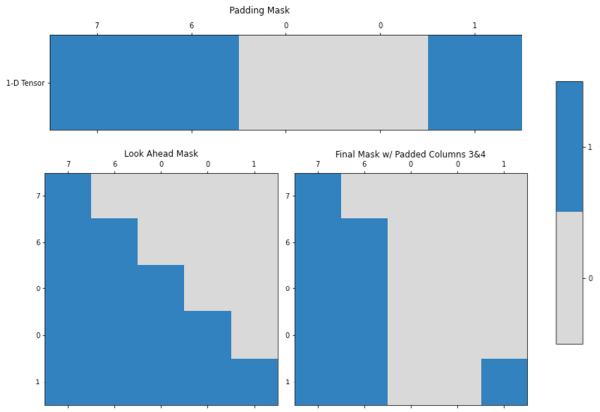

Fig. 1. Depiction of the masking process for an example 1-D tensor t = [7, 6, 0, 0, 1]. Masked indices are coloured in grey, while indices containing indices viewable to the model are depicted in blue. In the padding mask (top), indices 2 and 3 containing padding are masked. In the look-ahead mask (left), the tensor is represented as a 2D tensor where the initial upper triangular mask applied is viewable. The final mask (right) combines the padding and look-ahead masks, with the padding mask applied to the column tensor only of the 2D representation.

C. Positional encoding

As transformers lack convolution or recurrence to keep track of historical features, positional encoding is a key feature of their architecture originally introduced by Vaswani et al. that allows them to retain information about the relative positions of their inputted sequence tokens [6]. This allows transformers to incorporate order based information, or token context. In this project, the original sine (Equation 1) and cosine (Equation 2) functions used by Vaswani et al. were employed due to their demonstrated ability to adapt to "sequence lengths longer than the ones encountered during training" [6] which is highly of value for generation tasks.

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}}) \tag{1}$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}}) \tag{2}$$

In the given equations, pos represented the integer position of the token, i corresponded to its dimension, and d_{model} corresponded to the number of hidden dimensions in the transformer [6]. Using this positional encoding representation, each dimension in the encoding corresponded to a sinusoidal function and captured the relative positions of the music notes, where the sine function was applied to even indices and the cosine function was applied to odd indices [6], [8]. The positional encoding layer was the same size as each embedding, and was used by the model to learn how notes in the given input data related to one another.

D. Relative Scaled Dot Product and Multi-head Attention

Self-attention is one of the main advantages presented by transformers by comparison to other deep learning models: alongside providing a high potential for parallelizability, self-attention allows for the model to learn long-range dependencies [6]. This is highly desirable in music generation as previously discussed due to the importance of musical coherence. Challenges met by RNN-based music generation implementations in incorporating consistent musical motifs and a piece's long-range structure that can be addressed using self-attention [1].

The initial concept of scaled dot-product attention was introduced by Vaswani et al., where an attention function (Z_h) was first conceptualized as "mapping a query and a set of key-value pairs to an output" [6] as given below in Equation 3:

$$Z^h = Attention(Q^h, K^h, V^h) \tag{3}$$

Q^h, K^h, and V^h represent the queries, keys, and values each indexed by h corresponding to the number of heads in the sublayer and each consist of DxD vectors where $Q = XW^Q$, $K = XW^K$, and $V = XW^V$ [6]. The scaled-dot product attention of the transformer decoder calculated the vector softmax output for each h head as given in Equation 4 below, which is the modified scaled-dot product attention introduced by Huang et al. [8] building off of Shaw et. al's [7] modifications:

$$Z^h = softmax(\frac{(QK^T + S^{rel})}{\sqrt{D_h}})V^h \tag{4}$$

In this equation, S^{rel} represents the LxL logits matrix created by the interaction of the relative embeddings with the queries in the softmax as described by Shaw et al. [7], and is obtained by "skewing" this original QE matrix according to the skewing procedure described by Huang et al. [8]. The skewing operation allows for the relative distance terms to be added to the overall values in the query-by-key (QK) matrix, allowing the model to make substantially better use of relative positioning of music notes [8]. The final output of the multi-head attention layer is an attention vector consisting of the concatenated softmax values computed for each head.

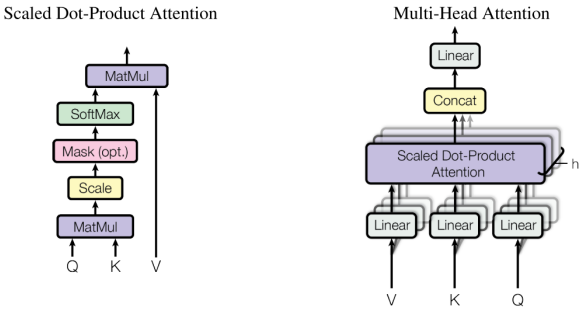

Fig. 2. Figure 2 from Vaswani et al. 2017 [6] depicting scaled dot-product attention (left) and multi-head attention architecture incorporating multiple attention layers (right).

E. Feedforward Network

The feedforward network acted as the secondary component of the Decoder, and immediately followed the multi-head attention layer. It consisted of two Linear layers with a ReLU activation function between them, with the network's

operations on the input sequence X denoted by Equation 5 below [8]:

$$FF(X) = ReLU(XW_1 + b_1)W_2 + b_2 \tag{5}$$

As given above, W_1 and W_2 correspond to the weights of the first and second layer, and b_1 and b_2 correspond to the biases of the first and second layer.

F. Decoder

Each decoder layer of the transformer combined a multi-head attention sub-layer with a feedforward network. These were followed by a normalization layer for added regularization, as its inclusion has been shown to reduce the training time of the feed-forward network layer [6]. While a single positional encoding layer was used for all model architecture configurations, the number of decoder layers could be varied to modify the number of consecutive decoder layers used before the final Linear output layer of the model.

G. Transformer Architecture Hyperparameters

A summary of all hyperparameters influencing the model architecture can be found below in Table I. A number in brackets following a parameter description denotes that the value was fixed during trials. The primary hyperparameters that were experimented with were the number of layers in the decoder block in the range [1, 2], the number of attention heads per decoder multi-head attention layer in the set [4, 8], and the learning rate of the Adam optimizer for training in the set [0.001, 0.01, 0.05]. This represented the maximum range we could reasonably test given hardware limitations, and in fact, it was only possible to train the 2-layer architectures with 4 attention heads at most due to these constraints.

TABLE I

SUMMARY OF TRANSFORMER DECODER MODEL ARCHITECTURE HYPERPARAMETERS.

Hyperparameter	Description
num_layers	Number of stacked decoder layers
hidden_dims	Hidden dimensions (256)
ffn_dims	Feed-forward network dimensions (512)
num_heads	Attention heads per multi-head attention sublayer
max_rel_dist	Max input sequence length (2050)
max_abs_position	Max absolute position for positional encodings (1)
epsilon_norm	Epsilon value for normalization layers ($1e^{-6}$)
vocabulary_size	Number of MIDI events (416)
alpha	Adam optimizer learning rate during training

H. Auto-regressive music generation

Music generation involved using a transformer decoder model trained on a smaller subset of the preprocessed Maestro database (typically 2000 - 4000 sequences of length 2050) to generate tokens using a greedy strategy. The decoding function of the model was used to autoregressively compute outputs when given a prompt list of MIDI events according to Oore et al.'s vocabulary, where for decoding note i, the model could consider notes in the range [0, $i-1$] while being shielded from future input by the application of the look-ahead mask to the data [8]. This smaller seq2seq like setup was used for generating the results, although hypothetically the model is able to generate music from scratch using its learned positional embeddings. The implementation of the generation function was highly referenced based on Huang et al.'s work [8], [12]. The tempo was fixed at 512820, and the temperature set at 1.0 consistently, where the temperature can be understood as being a measure of how diverse the decoder output could be [8].

I. Proximal Policy Optimization

The goal in PPO is to maximize a surrogate objective function (Equation 6) [11].

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}[min(r_t(\theta)\hat{A}_t, \\ clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \tag{6}$$

Where $\hat{A}_t$ is the advantage, or how much better/worse an action was than anticipated at a time-step and r_t is the reward. The use of the minimum value and clip function is to ensure that dangerously large leaps are not made, which is possible when advantage is sufficiently large. The clip restricts the effective range of rewards in the second term, based on the interval ϵ. We set ϵ to 0.2. We compute in batches every *T=20* timesteps, which corresponds to choosing 20 MIDI events, and do 3 iterations of learning after each 'episode' (or, each time we finish composing a song). Advantage is computed according to Equation 7 [11].

$$_t = -V(s_t) + r_t + \gamma r_(t+1) + ... \\ \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_T) \tag{7}$$

Where $V(s_t)$ is the value provided by the critic at time step t and γ is a discount factor, set to 0.95.

To refine our model using PPO, we initialize both the actor and critic to have the same layers as the pre-trained model. However, the critic's output layer must be changed to have an output with a cardinality of 1. This does require the critic to engage in some learning to refine the weights of the output layer, but otherwise ensures it will have the same power to interpret music as the actor. During each learning step, we use stochastic gradient descent based on the gradient computed by Equation 6.

J. Environment

The environment we wrote for the reinforcement learning process is quite simple. It is written in the style of OpenAI Gym environments in order to make it easier to use. The environment tracks the song as it has been constructed so far, and the number of MIDI events added to it. Each action step, it returns both the reward and the next observation, which is simply the song so far, with reward described in section K below.

K. Reward Function

The reward function assigns a score to each generated event as it is added to the song. The reward function takes the event and the sequence of events before it as parameters. It uses rules based on musical theory and the general structure of Midi files to analyze how well the added event fits into the song. Since our function which converts event lists into Midi files assumed that the song was written in the key of C with a 4/4 time signature and a tempo of 512820, the reward function assigns high scores when the events are relevant to these assumptions.

In music, a single note is often meaningless. Only a combination of notes can carry any significant meaning. Therefore, it is essential to compare every added event with the events which came before it.

"note_on" events have the most complex reward logic. Whenever a "note_on" event is added to the song, the function first checks if the note is on-key. Since we reward songs based off the standards of the key of C, the function checks if the note is in the C major scale which includes the notes C, D, E, F, G, A and B. The function then checks the number of notes that are currently playing (on a piano this would be keys which are currently being held in unison). This is achieved with a helper function. If no other notes are currently playing, the function simply checks if the added note creates a second consecutive leap (more then one tone between two consecutive notes). This is undesirable in music. Leaps can add be a tool for dramatic variation if used sparingly, however consecutive leaps make songs sound disorganized. If multiple notes are playing at the same time, the functions checks if the notes make up an interval or chord which is commonly used in the key of C. The function rewards these chords and intervals with a score of 10. To avoid the sparse reward problem, small rewards are returned if the added note adheres to the key of C, even if the note creates a consecutive leap or an uncommon interval.

The reward function rewards "note_off" with a score of ten when there is a matching "note_on" event before it and there is at least one "time_shift" event between them. As a way to deal with sparse reward, a smaller score is given to "note_off" events which have a matching "note_on" event but no "time_shift" event in between. In this case, the note will not be heard by the listener.

In 4/4 time, there are four beats in every measure and one beat is equal to one quarter note. At the specified tempo, one tick is equal to 1 millisecond of time and 480 ticks is equal to one beat. This means that one measure is equal to 1920 ticks. Since "time_shift" events are used to give timing to the generated song, all "time_shift" events should fit perfectly into 4/4 time. When a "time_shift" event is added to the sequence, the reward function checks if it is a valid length of time. We know that it's a valid length of time if it is divisible by four. If it is not, a score of zero is returned. If the added "time_shift" event is a valid length of time, a score of up to ten is returned. To avoid the trap of our creating a model which adds relevant "time_shift" events endlessly, the reward function checks if there are multiple "time_shift" events in row and returns a lower score if the aggregated time is longer than four measures. This amount of time was chosen arbitrarily, but it is very rare in music that a note or silence is held for longer than four measures. A score of ten is returned when the aggregated time is less than four measures. Since our model was originally trained using data generated from human performances, perfect timing is impossible. To address this issue, a margin of error was added the previously specified time constraints. This allows the reward function to reward "time_shift" events which sound good to the human ear, but are not perfect.

The reward function rewards "set_velocity" events with a score of zero if multiple "set_velocity" events take place in a row. This avoids the trap of the model producing endless "set_velocity" events. When there is more than one note playing at the same time, they should all carry the same velocity. This way, no note is overpowering another and the listener can hear all notes clearly. So, the reward function rewards "set_velocity" events with a score of ten when the velocity does not overpower any other notes and a reward of zero if it does.

Generally, songs end with the tonic note. This gives the listener a sense of resolve. Because of this, the reward function rewards the "<end >" condition with a score of ten if the last note of the song was a tonic note (C). This is the ideal condition. To avoid sparse reward, a smaller reward is given if the last note was not the tonic note but it was on-key. A reward of zero is given if the song ended without playing any notes.

Theoretically, the reward function could become endlessly complex for the purpose of generating more complex music. With further work, the reward function could be expanded and used to create more music with greater artistic variety.

III. RESULTS

A. Results of a Baseline Test

As a simplified test of whether it is possible to train a transformer using PPO, we used a mock environment where songs could be comprised of two notes. This was paired with a mock reward function which penalized continuous repetition of notes while rewarding alternation. That is, to maximize the objective function, the agent would need to constantly alternate notes. We observed a rapid increase in reward, with the model eventually outputting an optimal sequence several hundred notes long. This obviously does not demonstrate an ability to learn the full task music generation, but it does at least suggest our approach has potential and validates our implementation of both PPO and the transformer network.

B. PPO with Reward Function

Using PPO and our reward function to refine a transformer network demonstrated that the network was already fairly proficient in optimizing reward, but exploration over 10,000 episodes led to a stable optima as shown in Figure 3.

Fig. 3. Average Reward in Each Episode over 10000 Training Episodes. As PPO explores, the average will vary substantially vary until by the end it has converged on an optimal policy

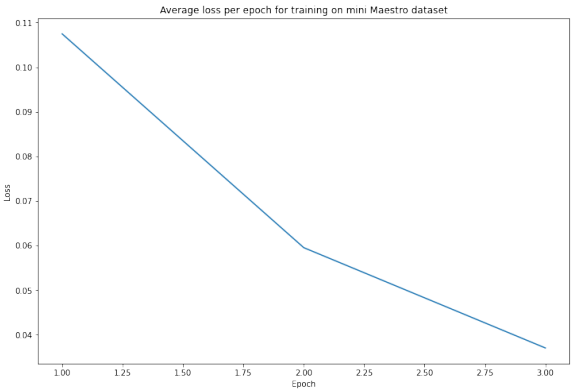

Fig. 4. Loss of the transformer model consisting of 2 decoder layers, 8 attention heads per multi-head attention layer, and learning rate of 0.01 over 3 epochs on the mini Maestro dataset.

C. Model and Generated Music

The hyperparameter search procedure described earlier revealed practically negligible differences between the varying model architectures. It is likely that a larger architecture, especially one combining a greater amount of attention heads, would have led to more differentiated results being seen in terms of the loss function's minimization. However, the model architecture that performed best on the training and validation sets was one with 2 layers, 8 heads, and a learning rate of 0.01. Its training loss curve for 3 epochs is depicted below in Figure 4.

Three pieces were selected for use as input data to "inspire" the music generation process of the transformer model. These included *allegro grazioso*, a simple warmup tune, as well as Bach's Prelude in C and Chopin's Etude in C. The output data was generated with the assumption that it would be in C scale, with a fixed tempo of 512820, and temperature set to 1.0. Links to the input and output songs in WAV format can be found in Table II below.

TABLE II

URLS TO WAV FILES OF THE INPUT SONG PROMPTS USED WITH THE FINAL MODEL AND TO THE OUTPUT SONGS GENERATED BY THE MODEL.

Song	Input URL	Output URL
Allegro grazioso	https://bit.ly/37CBgeg	https://bit.ly/3ODCINS
Prelude in C	https://bit.ly/3k7Mry9	https://bit.ly/3ODS058
Etude in C	https://bit.ly/3Lk8E7S	https://bit.ly/3OE2jXd

IV. DISCUSSION

The primary focus of this project was to demonstrate the viability of combining a transformer architecture for music generation with reinforcement learning to guide the music generation process more intentionally. It is clear that PPO was able to influence the behavior of the network, and we have compositions from transformers trained with and without PPO. In either case we succeeded in generating music, to a promising degree of quality. We do unfortunately lack an example where a network generated a composition both pre- and post-reinforcement Learning so we cannot make a direct comparison. This is a natural means to improve on the quality of the results we've established so far.

Future work will involve increasing the size of the data the transformer models are trained on, as due to hardware and GPU limitations, the described complete architecture including PPO could only train on a smaller subset of the Maestro dataset with 2072 instances of musical segments and the batch size was constrained to a size of 8. Additionally, during the generation process, more tempos and temperatures could be systematically experimented with to measure the variation of model compositions.

The reward function could be improved in several ways with future work. As previously mentioned, the reward function could theoretically become endlessly complicated for the purpose of producing more complex music. Further steps which can be taken are listed below.

- Rewards for "note_on" events could become more complex by increasing the maximum number of notes which can play concurrently while still earning a reward. For example, if there is already three notes playing which form a chord, a reward could be returned for adding a fourth note which is also a member of that chord but in another octave. This could give the illusion of a human using both hands to play at the same time.
- Rewards for "set_velocity" events could become more complex by rewarding slow builds in velocity over consecutive "set_velocity" events. This would result in a crescendo effect which is a common musical technique which is used to build tension and energy as the melody is coming to a climax.
- Rewards for "time_shift" events could be assessed by placing silences into bins and rewarding or penalizing certain distributions. This would allow the function to reward silence distributions which are concentrated around a period of silence that is deemed appropriate while still having a small number of short and long

silences for artistic variation. This is better than hard cut off of measures which was previously specified.

This idea could also be extended to note distributions. For example, a song may use the tonic note most frequently while using other notes in a more reserved manner.

V. ACKNOWLEDGEMENTS

We would like to express our deepest gratitude to our professor, Sageev Oore, both for being a wonderful professor this term who motivated us to want to pursue this project and who provided us with additional support, guidance, and invaluable feedback throughout the process. Without him this project would not have been possible.

REFERENCES

[1] J. P. Briot, "From artificial neural networks to deep learning for music generation: history, concepts and trends," Neural Comput. Appl., vol. 33, no. 1, pp. 39–65, 2021, doi: 10.1007/s00521-020-05399-0.

[2] S. Oore, I. Simon, S. Dieleman, D. Eck, and K. Simonyan, "This time with feeling: learning expressive musical performance," Neural Comput. Appl., vol. 32, no. 4, pp. 955–967, 2020, doi: 10.1007/s00521-018-3758-9.

[3] P. M. Todd, "A connectionist approach to algorithmic composition," Comput Music J 13(4):, pp. 27–43, 1989, doi: https://doi.org/10.2307/3679551

[4] S. Mehri et al., "Samplernn: An unconditional end-to-end neural audio generation model," 5th Int. Conf. Learn. Represent. ICLR 2017 - Conf. Track Proc., pp. 1–11, 2017.

[5] Ian Simon and Sageev Oore. "Performance RNN: Generating Music with Expressive Timing and Dynamics." Magenta Blog, 2017. https://magenta.tensorflow.org/performance-rnn

[6] A. Vaswani et al., "Attention is all you need," Adv. Neural Inf. Process. Syst., vol. 2017-December, no. Nips, pp. 5999–6009, 2017.

[7] P. Shaw, J. Uszkoreit, and A. Vaswani, "Self-attention with relative position representations," NAACL HLT 2018 - 2018 Conf. North Am. Chapter Assoc. Comput. Linguist. Hum. Lang. Technol. - Proc. Conf., vol. 2, pp. 464–468, 2018, doi: 10.18653/v1/n18-2074.

[8] C.-Z. A. Huang et al., "Music Transformer," pp. 1–13, 2018, [Online]. Available: http://arxiv.org/abs/1809.04281.

[9] N. Jaques, S. Gu, R. E. Turner, and D. Eck, "Tuning Recurrent Neural Networks With Rl," 5th Int. Conf. Learn. Represent. ICLR 2017 - Work. Track Proc., pp. 1–12, 2017.

[10] C. Hawthorne, et al., "Enabling Factorized Piano Music Modeling and Generation with the MAESTRO Dataset." In International Conference on Learning Representations, 2019. [V3.0] Available: https://magenta.tensorflow.org/datasets/maestro

[11] J. Schulman, et al., "Proximal Policy Optimization Algorithms.", arXiv., 2017

[12] Spectraldoy, "Spectraldoy/musictransformertensorflow: Tensorflow implementation of Music Transformer for training on TPU," GitHub. [Online]. Available: https://github.com/spectraldoy/MusicTransformerTensorFlow. [Accessed: 20-Apr-2022].