Listener-Adaptive Music Generation

Sterling Alic
Department of Computer Science
Stanford University
salic@stanford.edu

Christopher Wolff
Department of Computer Science
Stanford University
cw0@stanford.edu

Abstract

We envision a future in which people will be listening to an infinite, automatically generated stream of music that is perfectly attuned to their music taste. This project is meant to be a step in that direction. We propose an algorithm for training generative models of music that can adapt to listener feedback. Our approach consists of two phases. First, we train a recurrent neural network on an existing dataset of songs. Second, we iteratively present a listener with generated sequences, and use the feedback to improve the model. We evaluate our approach in a listening test, and show that the feedback training improves the music generated by the model ($p \approx 0.11$).

1. Introduction

Music generation is a challenging problem that requires reasoning over long sequences in order to attain coherent musical structure. Various deep learning-based approaches to this problem have previously been proposed. Long short-term memory networks (LSTMs) [1] are able to deal with long-range dependencies by maintaining a cell state that can retain information over many time steps. Music Transformer [2] and MuseNet [5] make use of self-attention, and demonstrate that the transformer architecture [8], which has seen much success in the natural language processing domain, is also well-suited for music generation. MusicVAE [6] uses a variational autoencoder to learn a latent distribution of note sequences, and then samples from this distribution to create new music.

Our focus is not on algorithmic or architectural advancements, but instead on a novel training procedure that involves the listener. Our intuition is that listener feedback may be able to guide the model to generate sequences that the listener liked and prevent it from generating ones that the listener didn't like, similar to how a reinforcement learning agent adapts to signals from its environment. Furthermore, this approach may allow a generative model to adapt its outputs to an individual's musical preferences. RL Tuner [3] takes a similar approach and uses an imposed reward function to refine a sequence predictor with deep Q-learning. In contrast to our work, their reward function is based on expert heuristics rather than human feedback.

2. Dataset

We use the POP909 dataset [9], which consists of piano arrangements of 909 popular songs. Originally, each arrangement is a MIDI file, with a duration between 90 to 300 seconds. Figure 1 shows an excerpt of such a file in a piano roll view.

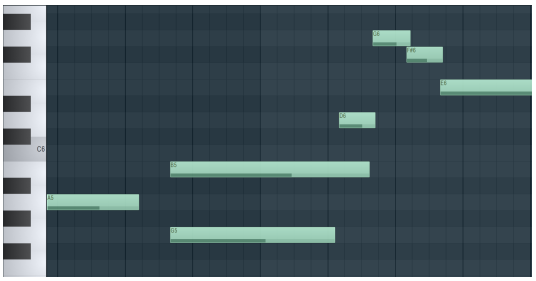

Figure 1: A piano roll representation of a snippet from a MIDI file. The horizontal axis captures time, and the vertical axis captures the pitch of a note. The music is polyphonic; that is, multiple notes can play at the same time.

We adopt an event-based representation similar to [2, 7] and convert each file to a sequence of musically meaningful tokens. These tokens represent:

- 128 **note-on** events, indicating the start of a new note.
- 128 **note-off** events, indicating the release of a note.
- 100 **time-shift** events, ranging from 1ms to 100ms.
- 32 **velocity** events, indicating how forcefully the note is played.

The note-on and note-off events span the entire range of MIDI notes, and the time-shift events are granular enough to capture subtle timing nuances. Hence, aside from minor time discretization errors, this representation is a lossless

transformation of the original file. During training and inference, each token is mapped to a unique integer between 0 and 387. We randomly split the dataset into three subsets: 90% for training, 5% for validation, and 5% for testing. In order to benefit from vectorized GPU operations, we split each song into sequences of length 1024, and pad the final part of each song with a special token to fit this length.

3. Methods

We formulate the music generation task as a supervised learning problem. As is common for language models, we factorize the joint likelihood of an event sequence as a product of conditional probabilities.

$$P(E_{1:N}) = P(E_1) \cdot P(E_2|E_1) \cdot ... \cdot P(E_N|E_{1:N-1})$$

Our goal is to learn the distribution of the next event given all previous ones using a sequence-to-sequence model. Once we have such a model, generating an entire sequence reduces to iteratively sampling from the probability distribution of next events. We begin by training a model to maximize the likelihood for the event sequences in our training dataset. Second, we repeatedly present a listener with examples of generated melodies, receive feedback, and then use this feedback to improve the model.

3.1. Training a generative model

We experiment with two model architectures: an LSTM and a transformer. Both are illustrated in figure 2. We chose these architectures because they are known to be capable of modeling long-range dependencies and showed promising results in recent work [2, 5]. In each case, the network input can be viewed as a sequence of one-hot vectors, and the labels are one-hot vectors corresponding to the next event at each step. In other words, the labels are the same as the inputs, but shifted one time step into the future. During training, we use gradient descent via adaptive moment estimation (Adam) [4] to minimize the cross entropy between the true labels and the predicted labels, averaged over the sequences from the training set.

$$\ell_{\text{train}}(y, \hat{y}) = -\frac{1}{N}\sum_{i=1}^{N} y_i^T \log \hat{y}_i$$

where N is the length of the sequence, y_i denotes the true label vector for event i, and $\hat{y}_i$ denotes the predicted label vector for event i.

The main difference between the two architectures is how they compute $\hat{y}$. The LSTM is a recurrent neural network that takes one token at a time as an input, and computes the next token as a function of that input, a hidden state, and a cell state. Intuitively, the cell state has a similar functionality as the human memory. At any point, the model can choose to forget its entries and update it with new values. It also helps to prevent vanishing gradients. In contrast, the transformer uses self-attention to compute the next token as a function of the entire input sequence and a positional encoding of the current input. It assigns an attention weight to every token in the input that indicates how relevant that token is for computing the output. Future tokens are masked so that they cannot be taken into consideration. This self-attention layer is followed by a fully connected layer and layer-wise normalization. This stack is repeated several times and then followed by final fully connected layer to create the transformer decoder architecture. In contrast to the original transformer architecture, we only use the decoder part of the network.

In order to generate a new event, we draw from the distribution

$$P(\cdot) = \frac{e^{z/T}}{\sum_i e^{z_i/T}}$$

where z are the logits of the model and T is a temperature parameter that controls the entropy of the resulting distribution. A high T results in higher stochasticity in the event distribution. We experimented with various value of T, but found that $T \approx 1$ worked best.

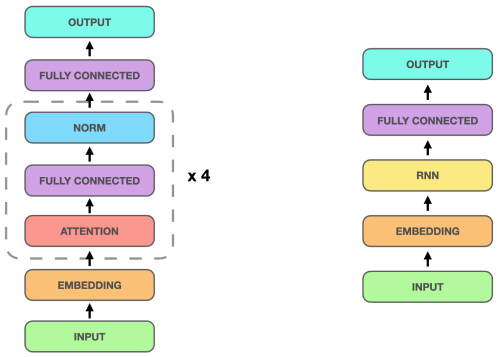

Figure 2: An illustration of our model architectures. The left pipeline shows the transformer and the right pipeline shows the LSTM.

3.2. Improving the model with listener feedback

In order to adapt the generated music to the preferences of the listener, we propose a feedback loop that iteratively re-trains the model using sequences it generates and presents to the listener. More specifically, we start by presenting the model with a priming sequence, and generating a piece of music using the initial model parameters learned in the first phase. Then, we allow the listener(s) to hear the generated song, and respond with a reward signal r_t at arbitrary times t, where t is an index for the events in the sequence. The times t are chosen voluntarily by the listener. The rewards r_t

correspond to their satisfaction with the generated sequence right before time t. For instance, if a listener enjoyed a sequence they heard, they might respond with with a reward of $+2$ at that time. The rewards take values in the set $\{1, 2, 3\}$, and there is no limit to the amount of feedback we allow the user to give.

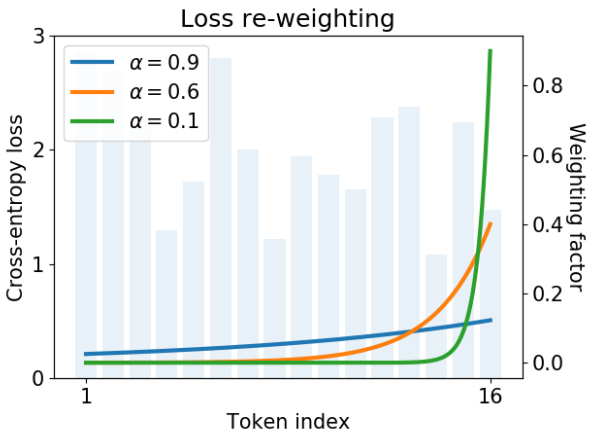

Figure 3: Examples of loss re-weighting for a sequence of length 16 and various values of α. A small α results in a weighting skewed towards the last tokens in the sequences, and a large α weights the tokens more uniformly.

For each (t, r_t), we then create a tuple $(E_{t-K:t}, r_t)$ that stores the last K events before time t along with the corresponding reward. These tuples are stored in a new dataset $\mathcal{D}_{\text{new}}$. Next, we train the model using gradient descent, starting from the most recent parameter estimates. For the original samples, we continue using the standard cross-entropy loss function. However, for the newly added samples, the question of credit attribution arises. Which notes are responsible for the reward signal? Intuitively, notes closer to the time of the signal should have a larger effect on the listener's decision to reply with a reward. Thus, we propose the following loss function:

$$\ell_{\text{new}}(y, \hat{y}) = -r \sum_{i=1}^{K} w_i y_i^T \log \hat{y}_i \quad w_i = \left(\frac{\alpha - 1}{\alpha^K - 1}\right) \alpha^{K-i}$$

This is essentially an exponentially weighted average of the cross-entropy loss at each time step of the sequence, with higher weights for events close to time t. The hyperparameter α controls the extent of this weighting. The normalizing factor for w_i ensures that the weights sum to one. Figure 3 shows the weight magnitudes for various values of α. The loss is also weighted by the reward r for the sequence. The intuition here is that we want to increase the probability of events from sequences with larger rewards more so than of those with smaller rewards. Finally, the total risk is a weighted average of the risk for the samples from the original training set and those from the new dataset, controlled by a weighting factor $\lambda \in [0, 1]$ that controls the relative importance of each dataset.

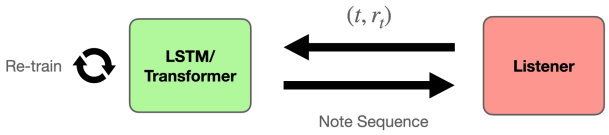

Figure 4: An illustration of our proposed training algorithm.

This process of generating new music, receiving feedback, and augmenting the training set, is then repeated for an arbitrary period of time. In order to ensure that the dataset does not grow arbitrarily large, we purge the least recently added sequences once $|\mathcal{D}_{\text{new}}| > d_{\max}$. An illustration of the algorithm is shown in figure 4.

4. Experiments and Discussion

4.1. Model architecture comparison

We start by training two initial models – one LSTM and one transformer. The LSTM has a single layer, a hidden state dimension of 512, and a total of about 1.9M parameters. The transformer consists of four decoders and has about 1.5M parameters. We use mini-batch sizes of 256 and 8 for the two models, respectively, as these were the largest sizes that fit into our GPU memory. Both architectures use an embedding layer for the inputs of dimension 256 and Adam with a learning rate of 0.001 and parameters $\beta_1 = 0.9$, $\beta_2 = 0.999$[1]. To prevent the gradients from exploding, we clip them at a threshold of 1. We then train each model for 200 epochs.

Model	Train PPL	Test PPL
LSTM	6.11	10.51
Transformer	6.93	9.30

Table 1: A comparison of the models' perplexity measures for seen and unseen songs.

To compare the two architectures, we evaluate them both quantitatively and qualitatively. First, we measure their perplexity on our test set. Table 1 shows our results. We can see that while the LSTM is able to fit the training data more closely, the transformer does a better job at generalizing to unseen data. Next, we conduct a listening test to compare the models subjectively. We prime the models with 10 sequences from our test set, and prompt them to generate a continuation

[1]We also trained a deeper network with 2 layers and an embedding dimension of 64, and found that the resulting music was not as pleasant as that generated by the shallow LSTM. For brevity, we do not include it here.

of 1024 tokens each. This corresponds to roughly 30 seconds, depending on the exact tokens. Then, we convert the outputs to MIDI files, and ask participants which one they prefer. We also include the real continuation for comparison. Figure 5 shows our results. We conclude that neither model comes close to the real songs. Surprisingly, the LSTM generates better-sounding continuations despite its higher perplexity. Hence, we decide to discard the transformer and only use the LSTM for the feedback-training phase. We

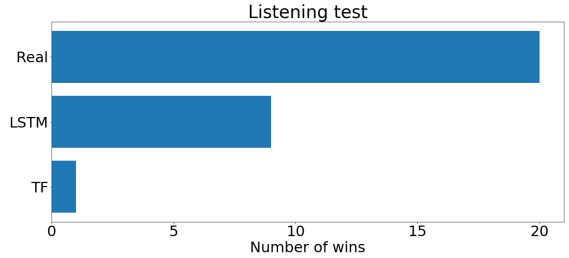

Figure 5: A qualitative comparison of generated continuations from priming sequences. The plot shows the number of wins in pairwise comparisons between the models.

found that the LSTM learned many basic principles of music theory, such as playing in key, maintaining tempo, and playing chords. On the other hand, the transformer outputs often sounded dissonant and incoherent. Both models seemed to steer away from the musical style of the priming sequence quite quickly, as the continuations often sounded unrelated to it. Additionally, we believe to have recognized some of the sequences the LSTM generated, suggesting that it may have memorized some sequence chunks from the training set.

4.2. Feedback training

Next, we use our proposed feedback-training procedure to improve the LSTM we trained. In each iteration, we sample a priming sequence from our test set, generate a continuation, and respond with zero to five (t, r_t) pairs. For each pair, we extract the last $K = 128$ generated tokens, and append them to a new dataset. After collecting 32 new samples, we use the algorithm outlined in the previous section to perform 16 additional epochs of gradient descent, with parameters $\alpha = 0.99$ and $\lambda = 0.5$. We repeat this process three times, for a total of 96 feedback signals and 48 training epochs.

To evaluate the effectiveness of this procedure, we compare the initial LSTM before feedback-training to the final one in a blind listening test. We ask participants to evaluate 25 continuations of priming sequences from our test by stating which of two continuations they prefer, without telling them which is which. We found that the listeners prefer the new model **16 of out 25 times**. By treating the experiments as Bernoulli trials, we can conclude that the sequences from the new model are better with a p-value of approximately 0.11. Subjectively, we found that the music generated after feedback training contained slightly more interesting musical motifs and coherent melodic sequences. However, the sequences from the two models do seem very similar, and it's not obvious to us in what way the additional training improved the model. We believe that a possible explanation for the improvement is that the model learned to overfit to the feedback sequences. Since we're using a relative weighting between the datasets of $\lambda = 0.5$, the few samples from the feedback dataset contribute greatly towards the overall loss. This means that the model has an incentive to learn from these samples and adapt its parameters to generate similar ones. If similar sequences are generated during the tests, it's likely that the listener will again respond positively, even if the model is just repeating what it has seen.

5. Conclusions and Future Work

In summary, we trained two models to generate piano music using a dataset of popular songs. We found that while a transformer-based architecture is able to generalize to unseen data better, an LSTM architecture generates sequences that are preferred by listeners. We then showed that rewards from a listener can be used to improve the model's performance by constructing new training samples and re-weighting the loss function by the reward signal and a weighting factor that allows for credit attribution.

In the future, we would like to scale up the feedback training and potentially collect several thousand, if not more, feedback signals. This should allow us to explore whether our approach really does refine the generative model to a listener's preferences. Alternatively, we may make the model publicly available in order to receive feedback on an even larger scale. Furthermore, we are interested in learning automatic reward signals. It's unclear how and why exactly we prefer some music to others, but perhaps it's possible to learn a model that can predict whether a certain sequence sounds good or not. This model could then be used to generate an unlimited about of feedback data and scale up the training procedure even further.

Additionally, we'd like to work on learning from negative feedback. We considered specifying a negative reward, but in our experiments, this approach resulted in divergence during training. Perhaps we can take inspiration from reinforcement learning and treat music generation as a reward maximization problem. If we were somehow able to start *tabula rasa* and learn only from feedback, we may be able to circumvent the memorization-and-playback problem, and instead discover how to generate music from first principles.

Code and audio samples

The code is available at github.com/cwolffff/musicgen. Audio samples from the model architecture evaluation are available at soundcloud.com/sterling-alic/sets/listener-adaptive-feedback1 and audio samples from the feedback training evaluation are available at soundcloud.com/sterling-alic/sets/listener-adaptive-feedback2.

Contributions

Christopher worked on training the models, implementing the feedback training loop, and the model evaluation experiments. He also wrote the Dataset, Methods, and Experiments sections of the report. Sterling worked on preprocessing the midi files into event tokens, as well as post processing of the data. He contributed to the Abstract, Introduction, and Dataset sections, created the visuals for and edited the project report video, and also created the figures for the model architecture and training algorithms.

References

[1] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, Nov. 1997.

[2] Cheng-Zhi Anna Huang, Ashish Vaswani, Jakob Uszkoreit, Noam Shazeer, Ian Simon, Curtis Hawthorne, Andrew M. Dai, Matthew D. Hoffman, Monica Dinculescu, and Douglas Eck. Music transformer, 2018.

[3] Natasha Jaques, Shixiang Gu, Dzmitry Bahdanau, José Miguel Hernández-Lobato, Richard E. Turner, and Douglas Eck. Sequence tutor: Conservative fine-tuning of sequence generation models with kl-control, 2017.

[4] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[5] Christine Payne. Musenet, 2019.

[6] Adam Roberts, Jesse Engel, Colin Raffel, Curtis Hawthorne, and Douglas Eck. A hierarchical latent vector model for learning long-term structure in music, 2019.

[7] Ian Simon and Sageev Oore. Performance rnn: Generating music with expressive timing and dynamics. `https://magenta.tensorflow.org/performance-rnn`, 2017.

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

[9] Ziyu Wang, Ke Chen, Junyan Jiang, Yiyi Zhang, Maoran Xu, Shuqi Dai, Xianbin Gu, and Gus Xia. Pop909: A pop-song dataset for music arrangement generation, 2020.