Proceedings of the 7th International Conference on Live Coding (ICLC 2023)
April 19th - 23rd – Utrecht, Netherlands

Genny: Designing and Exploring a Live Coding Interface for Generative Models

Junichi Shimizu
Creative Computing Institute, University of the Arts London
j.shimizu@arts.ac.uk

Rebecca Fiebrink
Creative Computing Institute, University of the Arts London
r.fiebrink@arts.ac.uk

ABSTRACT

We present Genny, a live coding interface for interacting with generative models. By implementing a streamlined generative model API within a Web-based live coding environment, we explore possibilities for how generative models can support live coding. We describe the implementation of our interface components and the backend system for working with generative models, show several design patterns with template code, and describe use of a practical generation method for live coding performance. The system currently enables the generation of two-bar rhythm patterns using lightweight models that support near-instantaneous inference, which are able to suggest and display alternative rhythm patterns based on live coders' own inputs. Finally, we discuss reflections on this implementation approach, its current limitations, and further possibilities.

1 Introduction

In this paper, we present Genny, a live coding interface for working with generative music models and a set of new, lightweight code patterns for generating musical rhythm sequences from these models. We explore several ways to use generative models such as variational autoencoders (VAEs) and recurrent neural networks (RNNs) to instantly suggest and play new rhythms. Although past work has explored some methods for integrating generative algorithms into live coding, there are open questions around how to expose live-coding interactions with models in ways that are transparent, understandable, and appropriate to real-time use. To explore these questions, we have implemented from scratch a Web-based code editor and simple live coding functions to support fast and transparent interaction with pre-trained models compatible with the existing Magenta.js JavaScript machine learning library (Roberts et al. 2019). These functions allow live coders to sample data from a model's latent space and convert it to a note sequence—visible and modifiable to the coder—as well as to manipulate and create variations on live coders' own rhythm patterns. As our goal in this work is to demonstrate a proof-of-concept implementation for fast and transparent model use, and to explore its interactive possibilities and consequences, we focus on the use of pre-trained MusicVAE and MusicRNN models, and we restrict the model in this first implementation to work with two-bar rhythm sequences as both model input and output.

In this paper, we also show how our implemented functionality supports particular design patterns for using generative models in live coding. We demonstrate how generative models may be seamlessly used alongside more conventional live coding techniques, opening up new musical and interactive possibilities. Finally, we discuss how this approach may inform the future use of generative models in musical live coding, and reflect on challenges around machine learning and live coding raised by previous work (e.g., Knotts and Paz 2021).

2 Related Work

2.1 Generative Machine Learning for Music Creation

Generative algorithms have long been used in computational music systems. Early machine learning (ML) approaches for generating musical sequences can be found, for instance, in Zicarelli's (1987) work with Markov Chains and Dubnov et al.'s (1998) work with incremental parsing.

More recently, deep learning approaches have led to an explosion of techniques and systems for generating music using symbolic (e.g., MIDI) and audio representations. For instance, Google's Magenta Studio (Roberts et al. 2019) provides tools for symbolic melody and drum generation, using RNNs and VAEs. OpenAI's MuseNet (Payne 2019) and Jukebox (Dhariwal et al. 2020) employ transformer models to generate musical MIDI and audio, respectively. Extensive recent research explores approaches to applying generative ML to tasks such as drum accompaniment generation (Haki et al. 2022), audio synthesis (Caillon and Esling 2021), and human-"steerable" composition tasks (Young et al. 2022).

Such contemporary generative ML approaches afford a variety of modes of interaction. One possibility is to generate entire novel musical sequences from scratch: for instance, RNNs and their variants such as LSTMs can be used to generate a sequence of symbols (e.g., representing pitches and/or durations). Another possibility is to generate continuations of partial sequences provided by a user, for instance to continue a melody played on a keyboard. And some models can be used to fill in new musical parts, for instance adding a harmony or a drum sequence to a melody.

Furthermore, models such as VAEs are capable of mapping between a lower-dimensional representation—a vector representing a point in "latent space"—and a higher-dimensional one—generating a fully realized melody, rhythm, etc. This approach enables users to manipulate lower-dimensional representations to achieve musically interesting results. For instance, variations of a melody may be achieved by mapping the melody to its corresponding latent vector, slightly perturbing this vector, and mapping these perturbations back to new melodies. Or—as implemented in tools such as Magenta Studio's Interpolate (Roberts et al. 2019)—musical interpolations between two source melodies may be generated by interpolating between their latent vectors, generating new melodies for each in a sequence of latent vectors that incrementally hop from the first source melody to the second. Or, a user interface may provide control over navigating the latent space itself, as in R-VAE (Vigliensoni et al. 2022), providing a convenient and understandable way to manipulate the generation in realtime.

2.2 Tools and Interfaces for Generative Machine Learning in Music

Most research work developing generative ML techniques for music is made available to others—if at all—as source code, typically in Python or occasionally JavaScript. However, some examples exist of generative ML music systems being released with graphical user interfaces (GUIs) that enable people to experiment with the generative models in real-time, without programming: these include Magenta Studio (Roberts et al. 2019), which is available as a desktop application and as an Ableton Live plugin, as well as other Web demos released by the Magenta research team[1]. However, such applications generally lack the flexibility of model use afforded by programming.

2.2.1 Generative Machine Learning Tools for Live Coding

A number of projects have thus far provided implementations of generative ML music models in live coding environments. MIMIC (McCallum and Greirson 2020) is a Web-based programming environment providing a high-level API wrapping machine learning functionality from tensorflow.js, Magenta.js, and RAPID-MIX; while designed to support "musicians using machine learning to build new musical instruments in the browser" (McCallum and Grierson 2020, p.271), it can also used as a live coding environment (Grierson et al. 2021). Nevertheless, creating a new music generation project in MIMIC requires substantial boilerplate code, and the ML syntax lacks the efficiency and transparency optimal for live-coding.

Sema (Bernardo et al. 2020), built on the MIMIC platform, is a "playground" for programmers to build their own browser-based live-coding languages, which can themselves employ ML libraries Tensorflow.js and Magenta.js for generation. Sema provides a model architecture for high-efficiency and low-latency live-coding language design, but does not itself strongly impose a syntax or design patterns for generative ML use—leaving it up to language designers to navigate these challenges.

Reppel's Mégra (2020) uses probabilistic finite state automata to create musical sequences based on very small datasets, including those supplied on-the-fly within a live coding performance. Similar to Reppel's 2020 ICLC paper, we aim here to provide "a case study on how a machine learning method can be put into action within live coding practice," though in our case focusing on different ML methods commonly used to generate melody and rhythm sequences (RNNs and VAEs).

[1] https://magenta.tensorflow.org/demos/web/

Other work has used generative models (or similar techniques) in different ways, for instance generating executable code rather than musical sequences themselves, as in IxiLang (Magnussen 2011) and Cibo (Stewart et al. 2020), or generating complementary patterns to those played by a human performer (Navarro and Ogborn 2017).

Knotts and Paz (2021) provide a discussion of "tensions and frictions" that arise when using ML in live coding, particularly against the aims of the TOPLAP draft manifesto[2], which has "often provided a guiding philosophy for the development of [live-coding] practices." In the Discussion section below, we explore certain of these tensions and frictions in more depth, grounded in the experience of designing and using the interface we have designed. In addition to this dialogue with Knotts and Paz's work, we contribute to the existing literature on generative ML tools for live coding through describing new approaches to implementing ML in a live coding environment, and discussing some of the practical challenges and considerations that arise when using ML in live coding.

3 Implementation

In this section, we describe the implementation and usage of Genny. Because our primary aim is to explore interaction mechanisms and musical affordances for using generative models in live coding, we have not taken the more substantial engineering work of integrating new generative functionality into an existing live-coding language; rather, we have designed a new environment from scratch. Further, our focus thus far has been on enabling generation and manipulation of rhythms.

3.1 Coding Interface

The coding interface is shown in Figure 1. The accompanying video also demonstrates the interface functionality.[3]

Figure 1: *The appearance of the interface.*

The user interface consists of the code editor where live coders mostly run their code, with a console view for notifying the UI status when updating code and getting the results from generative models. The interface also includes a *Pattern Sample* button to switch the different template code to support programmers in learning how Genny works. The following section describes the key components of this interface in more detail.

[2] https://toplap.org/wiki/ManifestoDraft

[3] Demo video: https://vimeo.com/781969647/824b802a67

3.1.1 Code Editor

The code editor interface enables the live coder to explore effects of model parameters and produce models' generation results, which can be represented explicitly in the code. This code editor parses the text written on the editor and calls each defined function. This editor implementation uses the Code Mirror[4] React library to support a simple JavaScript-based syntax. The editor supports autocompletion, with suggested functions displayed in-line in the editor, enabling coders to quickly browse and choose from functions for setting each model parameter (as defined by the model configuration—see 3.2.3). The code editor also accepts keyboard commands: *Cmd+Enter* to execute code and *Cmd+Shift+Enter* to stop the audio.

3.1.2 Audio Control

For real-time audio control, we used Tone.js[5], a wrapper for the Web Audio API for creating interactive music in the browser. We use the same syntax for manually defining a rhythm sequence in code and representing rhythm patterns generated by the models. Specifically, we define a rhythm sequence using the "Transport Time Code" of this system as

`<measure> : <beat> : <subdivision>`

For instance, if the transport code is written as "1:2:3", this means the measure is two, the beat is three and subdivision is four. Note that each value starts from zero. In other words, the actual sequence timing must be considered as an incrementing number. The live coder can update the playing stream whenever executing a new command without stopping the audio or interrupting the metric flow.

3.1.3 Visual Exploration

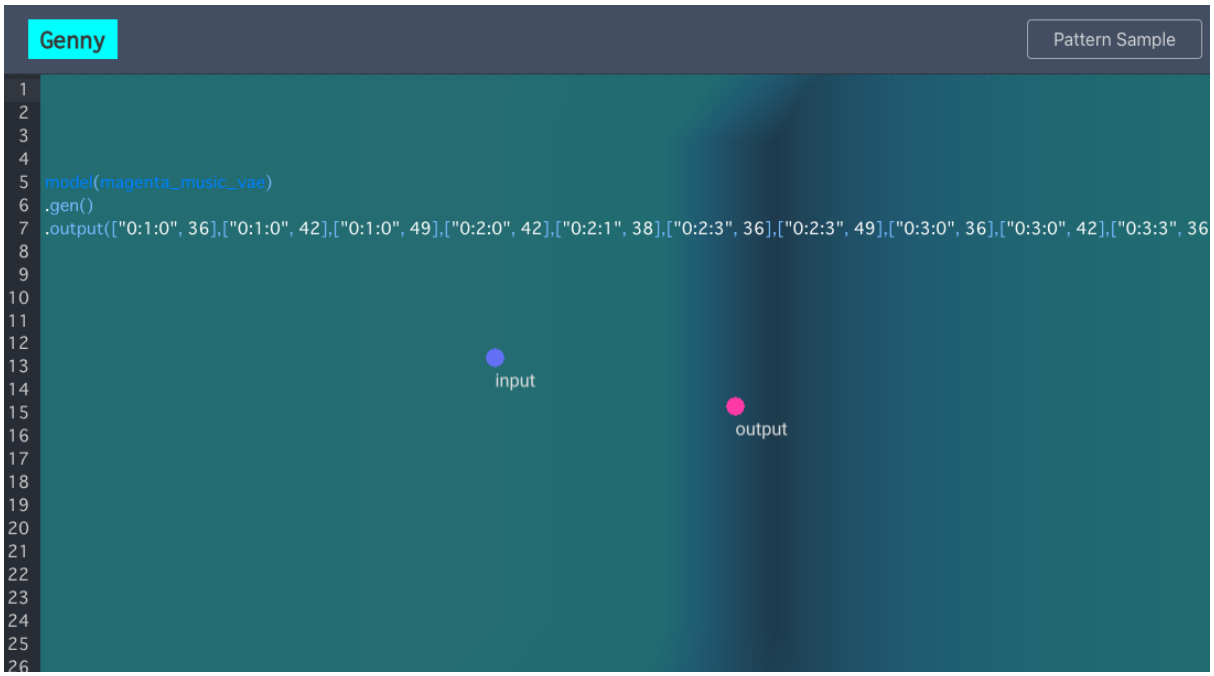

Figure 2: *Visualization of two different vectors to show similarity.*

We have implemented a simple visualization to aid live-coders and audiences in understanding how the generative models work and how they are being used. This visualization communicates the degree of similarity between latent vectors representing a model's current input and output. More specifically, we compute cosine similarity between these two vectors—if this similarity value is close to zero, this indicates high similarity, while increasing values indicate lower similarity. The visualization, implemented using GLSL[6], continually updates to show the degree of similarity of current and previous generated data as the distance between two points. (Note that only distance between these points is

[4] https://github.com/uiwjs/react-codemirror
[5] https://tonejs.github.io/
[6] https://github.com/onnovisser/babel-plugin-glsl

meaningful; absolute position on screen is not.) While more sophisticated dimensionality reduction techniques such as multidimensional scaling, t-SNE, or PCA are often used to visualize points in a higher dimensional space (e.g., a latent space) in two dimensions while maintaining approximate distance relationships, this approach is computationally much simpler and communicates the same relevant information about the two points (i.e., their distance).

3.2 Model API

3.2.1 Supported Model

The environment supports use of pre-trained MusicVAE and MusicRNN models through the Magenta.js library (Roberts et al. 2019). MusicVAE is a variational autoencoder that can generate musical sequences. MusicRNN is an LSTM-based language model for continuing sequences from a given input. Both MusicVAE and MusicRNN have open-source implementations[7], enabling the training of new models on arbitrary music datasets. Further, their creators have released multiple pre-trained models as checkpoints on the Magenta.js repository.[8] Users of Genny can additionally choose from amongst these checkpoints.

3.2.2 Limitations

Currently, any input sequence used with the generative models must be limited to two measures, and the output will be also exactly two measures. While this limitation may be problematic for some use cases, it can also be changed depending on which pre-trained checkpoint the user will load. MusicVAE and MusicRNN can potentially support not only drum pattern generation but also such as melody, chord, or even 4-bar generation. However, to support easy real-time iteration in our prototype, we currently only use checkpoints of drum models. Also, the implementation as a JavaScript Web interface imposes some limitations on model size and computational resources and running without hosting server for the model inference.

3.2.3 Configuration

In our implementation, a YAML file specifies the configuration for each model type. The configuration consists of the name of the model type (e.g., "magenta_music_vae"), its set of available parameters, these parameters' default values, and their available functions as well as informative descriptions of those functions. Figure 3 shows an example configuration. This approach is similar to those used by configuration frameworks such as Hydra[9], and it has allowed us as live-coding environment designers to easily experiment with different versions of the model-specific APIs as well as linking the model APIs to the interface (e.g., for use in the autocomplete function).

Each parameter can be updated from the code. For instance, in Figure 3 the default temperature setting is 1.0. If a coder wants to change this, they can call the corresponding function on the model with a new parameter value, for example `temperature(0.3)`.

3.3 Pattern Representations for Rhythms

The pattern() object allows the input of rhythm sequences data in Tone.js format. The pitch representation is mapped to drum types based on General MIDI mapping conventions. To enable a live coder to explore many patterns within the same code, the `pattern()` object allows numbering such as `pattern1, pattern2, ... patternN`.

```
pattern(
    ["0:0:0", 36],
    ["0:3:0", 36],
    ["1:3:0", 38],
)
.play()
```

Calling the `play()` function on a pattern, as in the code above, will play it. Multiple patterns may be played simultaneously.

[7] See music_vae, drums_rnn, melody_rnn models at https://github.com/magenta/magenta/tree/main/magenta/models

[8] https://github.com/magenta/magenta-js/blob/master/music/checkpoints/README.md

[9] https://hydra.cc/

```yaml
model:
    name: "magenta_music_vae" # Define the model name to load from code editor
    parameter: #parameter properties to access the model properties.
        - checkpoint: "checkpoint url"
        - temperature: 0.0
        - similarity: 1.0
        - zDim : 256
    function: #function properties to access model function
        - input: "input sequence encoding to tensor and decode the sequence as output"
        - similar: "similar function, sampling z and interpolate from input sequence"
        - interpolation: "interpolation function"
```

Figure 3: *An example YAML model configuration file specifying model parameters and default values as well as functions and their descriptions.*

4 Design Patterns for Live Coding with Generative Models

In this section, we describe how the basic syntax above can be combined with function calls on the generative models to support several types of building blocks for musical interaction in a live-coding setting. We strongly suggest the reader refer to the accompanying video for a demonstration of these techniques.

4.1 Generating New Patterns

The `model()` object is used for loading a specific VAE or RNN model. This model name should be same as that specified as the name in the configuration file (Section 3.2.3). The supported autocompletion is automatically changed to reflect the functions available for the model name being loaded.

```
model(magenta_music_vae)
.temperature(0.3)
.gen()
```

Each model parameter can be updated through calling the associated function as defined in the configuration file, with default values used for parameters not explicitly set in the code. Calling the gen() function causes a model to generate a new pattern using the current parameters. In the code above, a new sequence is sampled from the MusicVAE, using a temperature setting of 0.3 (lower temperatures generally lead to more conventional sequences, and higher ones can produce more unusual or unexpected sequences).

4.2 Capturing Generated Model Output as a Code Representation

The `output()` function can optionally be used to produce an in-line code representation of the sequence generated by a model. For example, when the code in Figure 4 at left is run, it transforms on screen to the code in Figure 4 at right, which contains the generated sequence. This sequence can now be manipulated in the code, for instance stored as a new pattern, manually edited, or used to generate similar or interpolated patterns as described in the following section.

4.3 Generating Similar Patterns to an Input Pattern

In a MusicVAE there are several types of generation methods available. As shown in the example code below, the `input()` method will encode the rhythm pattern sequence into a latent vector representation and decode this latent vector into note sequences. The `output()` function can optionally then be used to capture the generated output in code,

```
//Before running
model(magenta_music_vae)
.gen()
.output()
```

```
//After running
model(magenta_music_vae)
.gen()
.output(["0:0:0",36],["0:1:3",44],["0:3:0",36],
        ["1:2:0", 44],["1:3:0",38])
```

Figure 4: *The* `output()` *function generates a code representation of model output.*

which enables the user to for instance store the new sequence in a new pattern, which can be manually edited or itself be used as a model input.

MusicVAE models provide a number of other functions, including `similar()`. Like `input()`, the `similar()` function inputs pattern data and generates a new pattern by sampling latent vectors and interpolating encoded input. However, the `similar()` function exposes a similarity parameter where 1 is the most similar and 0 is the least similar, enabling users to tune how similar the output is. This can be also accessed from the model property from the code editor.

```
pattern1(
    ["0:0:0", 36],
    ["0:3:0", 36],
    ["1:3:0", 38],
)

model(magenta_music_vae)
.input(pattern1) // or .similar(pattern1)
.gen()
.output (
    ["0:0:0", 36],
    ["0:1:3", 44],
    ["0:3:0", 36],
    ["1:2:0", 44],
    ["1:3:0", 38],
)
```

4.4 Interpolating Between Patterns

The `interpolate()` function is given two input sequences and interpolates between them in latent space.

```
model(magenta_music_vae)
.interpolate(pattern1, pattern2)
.gen()
```

4.5 Musical Affordances: Global and Local Exploration and Improvisation

As generative models support multiple functions and parameters, live coders can use different functions depending on the situation. The simplest entry point to start with live coding is using `gen()`, which encapsulates all necessary functionality in a simple function call without the need for the live-coder to specify any model parameters. In the early part of a performance, it can be relatively hard to make a good groove within a short time, without pre-prepared code material. A generative model can help here by generating a random sequence instantly. Furthermore it is easy to view the generated pattern using `output()`, potentially reusing this (perhaps in an edited form) in a new `pattern()`.

The default generation method resamples data from random latent vectors, so every time `gen()` is executed, the live coder will get new outputs, which may or may not be musically desired. However, in our experience, the potential to quickly obtain very wide-ranging outputs can be musically helpful in certain stages of performance. In contrast, the `similar()` and `input()` functions can be used for more local exploration around patterns of current interest to the live-coder. Thus, the act of performing with generative models can be one of alternating between higher and lower degrees of surprise, exploring wider or narrower musical spaces as desired.

5 Discussion

In this section, we discuss our reflections on four considerations raised by Knotts and Paz (2022) concerning the use of ML in live coding, informed by our experimentation with the system described in this paper. These considerations are transparency, authorship, creative exploration, and visualization.

5.1 Transparency

Genny currently support the use of multiple pre-trained models, loaded from several checkpoints. The live coders start by loading the model from a specific checkpoint. The acts of model generation are made transparent through the need to explicitly call generation functions on the model, and these can be made even more transparent through the use of the `output()` function. These behavoiurs are key to giving coders and audience confidence that the generative models are working and to better understand what kind of rhythm sequences are being used. Further, each model property can be shown in the code editor, even those such as the chosen checkpoint which might not be changed during a live performance.

Data reliability and bias are other unaddressed challenges relevant to transparency. In this system, when a coder uses available pre-trained models, even when descriptions of the training dataset are supplied, neither the live coder nor the audience have a way to easily understand where the training comes from or what types of biases are present. One partial approach to addressing this in the future is to make visible not only a text description of the chosen checkpoint but also supplemental text such as dataset description, size, and license.

5.2 Authorship

Our implementation still has many restrictions, including only enabling the use of rhythm pattern. One key question for us in the design process was how to represent model generation results to the live coder, and how to bring those generated patterns into their own code. In devising an approach reliant on the `pattern()`, `input()`, and `output()` functions, we provide several avenues for live coders to exercise significant authorship over both human-created and model-generated rhythms. First, live coders can create variations on their own rhythms using `input()` and `similar()`. Second, live coders can choose how to manipulate and use model-generated rhythms by getting results through the `output()` and transforming this to a new `pattern()` (e.g., as shown extensively in the accompanying video). Often, in this approach, the sense is that one is delegating certain responsibilities to a model but still exercising authorship over the musical processes on many levels. Third, of course, live coders can layer their own patterns over model-generated ones. For example, one pattern may be a live coder's working bass drum sequence, and a generative model can generate a snare pattern that is similar to a live coder's snare input template.

At present, the system does not support live authorship of the models themselves, through training or fine-tuning on the coder's own data. The interface can be used with models trained separately by live coders, if those models conform to the MusicVAE or MusicRNN formats. Still, there remain a number of practical challenges around effectively training such models on new datasets, from the technical expertise and hardware required to the potential inability to train effectively on small data. Future work might explore how small-data modeling approaches like R-VAE (Vigliensoni et al. 2022) could be integrated into this interface, offering not only an increased sense of authorship but also much-increased control over the musical space provided by the models.

5.3 Creative Exploration

We have already described above how the current system supports both global exploration in a large space of possibilities and local exploration of musical variations, enabling a live coder to chart musical trajectories between these two over time.

To encourage more diverse and creative output, it is important to consider the level of abstraction being used. There is a tradeoff between a high abstraction level and complexity. Currently, we have made available a small number of parameters for the generative models which we have identified as most relevant to creative exploration. Input and function arguments are also highly simplified. One challenge we encountered is that exposing more parameters and functions requires more learning by live coders (and presents more room for error, for instance parameterizations in which a model may not produce useful results). We do not pretend to have made optimal decisions about these tradeoffs in the current implementation; future work should explore code and non-code (e.g., documentation) mechanisms to expose more model flexibility while maintaining usability and a low bar to entry for new users.

There are also still limitations in our representation of musical sequences that should be improved to support more exploration. First, we obviously support only rhythm sequence patterns. Extending this to melodic patterns is rather straightforward, but extending to other data formats such as audio domain sequences presents new challenges.

5.4 Visualization

In our experience, even the simple visualization technique implemented here can be helpful to understand model activity and to reflect global and local pattern exploration activities.

In contrast to simply showing the latent vector itself, 2D visualizations can be more suitable for communicating contextual information. As mentioned above, there exist more sophisticated techniques (e.g., t-SNE, PCA) for representing high-dimensional latent spaces in a 2D visualization. But these techniques themselves require a set of training data and potentially a lengthy training process, making them unsuitable for live-coding, and each still presents challenges for people trying to interpret the resulting visualization.

In our experience, the simple visualization can be useful. However, there can still be a significant gap between machine and human interpretation. For instance, cosine similarity (and consequently visual distance) does not always correlate with perceptual similarity. Thus, the more way of visualization techniques could be explored in the future.

6 Conclusions

In this paper, we introduced Genny, a Web-based live coding interface designed for working with generative models such as VAEs and RNNs. We described the implementation of the coding interface and the generative model API. We describe in the paper and demonstrate in a supplementary video how these afford several unique design patterns for live-coding interactions with rhythm generation. We reflect on how this implementation of generative models can be evaluated with regard to concept relevant to ML and live-coding highlighted in recent work—specifically, transparency, authorship, creative exploration, and visualization—and how our work informs further thinking about these concepts.

7 References

Bernardo, F., C. Kiefer, and T. Magnusson. 2020. "Designing for a pluralist and user-friendly live code language ecosystem with Sema." In *Proc. International Conference on Live Coding*, pp. 41-57.

Caillon, A., and P. Esling. 2021. "RAVE: A variational autoencoder for fast and high-quality neural audio synthesis." *arXiv preprint arXiv:2111.05011.*

Dhariwal, P., H. Jun, C. Payne, J.W. Kim, A. Radford, and I. Sutskever. 2020. "Jukebox: A generative model for music." *arXiv preprint arXiv:2005.00341.*

Dubnov, S., G. Assayag, and R. El-Yaniv. 1998. "Universal classification applied to musical sequences." *Proceedings of the International Computer Music Conference.*

Grierson, M., M. Yee-King, and L. McCallum. 2021. "Executive Order." *Proceedings of the International Conference on New Interfaces for Musical Expression.*

Knotts, S. and I. Paz. 2021. "Live coding and machine learning is dangerous: Show us your algorithms." *Proceedings of the International Conference on Live Coding.*

Magnusson, T. 2011. "The ixi lang: A supercollider parasite for live coding." *Proceedings of the International Computer Music Conference.*

McCallum, L., and M. S. Grierson. 2020. "Supporting Interactive Machine Learning Approaches to Building Musical Instruments in the Browser." *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 271-272.

Navarro, L., and D. Ogborn. 2017. "Cacharpo: Co-performing Cumbia Sonidera with Deep Abstractions." *Proceedings of the International Conference on Live Coding.*

Payne, C. 2019. "Musenet." OpenAI blog, https: //openai.com/blog/musenet

Reppel, N. 2020. "The Mégra System - Small Data Music Composition and Live Coding Performance." *Proceedings of the International Conference on Live Coding.*

Roberts, A., J. Engel, Y. Mann, J. Gillick, C. Kayacik, S. Nørly, M. Dinculescu, C. Radebaugh, C. Hawthorne, and D. Eck. 2019. "Magenta studio: Augmenting creativity with deep learning in Ableton Live." *Proceedings of the International Workshop on Musical Metacreation (MUME).*

Roberts, A, C. Hawthorne, and I. Simon. 2018. "Magenta.js: A JavaScript API for augmenting creativity with deep learning." *Proceedings of the Joint Workshop on Machine Learning for Music (ICML).*

Stewart, J., S. Lawson, M. Hodnick, and B. Gold. 2020. "Cibo v2: Realtime Livecoding AI Agent. Proc. ICLC.

Vigliensoni, G., L. McCallum, E. Maestre, and R. Fiebrink. 2022. "R-VAE: Live latent space drum rhythm generation from minimal-size datasets." *Journal of Creative Music Systems*, 1 (1).

Young, H., V. Dumoulin, P. S. Castro, J. H. Engel, C.-Z. A. Huang. 2022. "Compositional Steering of Music Transformers." *Proc. 3rd IUI Workshop on Human-AI Co-Creation with Generative Models.*

Zicarelli, David. 1987. "M and jam factory." *Computer Music Journal* 11, no. 4 (1987): 13-29.