Performing Structured Improvisations with Pre-trained Deep Learning Models

Pablo Samuel Castro
Google Brain
psc@google.com

Abstract

The quality of outputs produced by deep generative models for music have seen a dramatic improvement in the last few years. However, most deep learning models perform in "offline" mode, with few restrictions on the processing time. Integrating these types of models into a live structured performance poses a challenge because of the necessity to respect the beat and harmony. Further, these deep models tend to be agnostic to the style of a performer, which often renders them impractical for live performance. In this paper we propose a system which enables the integration of out-of-the-box generative models by leveraging the musician's creativity and expertise.

Introduction

The popularity and quality of machine learning models has seen a tremendous growth over the last few years. *Generative models*, which are trained to produce outputs resembling a pre-specified data distribution, have attracted much attention from both the scientific and artistic community in large part due to the realism of the outputs produced.

In the musical domain, recent works produce music that is both realistic and interpolatable (Roberts et al., 2018), closely resembles human performance (Huang et al., 2019), and can aid in automatic composition[1]. The increased realism of these models is typically accompanied with an increase in the amount of processing time required to generate outputs. Unfortunately, long processing times generally renders these models inadequate for live performance. This issue is particularly stark in structured improvisation, such as in traditional jazz, where the music produced must respect the beat and harmony of the piece.

In this paper we introduce a software system that enables the incorporation of generative musical models into musical improvisation. This can be used as both a solo-performance or in an ensemble. Our system produces a performance that is a hybrid of human improvisation with melodies and rhythms generated by deep learning models. Our hybrid approach enables us to address real-time compatibility and stylistic personalization.

[1]https://www.ampermusic.com/

Background

We use Recurrent Neural Networks (RNNs) (Rumelhart, Hinton, and Williams, 1986) as the machine learning models for generating drum beats and melodies. Recurrent Neural Networks are a special type of neural network which process a *sequence* of tokenized inputs one token at a time, updating an internal state after processing each input. A trained RNN can be used for generation: after processing a sequence of tokens $t_{1:n}$, sample from the resulting internal distribution over the token dictionary. When using these models for generation, we will refer to the initial sequence $t_{1:n}$ fed into the model as the *primer sequence*.

We will make use of two LSTM-models from Google Magenta. The first is MelodyRNN (Magenta, 2016b). It processes *note events* as tokens, where a note event contains a note's pitch and its duration. The model assumes monophonic melodies (i.e. only one note played at a time) and is instrument agnostic. Thousands of MIDI files were used for training. These MIDI files were quantized into 16th notes: that is, the minimum allowable time between two notes are one 16th note[2]. When using this model to generate new melodies, the melodies produced tend to match the key signature and note density of the primer melody sequence fed into it, which is a desirable property for our use case. The second is DrumsRNN (Magenta, 2016a). The model is similar to MelodyRNN, but here there is polyphony as multiple drums can be hit simultaneously. As for MelodyRNN, this model was trained on thousands of MIDI files, quantized into 16th notes.

Related Work

There have been a number of works proposing new types of digital instruments which make use of machine learning models. The Wekinator (Fiebrink, 2009) enables users to train new models in a *supervised* fashion by providing pairs of inputs and expected outputs; inputs can be provided in many forms including using computer controllers and physical gestures, while outputs can be sent to any musical, digital or physical actuator. This contrasts with our proposed framework, which does not require retraining a model, but rather adapt the outputs of a pre-trained deep learning model to a performer's style.

[2]There are sixteen 16th notes in one bar of 4/4 time.

Thom (2000) and Thom (2001) build probabilistic models to emulate an improviser's tonal and melodic trends. Johnson, Keller, and Weintraut (2017) makes use of two LSTMs: one for intervals between notes and the other for note intervals relative to the underlying chord progression; these trained models are then combined to generate melodies in a recurrent note-by-note fashion. In (Weinberg et al., 2009) the authors introduce *shimon*, a robot marimba player capable of interacting with human players. The robot has human-like movements (such as head-bobbing, "gazing" to pass on the solo to another player, etc.) which make it natural to interact with. Closely related to our use of 'continuations' are *The Continuator* of Pachet (2003), where the authors use Markov models to adapt to a user's style. In contrast to our work, however, the continuator is agnostic to the underlying beat of a performance, which is essential to jazz improvisation. Bretan et al. (2017) propose training a deep autoencoder to encode melodies played by a performer into a latent space that has been trained to capture musical consistency; the closest melody from a library that has been embedded into the same latent space is returned, allowing their system to respond in near real-time. Roberts et al. (2018) propose a deep autoencoder model for encoding melodies into a latent space, combined with a deep decoder for converting points from that latent space into cohesive melodies. Huang et al. (2019) trained a transformer model (Vaswani et al., 2017) on a dataset of virtuoso piano performances, resulting in a model that can produce highly realistic and novel musical snippets.

System setup

Our setup assumes a piano keyboard connected to a computer via MIDI used for input, along with an additional controller for enabling more MIDI control messages; in our case we are using the Korg Nanokontrol2 MIDI controller but the system can be used with any MIDI controller. We use SuperCollider[3] to detect all incoming MIDI events and pipe them as OSC[4] messages to a Python backend running on the same machine. The Python backend processes the notes and may then send an OSC message containing notes to be played to SuperCollider, which either generates the sound or forwards them to an external MIDI controller for producing the sound.

The SuperCollider component acts mostly as a bridge between the MIDI controllers and the Python backend. It defines a set of handlers for routing MIDI input messages to the backend via OSC messages, and handles OSC messages from the backend. When a note on/off message is received from the backend, it can either redirect to an external MIDI controller or produce the sound itself. For the latter, the SuperCollider code loads a set of WAV files as well as a few synthetic instruments for playback.

Backend design

At its core, the Python backend is running a continuous loop over a customizable number of bars, each with a customizable number of beats. Each is discretized it into 16th note segments (so one bar in 4/4 time signature will have 16 intervals). Multi-threading is used to allow for real-time response, and we maintain a set of global variables that are shared across the different threads, the most important of which are listed below:

- $\mathbf{time\_signature}$: An object containing a pair of integers denoting the $numerator$ (4, 6, 7, etc.) and $denominator$ (4, 8, or 16) of the time signature.
- $\mathbf{qpm}$: A float indicating the speed (quarters-per-minute) of playback. One quarter note is equal to four 16th notes, so this value indicates the time needed to process four 16th note events.
- $\mathbf{playable\_notes}$: A SortedList where we store each playable note event. Each element contains the type of playback event (click track, bass, drums, etc.), the note pitch, the instrument itself (bass, keyboard, hi-hat, bass drum, crash, etc.), and the 16th note in the bar where the event occurs.
- $\mathbf{bass\_line}$: Similar to $playable\_notes$ but containing only the current bassline.
- $\mathbf{accumulated\_primer\_melody}$: A list which will accumulate the note pitches played by the human improviser. Once enough notes have been accumulated they will be sent as a 'primer' melody to MelodyRNN. This is discussed in more detail in the Improvisation section.
- $\mathbf{generated\_melody}$: A list containing the note pitches produced by MelodyRNN. When full, the note pitches played by the human will be replaced by the pitches in this buffer.

The open source-code can be accessed at https://github.com/psc-g/Psc2.

Click-track generation

The first step is setting the number of bars, time signature, and tempo (qpm). The user may change the number of bars, time signature numerator, and time signature denominator via a set of buttons on the Nanokontrol2. The qpm may be adjusted via a knob or by tapping the beat on a button. These define the length and structure of the *sequence*, which the system will loop over. Once these are set the user may start playback by hitting the 'play' button on the Nanokontrol2. This will start a click-track which will make use of 3 different click sounds:

1. The first will play on the first beat of the first bar, to indicate the start of the sequence. This is important for the user to known the start of the sequence when recording a bassline or chords.
2. The second will play on the first beat of the remaining bars in the sequence (if at least two bars were selected)
3. The third will play within each bar at a frequency marked by the time signature denominator: if the denominator is 4, it will play a click every four 16th notes; if it is 8, it will play every two 16th notes; if it is 16 it will play a click every 16th note.

[3]https://supercollider.github.io/

[4]http://opensoundcontrol.org/

Once the click-track has been started, the user can place the system in one of four *modes* via buttons on the Nanokontrol2. When SuperCollider is in charge of producing sounds, ach mode uses a different instrument for playback.

- **bass:** The user can record a bassline which will be looped over. After entering this mode, recording begins as soon as a note is pressed and proceeds until the end of the sequence is reached.
- **chords:** The user can play a set of chords to include in the loop playback. As in bass mode, recording begins as soon as a note is pressed and proceeds until the end of the sequence is reached.
- **improv:** Used for improvising over the loop playback in a call-and-response between the human and the machine learning model. This mechanism is discussed in more detail in the Improvisation section.
- **free:** Free-play mode, where the human can improvise freely over the loop playback.

Drums generation

Our system generates two types of drum beats: a deterministic one and and another which is generated by a machine learning model. The deterministic one is built off of the bassline as follows:

1. A bass drum note is added at the first beat of every bar.
2. A snare note is added at each bass note onset.
3. Hi-hat notes are added at each 8th note (so every two 16th notes).

By pressing one of the Nanokontrol2 buttons, this deterministic drum beat is fed into DrumsRNN as a 'primer' to produce a new beat. Figure 1 illustrates this process in musical notation.

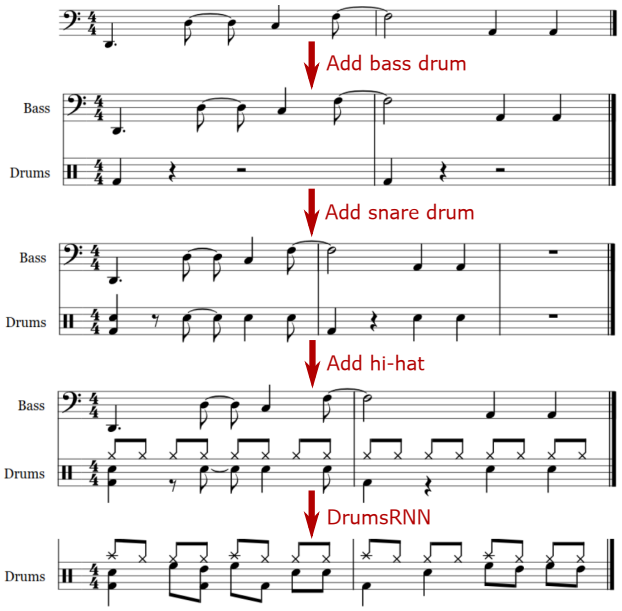

Figure 1: Building the drum beats. From top-to bottom: starting from a specified bassline, bass drum notes are added on the first beat of each bar, snare drum notes are added for each bass-note onset, and finally hi-hat notes are added at each 8th note. This deterministic drum beat can then be sent as a 'primer' to DrumsRNN which will generate a new beat.

Improvisation

The improvisational part of our system is inspired on the call-and-response improvisations that are common in traditional jazz. In these sections two or more musicians take turns improvising over the same piece, and each musician usually incorporates melodies and/or rhythms played by previous musicians into their improvisations.

There are two main components to an improvisation: the pitches chosen and the rhythm of the notes. In our experience playing with generative models, such as MelodyRNN, we found that the rhythm of the melodies produced is not very reflective of the types of rhythms observed from professional improvisers. This may be due in large part to the 16th note quantization that is necessary for training the models. To overcome this issue, we propose a hybrid approach: the machine learning models provide the pitches, while the human provides the rhythm.

The way this is achieved is as follows:

1. Collect the pitches played by the human improviser in the $accumulated\_primer\_melody$ global buffer.
2. Once the number of notes in the buffer is above a prespecified threshold, the buffer is fed into MelodyRNN as a primer melody in a separate thread.
3. When the MelodyRNN thread has finished generating a new melody, it will store *only the pitches* in the $generated\_melody$ buffer (the rhythmic information is dropped).
4. When the main looper thread detects that the $generated\_melody$ buffer has been filled, it will **intercept** incoming notes played by the user and replace their *pitches* with the pitches stored in $generated\_melody$ (and removing said pitch from the buffer). Figure 2 illustrates this process.
5. Once $generated\_melody$ is empty, return to step 1.

Our hybrid approach to machine-learning based improvisation allows us to mitigate the two problems mentioned in the introduction: real-time compatibility and stylistic personalization. The former is handled by performing the inference in a separate thread and only using it when it is available. The latter is handled by maintaining the rhythmic inputs from the human performer. It has been found that rhythm can significantly aid in facilitating melody detection (Jones, 1987), which we believe also carries over to enhancing the personalized style of performance. Further, by leveraging the human's rhythmic input, we are able to avoid having the limitation of the 16th-note quantization that the RNN models require.

We provide some videos demonstrating the use of this system at https://github.com/psc-g/Psc2/tree/master/research/nips2018.

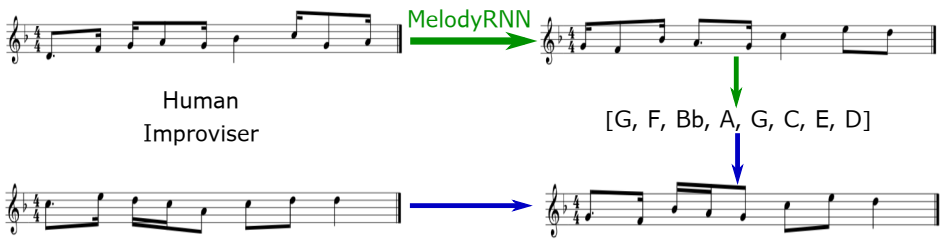

Figure 2: Building the hybrid improvisation. 1. The melody from the human improviser (top-left) is fed into MelodyRNN. 2. MelodyRNN produces a new melody (top-right). 3. The human improviser plays a new melody (bottom-left). 4. A new hybrid melody is created by combining the pitches from the MelodyRNN model with the rhythm from the most recent human improvisation (bottom-right).

Evaluation

Our proposed system has been used for live jazz performance in a piano-drums duet. The songs performed were some that the duo regularly plays, and our system was engaged during the improvisation sections of these songs. Since there was a human drummer performing, only the improvisation (MelodyRNN) part of our system was used. We report the feedback received from these performances.

Strengths

- The system was able to respond in real-time.
- The audience (many of which are familiarized with the pianist's style) reported not noticing that there was an external system affecting the improvisations (they were only made aware of it after the show).
- The pianist felt creatively challenged when improvising with the system engaged, which led to a different way of playing.

Weaknesses

- The system did not work well on songs with many harmonic changes.
- The system would sometimes break up the pianist's lines before they were done.
- The system would sometimes jump octaves when engaging.

Conclusion and Future Work

In this paper we have introduced a system that enables the integration of out-of-the-box deep learning models for live improvisation. We have designed it in a way that it does not require machine learning expertise to use, and can be extended to other types of musical generative models with little effort. Our hybrid approach for generating machine-learning based improvisations maintains the style of the human improviser while producing novel improvised melodies.

Although our system was built with MelodyRNN and DrumsRNN, the setup can be used with any musical generative model with relatively little effort. Along these lines, one avenue we would like to explore in the future is the incorporation of models which do not require quantization, such as PerformanceRNN (Simon and Oore, 2017); one challenge is to ensure that the personal style of the human improviser is maintained.

Expert musicians are able to produce high-quality improvisations *consistently* from having honed their craft over many years of practice. A common frustration with these artists, however, is that they often find their improvisations too predictable, and struggle escaping their "pocket". Our hope is that systems like the one we are proposing here can push expert musicians, and artists in general, out of their comfort zone and in new directions they may not have chosen to go to on their own. The experience of the pianist we reported in the previous section perfectly showcases this.

We hope to improve the system by allowing the performer to have more control over when the system begins recording, and when the system replaces the notes. We have already begun experimenting with this extra modality using a MIDI footpedal. Initial experiments suggest this added level of control mitigates for many of the issues raised by the human performer regarding the timing of when the system is engaged, while maintaining the novelty of the melodies produced.

References

Bretan, M.; Oore, S.; Engel, J.; Eck, D.; and Heck, L. 2017. Deep Music: Towards Musical Dialogue. In *AAAI*, 5081–5082.

Fiebrink, R. 2009. Wekinator. `http://www.wekinator.org/`.

Huang, C. A.; Vaswani, A.; Uszkoreit, J.; Shazeer, N.; Hawthorne, C.; Dai, A. M.; Hoffman, M. D.; and Eck, D. 2019. An Improved Relative Self-Attention Mechanism for Transformer with Application to Music Generation.

Johnson, D. D.; Keller, R. M.; and Weintraut, N. 2017. Learning to Create Jazz Melodies Using a Product of Experts. In *Proceedings of the The Eighth International Conference on Computational Creativity*.

Jones, M. R. 1987. Dynamic pattern structure in music: Recent theory and research. *Perception & Psychophysics* 41(6):621–634.

Magenta, G. 2016a. Drumsrnn. `https://github.com/tensorflow/magenta/tree/master/magenta/models/drums_rnn`.

Magenta, G. 2016b. Melodyrnn. https://github.com/tensorflow/magenta/tree/master/magenta/models/melody_rnn.

Pachet, F. 2003. The continuator: Musical interaction with style. *Journal of New Music Research* 32(3):333–341.

Roberts, A.; Engel, J.; Raffel, C.; Hawthorne, C.; and Eck, D. 2018. A Hierarchical Latent Vector Model for Learning Long-Term Structure in Music. In *Proceedings of the International Conference on Machine Learning*.

Rumelhart, D. E.; Hinton, G. E.; and Williams, R. J. 1986. Learning representations by back-propagating errors. *Nature* 323:533–.

Simon, I., and Oore, S. 2017. Performance rnn: Generating music with expressive timing and dynamics. https://magenta.tensorflow.org/performance-rnn.

Thom, B. 2000. Unsupervised Learning and Interactive Jazz/Blues Improvisation. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*.

Thom, B. 2001. Machine learning techniques for real-time improvisational solo trading. In *Proceedings of the 2001 International Computer Music Conference*.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L. u.; and Polosukhin, I. 2017. Attention is All you Need. 5998–6008.

Weinberg, G.; Malikarjuna, T.; ; and Raman, A. 2009. Interactive jamming with shimon: A social robotic musician. In *Proceedings of the ACM/IEEE International Conference on Human Robot Interaction*, 233–234.