RaveForce: A Deep Reinforcement Learning Environment for Music Generation

Qichao Lan
RITMO
Department of Musicology
University of Oslo
qichao.lan@imv.uio.no

Jim Tørresen
RITMO
Department of Informatics
University of Oslo
jimtoer@ifi.uio.no

Alexander Refsum Jensenius
RITMO
Department of Musicology
University of Oslo
a.r.jensenius@imv.uio.no

ABSTRACT

RaveForce is a programming framework designed for a computational music generation method that involves audio sample level evaluation in symbolic music representation generation. It comprises a Python module and a SuperCollider quark. When connected with deep learning frameworks in Python, RaveForce can send the symbolic music representation generated by the neural network as Open Sound Control messages to the SuperCollider for non-real-time synthesis. SuperCollider can convert the symbolic representation into an audio file which will be sent back to the Python as the input of the neural network. With this iterative training, the neural network can be improved with deep reinforcement learning algorithms, taking the quantitative evaluation of the audio file as the reward. In this paper, we find that the proposed method can be used to search new synthesis parameters for a specific timbre of an electronic music note or loop.

1. INTRODUCTION

In a computational music generation task, what is essentially generated? This question leads to a debate on either to generate music in symbolic music representation, e.g. MIDI (Music Instrument Digital Interface) or to generate the audio waveform directly. Symbolic music representations can generally reflect the idiosyncrasy of a music piece, but they can hardly trace detailed music information, such as micro-tonal tunings, timbre nuances and micro-timing. Signal-based music representations are better at preserving micro-level details that are not captured well by the symbolic representations. Thus signal-based workflows—including raw audio generation—may be a solution for computational music generation. However, since raw audio generation requires much more computational resources than symbolic representation methods, there are still some difficulties for this method to generate long multi-track music pieces [1]. Furthermore, without a symbolic representation, these methods can be too sophisticated to explain from a music-theoretical perspective. Hence, our motivation is to find a balance between these two forms of music representation in computational music generation.

Our research question is: how can an A.I system be trained to consider the music sound while generating symbolic music representation? Technically speaking, we hope that the neural network in an A.I system can not only generate symbolic sequences but also convert the symbolic representation into an audio waveform that can be evaluated. To do so, we need to use non-real-time synthesis for the transformation from symbolic music representation to an audio file which will become the input of the neural network, and the output will be accordingly the next symbolic representation. Compared with pure symbolic generation, this method also outputs the corresponding audio waveform, which may broaden the application fields. Besides, different from raw audio generation, we fix the transforming function for the neural network, which may make the computational resource focus more on the target music information than on the function estimation.

In this paper, we will explain the proposed method and provide a programming implementation as well as two simplified music tasks as examples. We start with the background of deep learning music generation in Section 2, demonstrating the relationship between the data type and the neural network architecture. In Section 3, we present our method to improve the symbolic representation and the reason why we choose to use deep reinforcement learning. Section 4 introduces the implementation details of our deep reinforcement learning environment with an emphasis on how we optimise it for a musical context. Section 5 describes the reward function design in customised tasks and explains the evaluation from running time and music quality perspective. In Section 6, we summarise the innovations and limitations of our method as well as our future directions.

2. BACKGROUND

Computational music generation has for a long time been an intriguing topic for musicologists and computer scientists [2]. Of current algorithmic methods, deep learning seems to be particularly relevant for music generation tasks [3]. Deep learning is a method that learns from data representations, so in terms of music generation, it is essential to study the background of how the music representation influences the learning process and result.

2.1 Symbolic vs signal-based representations

Music can typically be represented as either signals (audio) or symbols (score representations). Popular symbolic representation methods include MIDI, musicXML, MEI, and others [4]. Among them, MIDI is one of the most popular data formats being used in deep learning music generation tasks. In some particular styles of music, and particularly the ones based on traditional music notation, MIDI data can be an efficient representation. One example is the piano score generation in the DeepBach project [5]. Another example is that of machine-assisted composition applications, in which MIDI allows for editable features [6]. However, as mentioned in the introduction, there are also many cases in which symbolic representations are inadequate in capturing the richness and nuances of the music in question.

One way to address limitations of symbolic representations is the use of sample-level music generation, as demonstrated in WaveNet [7] and WaveRNN [8]. However, although some progress has been made, the raw audio generation requires a lot of computational resources, and it is too complicated to explain how these samples get organised from a musicology perspective.

The data format can also influence the design of the neural network. In symbolic representations, supervised learning can be found in many applications [9]. For raw audio signals, unsupervised learning techniques such as autoencoder and generative adversarial network (GAN) are frequently adopted [10, 11].

2.2 Reinforcement learning

Reinforcement learning is different from supervised or unsupervised learning techniques in that its updating strategy relies on the interaction between an agent and the environment rather than the function gradient. In a given period—that is, an *episode* in reinforcement learning—the agent will try to maximise the reward it can get. The reward is calculated in each episode, and it is used to update the parameters of the agents [12].

The connection between reinforcement learning and music generation goes back to the use of Markov models in algorithmic composition. As one of the pioneers in automated music generation, in the piece called *Analogique A*, Iannis Xenakis uses Markov models for the order of musical sections [13]. The use of Markov models in composition reveals its connection with reinforcement learning as the action of the agent only depends on the current state. However, in previous research on reinforcement learning in computational music generation [14], the reward function calculation is not based on the sample-level evaluation.

Recently, deep learning technology has brought new possibilities to reinforcement learning as it allows the agents to examine higher-level information. In deep reinforcement learning, the agent can be represented by a neural network, which makes it capable of evaluating the raw audio signal and then output the decision. Deep reinforcement learning has been a success during the past few years since it shows that a virtual agent can surpass human beings in several tasks, e.g. Atari games [15]. After that, there appear more and more algorithms such as Proximal Policy Optimization (PPO) [16]. For testing these algorithms, there are many simulation environments, e.g. the OpenAI Gym [1]. For music, deep reinforcement learning has been used for the score following [17]. However, there is still no environment designed for music generation.

3. DESIGN CONSIDERATION

Though symbolic representations have shown some limitations, generating music at the audio sample level can be computationally expensive. Therefore, we propose to generate the symbolic representation first, and then use these representations to synthesise audio for evaluation.

3.1 From symbolic notation to audio

Our first step is to choose a proper method to convert a symbolic representation to an audio file. Three options are considered:

1. to send the generated sequence to an instrument and record the sound for evaluation.
2. to use other general-purpose programming languages such as C++ for the sound synthesis.
3. to use music programming languages like Max/MSP, Pure Data, Csound and SuperCollider for non-real-time synthesis.

We exclude the first option because it would be too time-consuming, considering there would be a considerable number of iterations in the deep learning training process. The second option is the most efficient in synthesis speed, but it lacks the extensibility from a music perspective as users have to be familiar with the C-style programming languages. The third option best balances the efficiency and usability as music programming languages have already been ubiquitous in the electronic music field [18].

However, both the second and the third option are faced with the same challenge—the *gradient*. In supervised learning, we need to know all the functions and their gradient. After comparing the output of the neural network and the training data, we should fine-tune the parameter of the neural network to minimise the loss with the help of these gradients [19]. In our proposed method, since we involve the non-real-time synthesis, back-propagation cannot be done in this context as the functions used for transforming symbolic representation to audio files are unknown.

3.2 Addressing the gradient problem with deep reinforcement learning

Deep reinforcement learning can solve the gradient problem mentioned above as it relies only on the interaction reward rather than the gradient. Though we cannot get the gradient from the symbolic-to-audio transforming function, We can quantitatively evaluate the synthesised audio to get a reward. Concretely, we train a neural network to output

[1] `https://gym.openai.com`

a sequence of symbolic music notations (such as the parameters for a synthesiser) and send the information to an audio programming language for non-real-time synthesis. Then, we compare the synthesised audio file with the target file, or we can use a neural network to grade the audio file directly. When an action brings a positive reward, the probability of the action should increase, and vice versa.

There are several important concepts in deep reinforcement learning that need to be defined in the music context (see Fig. 1):

1 *Step* refers to the process of executing what has been decided to do in the next 16 th note or rest.

2 *Episode* refers to a series of continuous interactions before the *done* attribute turns to *true*, e.g. the end of a game. In a musical context, we use a *total-step* attribute to decide the length of an episode. Thus, it can vary from one single note to a note sequence.

3 *Observation-space* refers to the current state. In our musical context, we set the currently synthesised audio file to be the observation-space. In other words, the agent should be "aware" of the previous state (synthesised audio) and take the next step accordingly.

4 *Action-space* refers to the set of action choices for the agent. In a musical context, the action-space can be discrete (e.g. a note pitch) or continuous (e.g. the amplitude).

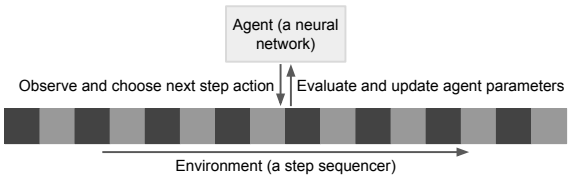

Figure 1. RaveForce architecure: in each note (step), the agent (neural network) will choose an action according to its observation on the current state (currently synthesised audio).

4. IMPLEMENTATION AND OPTIMISATION

As is discussed above, the key to our proposed method is to have an environment that can handle the non-real-time synthesis and evaluate the result. In our implementation of RaveForce [2], we follow the OpenAI Gym interfaces in our Python module, and in SuperCollider, we create a quark to execute the non-real-time audio synthesis. In order to connect with deep learning frameworks, some optimisation is necessary for the observation space.

4.1 The idea from a live coding session

To implement the environment, we refer to a live coding session [20]. In many live coding sessions, SuperCollider [3] has been used as the audio engine as it tracks the time and beat accurately [21]. SuperCollider employs a client-server architecture that contains two parts: the *scsynth* (SuperCollider Synthesiser) and the *sclang* (SuperCollider Language). The *sclang* will be combined in real-time to a simplified version of Open Sound Control (OSC) messages [22] and sent to the *scsynth* to control the sound. This architecture allows the *scsynth* server to run alone, while *sclang* can be replaced by other domain-specific languages (DSLs) like TidalCycles [4]. In short, in a live coding session, the live coders use DSLs as a client to control the real-time sound synthesis in the SuperCollider server. For our need, instead of using SuperCollider to output real-time audio signals, we use it for non-real-time audio synthesis.

As for the client, we choose to write it in Python because several deep learning frameworks (such as PyTorch [5]) have been implemented in Python, and the Python module Gym is one of the most important benchmarks for deep reinforcement learning. By designing the client part in Python, we can follow the Gym interface and connect with a deep learning framework, while we move the interaction part (the audio synthesis) to the SuperCollider server. With the help of Open Sound Control messages, we link the neural network training with the audio synthesis (see Fig. 2).

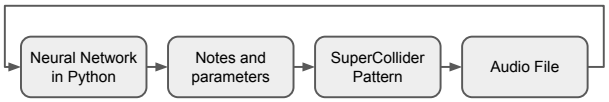

Figure 2. Python-SuperCollider communication: a neural network (agent) is trained in Python; it sends symbolic music representations(e.g. notes and synthesiser parameters) as Open Sound Control messages to the SuperCollider pattern; then the pattern will be synthesised to an audio file in non-real-time and sent back to Python as the input of the neural network, forming an iteration.

4.2 Code implementation

The pseudo-code of the implementation is as follows:

1 Use *make* function in the client to create the required environment, which will send a message to the server, asking the server to load related music patterns, synthesise an empty file and return the address of the file to the client side. On receiving the returning message, the client should read the action space and the observation space.

2 Send the *reset* message to the server side. Empty the observation space if it is not.

3 According to the observation space, decide what action to take. Send the *step* message to the server side with chosen actions in each step. The server will do non-real-time synthesis in each step according to the given

[2] https://github.com/chaosprint/RaveForce
[3] https://supercollider.github.io
[4] https://tidalcycles.org
[5] https://pytorch.org

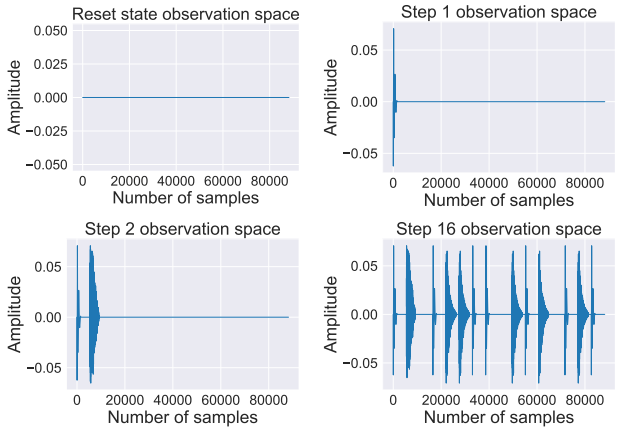

Figure 3. The observation space in each state has the same length. For instance, the step 1 only has the audio information for the first 16th note, but it is padded to have the same length as the reset state (about 90000 samples).

action message. Also, the server should return the client with the synthesised file address.

4 The client should use the address to load the currently synthesised sound file and set it as the observation space. Calculate the reward by comparing the generated audio file with the target audio file.

5 Send the reward back to the client for updating the neural network.

6 Repeat from Step 3 until the limit of episode length is reached

4.3 Optimisation

In the implementation, a unique strategy is designed for the observation space. As neural networks typically require a fixed length input, the observation space needs to be padded to have the same length in every step. Hence, in the initialisation stage, we require SuperCollider to generate an empty full-step (16-step by default) long audio file corresponding to the beats per minute (BPM) parameter. The length of this empty file will be set as the *total-length* attribute. In the following steps, though the actual output of the audio file varies in length, it will be padded with zeros to match the total-length attribute. With this strategy, the observation spaces in each step can share the same length (see Fig. 3).

5. TASK DESIGN AND EVALUATION

After implementing the environment, it is necessary to examine what kind of tasks it can handle and evaluate how the environment performs with the given task.

5.1 Challenges with the reward function design

The reward function in reinforcement learning measures how well the agent chooses the action in the current step. Its design is challenging for music generation, especially in those tasks whose evaluation criteria are subjective. It can be feasible to evaluate the similarity between the generated music piece and the songs in a music corpus. At the same time, pursuing similarity in music can lead to plagiarism, which is an essential issue to address [23].

Currently, we provide four criteria for evaluation: (1) mean square error (MSE) of all the samples; (2) MSE of the Mel-frequency cepstral coefficients (MFCCs); (2) MSE of the short-time Fourier transform (STFT) coefficients, both real and imaginary parts; (4) MSE of the constant-Q transform (CQT) coefficients, both real and imaginary parts. These four criteria are used to measure the similarities between two audio files. Also, as the whole programming framework is customisable, it can be connected with other criteria, e.g. a well trained neural network that can grade a music file.

5.2 The example tasks

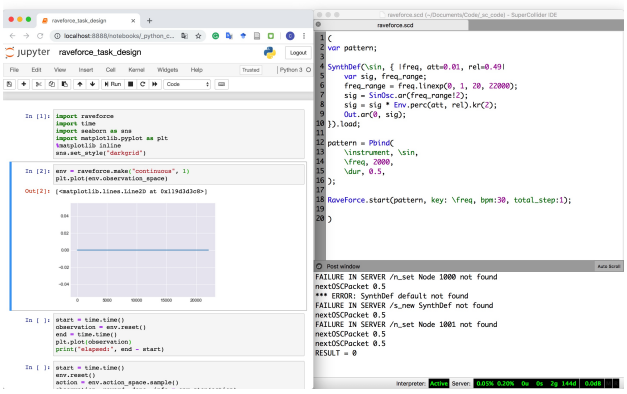

Figure 4. RaveForce workflow: first run SuperCollider code, and then open Python IDE (e.g. Jupyter Notebook) to train the agent.

In RaveForce, the music task should be defined by the user (see Fig. 4). We provide two music examples to explain the environment better.

5.2.1 Drum loop imitation

The example task *drum-loop* uses music samples from three drum components (kick drum, snare drum, and hi-hat) to imitate the target drum loop as much as possible. The action space in the example is a discrete set that contains all eight possible combinations in each note from which the agent should choose one action, and a reward will be calculated according to the choice (see Fig. 5).

Different from some other reinforcement tasks, the reward in this task is precisely the state value function. If we use Deep Q-learning (DQN) for this task, the Q function in each step can be calculated as follows:

$$Q^{\pi}(a|s) = V(s_{t+1}) - V(s) \tag{1}$$

Also, as a specific drum combination only has a fixed reward, we can use the traditional dynamic programming method to find the best drum pattern in this case.

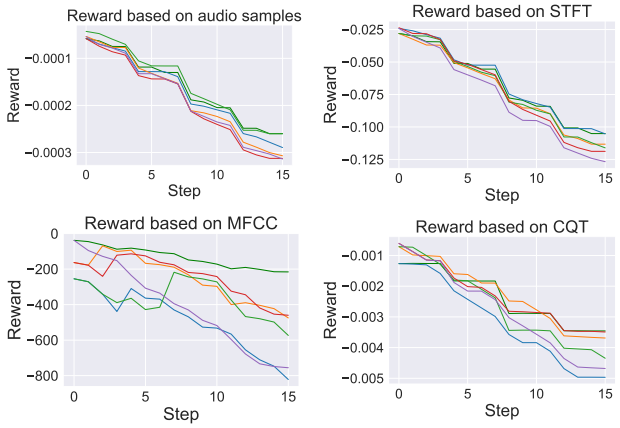

Figure 5. Drum loop combination reward with different criteria. The green line represents the reward of the optimal drum combination which is closest to the target drum loop while the rest are random combination rewards. The MFCC criterion tends to outperform others in this task.

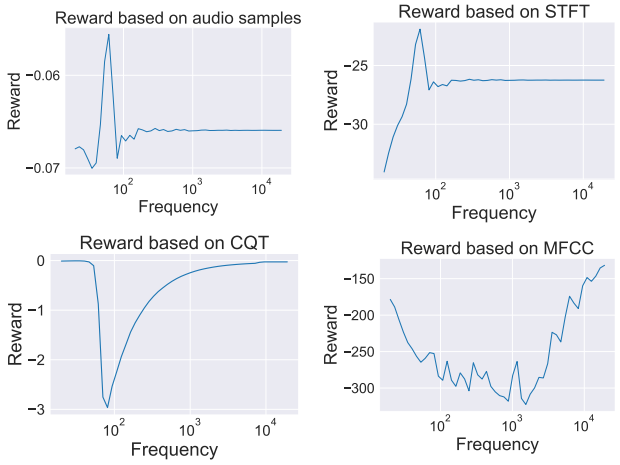

Figure 6. Different criteria for kick drum simulation task. MFCC and CQT tend to show poor performance in this task.

5.2.2 Kick drum optimal frequency search

In this example, we aim to use a sine wave oscillator, controlled by an amplitude envelope to simulate a kick drum audio sample. To make it easier for visualisation, we fix the envelop shape and make the frequency of the sine wave oscillator the only controllable parameter. The relationship between the frequency and the reward is shown in Fig. 6. The *total-step* attribute in SuperCollider can be set to one, which makes the pattern become a single note. In each iteration, the parameter updating of the whole loop is done for this single note. Also, the example can be extended to more parameters and more steps.

With the frequency-reward distribution, we can use the neural network to search for the optimal frequency. First, we train a critic-network which takes the frequency as input and predicts the reward. When connected with the critic-network, an actor-network can be trained until it converges to the optimal frequency.

5.3 Evaluation

We will evaluate the environment from two angles: (1) whether the environment is fast enough for the training; (2) if the symbolic-to-audio conversion can help the music generation.

As a support to our method, the programming framework implementation is the focal point of this paper. In previous sections, we have introduced our environment design and the optimisation we have made, which makes it feasible to use audio evaluation methods for symbolic generation within an acceptable running time. To illustrate, we provide the running time of a 16-step task in one episode (see Fig. 7), which is calculated with the *drum-loop* task mentioned above.

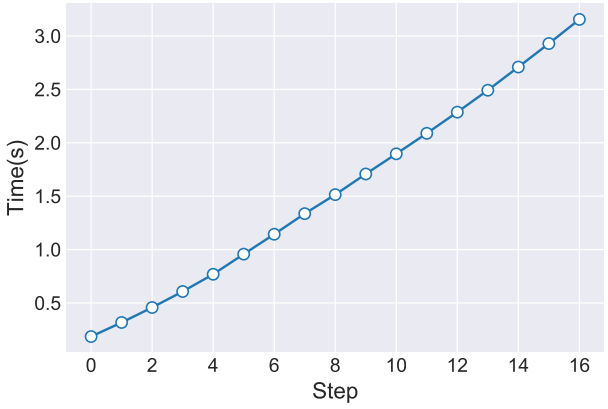

Figure 7. The running time of RaveForce example task *drum-loop*. Step 0 refers to the reset state and some time will be spent for calculating the total-length. All 16 steps will take around 3.2 seconds on an Apple MacBook Pro 13-inch (Mid 2017, i5, without Touch Bar).

Regarding the quality of the music, there are still some uncertainties, for the generated music quality may change with different algorithms, tasks and music genres. Currently, limited by computational resources, we focus mainly on the programming framework implementation, and only pay particular attention to electronic music loop or note.

Also, it is arguable that the predefinition of synthesiser architecture can be a limitation of music complexity. However, this trade-off is significant to our proposed method. With a fixed transforming function, for example, the neural network will no longer need to organise all the audio samples to form an audio waveform which is aurally similar to an FM synthesis tone. Instead, the computational resources can be used to focus on optimising the parameters of a predefined FM synthesiser. This trade-off may even bring new possibilities in music creation because mismatching the target tone with a random synthesiser architecture can potentially generate a tone which is similar but slightly different from the target.

6. CONCLUSION

In this project, we propose a new music generation design that employs deep reinforcement learning, and we have im-

plemented an environment for testing the design. It follows the OpenAI Gym interfaces but moves the interaction to SuperCollider. It turns out that the SuperCollider is fast enough in non-real-time audio synthesis, which makes the reward calculation and the neural network training feasible. Meanwhile, there are some uncertainties if this method can improve the music generation, which should be tested with different tasks, algorithms and music genres. It can be one of our future directions. Nevertheless, the whole implementation produces an environment for researches to explore new algorithms for music generation tasks, e.g. music sequence generation or timbre parameter searching. It provides a new perspective to music generation, especially for those tasks in which users can find a determined reward function.

Acknowledgments

This work was partially supported by the Research Council of Norway through its Centres of Excellence scheme, project number 262762 and by NordForsks Nordic University Hub Nordic Sound and Music Computing Network NordicSMC, project number 86892.

7. REFERENCES

[1] S. Dieleman, A. van den Oord, and K. Simonyan, "The challenge of realistic music generation: modelling raw audio at scale," in *Advances in Neural Information Processing Systems*, 2018, pp. 8000–8010.

[2] D. Herremans, C.-H. Chuan, and E. Chew, "A functional taxonomy of music generation systems," *ACM Computing Surveys (CSUR)*, vol. 50, no. 5, p. 69, 2017.

[3] J.-P. Briot, G. Hadjeres, and F. Pachet, "Deep learning techniques for music generation-a survey," *arXiv preprint arXiv:1709.01620*, 2017.

[4] I. Fujinaga, A. Hankinson, and L. Pugin, "Automatic score extraction with optical music recognition (omr)," in *Springer Handbook of Systematic Musicology*. Springer, 2018, pp. 299–311.

[5] G. Hadjeres, F. Pachet, and F. Nielsen, "Deepbach: a steerable model for bach chorales generation," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 1362–1371.

[6] I. Simon and S. Oore, "Performance rnn: Generating music with expressive timing and dynamics," 2017.

[7] A. Van Den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *CoRR abs/1609.03499*, 2016.

[8] N. Kalchbrenner, E. Elsen, K. Simonyan, S. Noury, N. Casagrande, E. Lockhart, F. Stimberg, A. v. d. Oord, S. Dieleman, and K. Kavukcuoglu, "Efficient neural audio synthesis," *arXiv preprint arXiv:1802.08435*, 2018.

[9] B. L. Sturm, O. Ben-Tal, U. Monaghan, N. Collins, D. Herremans, E. Chew, G. Hadjeres, E. Deruty, and F. Pachet, "Machine learning research that matters for music creation: A case study," *Journal of New Music Research*, vol. 48, no. 1, pp. 36–55, 2019.

[10] S. Mehri, K. Kumar, I. Gulrajani, R. Kumar, S. Jain, J. Sotelo, A. Courville, and Y. Bengio, "Samplernn: An unconditional end-to-end neural audio generation model," *arXiv preprint arXiv:1612.07837*, 2016.

[11] C. Donahue, J. McAuley, and M. Puckette, "Synthesizing audio with generative adversarial networks," *arXiv preprint arXiv:1802.04208*, 2018.

[12] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[13] I. Xenakis, *Formalized music: thought and mathematics in composition*. Pendragon Press, 1992, no. 6.

[14] N. Collins, "Reinforcement learning for live musical agents." in *ICMC*, 2008.

[15] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[16] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[17] M. Dorfer, F. Henkel, and G. Widmer, "Learning to listen, read, and follow: Score following as a reinforcement learning game," *arXiv preprint arXiv:1807.06391*, 2018.

[18] G. Wang, "A history of programming and music," *The Cambridge Companion to Electronic Music*, pp. 55–71, 2007.

[19] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.

[20] A. McLean and R. T. Dean, "Musical algorithms as tools, languages, and partners," *The Oxford Handbook of Algorithmic Music*, p. 1, 2018.

[21] J. McCartney, "Rethinking the computer music language: Supercollider," *Computer Music Journal*, vol. 26, no. 4, pp. 61–68, 2002.

[22] M. Wright, "Open sound control: an enabling technology for musical networking," *Organised Sound*, vol. 10, no. 3, pp. 193–200, 2005.

[23] A. Papadopoulos, P. Roy, and F. Pachet, "Avoiding plagiarism in markov sequence generation." in *AAAI*, 2014, pp. 2731–2737.