UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Ingvar Baranin

Generating Real Time Adaptive Game Music

Bachelor's Thesis (9 ECTS)

Supervisor: Anna Aljanaki

Tartu 2021

Generating Real Time Adaptive Game Music

Abstract:

This thesis aims to construct and train an LSTM-based machine learning model that can automatically generate real time adaptive video game music. In order to demonstrate its capabilities, a game using the model's music is developed. A review of similar existing works and the methodology of the thesis is provided, as well as the analysis of the final product.

Keywords: Video game music, artificial intelligence, LSTM

CERCS: P176 Artificial intelligence

Adapteeruva arvutimängumuusika genereerimine reaalajas

Lühikokkuvõte:

Bakalaureusetöö eesmärk on luua ja treenida LSTM-põhine masinõppe mudel, mis suudab reaalajas genereerida reaktsioonivõimelist arvutimängumuusikat. Mudeli võimekuse hindamiseks luuakse töö käigus mäng, mis kasutab reaalajas genereeritud muusikat. Töö uurib varasemaid sarnaseid lahendusi, kirjeldab muusika genereerimiseks vajalikke samme ning analüüsib lõpptulemust.

Võtmesõnad: Arvutimängumuusika, tehisintellekt, LSTM

CERCS: P176 Tehisintellekt

Table of contents

Introduction

Most have felt the power with which music can convey an emotion or immerse one in a world. Communication mediums that do not use this power to at least some extent are few and far between, simply because music adds a whole other dimension of sensation to a given experience [1]. One medium with a possible advantage of using this dimension, considering its interactive nature, is video games.

The presence of music in a video game is, in effect, mandatory. Games without music are nearly impossible to come by, and with good reason: a study by Klimmt et al. [2] has shown that a player's enjoyment of a game notably increases when there is a soundtrack present.

However, the task of creating video game music is not an easy one. The development of a game contains several other features such as programming, writing the plot, drawing the art, and designing the levels – all of which take a considerable amount of time [3]. Composing a unique soundtrack on top of all these features requires significant skill and iteration: Hoffman points out that the majority of composers write at most 10 minutes of music a day [4].

Thus, with the appropriate knowledge, it is still unlikely that the soundtrack's length will come close to the length of the gameplay, as the financial cost of such compositional undertaking is steep. Plut and Pasquier [5] showcase a clear example: if the composers of *Pillars of Eternity* had created unique music for the entirety of the 60-hour-long game, the budget of the project would have been approximately twice as big.

The first aim of this thesis is to address the costly problem by creating a machine learning model, which would make the described process of composing simpler by generating music automatically. The second aim is to create a demo-game that dictates different parameters of the generated music, thus illustrating the potential of said model.

The thesis consists of five chapters. The first chapter covers required background knowledge. The second chapter gives an overview of how to generate music. The third chapter outlines the design reasons behind the game and related work. The fourth chapter, in a sense, brings together the previous two as it explores integrating music and gameplay. The fifth and final chapter analyses the results and feedback on the game and proposes ideas for future work.

1. Background

1.1 Music

As Andrew Kania points out, the exact definition of music is a complicated philosophical question, but when simplifications are made, it can be said that music is organized sound [6]. The goal of this work agrees with this definition, in the sense that it aims to organize sound in a way pleasing to the listener – that is the player.

The possible set of sounds, along with their duration and rhythm, structured by the machine learning model spans the 88 keys of the piano. These keys cover all of the unique musical notes used in the digital representation of music shown in chapter 2.3, which will be the basis of training for the model.

1.2 Digital representation of music

Training a machine learning model requires data. Musical data can be represented in many ways, the most common among musicians being sheet music, but getting a computer to comprehend a musical piece by presenting it on a sheet of paper is difficult. Optical music recognition, as this field of study is called, is a problem on its own [7].

Audio waveforms are another popular representation used in music information retrieval, which capture music in a way most similar to human hearing [8]. However, in contrast to waveforms, which include low-level audio features, there is a friendlier approach: MIDI files. These files describe high-level musical events, making the process of meaningful encoding simpler.

1.3 MIDI

MIDI (Musical Instrument Digital Interface) is a standard that helps digital instruments communicate with each other and, more importantly, with computers. The standard defines a set of messages - specifically two types of message bytes: status bytes and data bytes [9] - that can be sent from one device to another.

The messages describe, among other metadata, various musical performance-related events: which note is pressed, when is it pressed, how loud is it, when is the note released, and so on.

MIDI (.mid) files, similarly, contain sequences of such events, making them a great candidate for input data.

1.4 Long Short Term Memory (LSTM)

Recurrent neural networks (RNN) are made to handle sequence-based predictions [10]. In more detail, RNNs use past input sequences to influence its current output prediction [11]. Intuitively, this is perfect for music, as music contains a lot of repetition and variation on past melodies.

However, there is a constraint: basic RNNs have difficulties learning long-term dependencies, thus limiting the distance of how far back the neural network can look to impact its current prediction output [11]. This property is especially problematic in the case of music, which relies on repetition to introduce coherence into musical storytelling. Losing the long-term structure denies music one of its primary expression mechanisms. Long Short Term Memory, proposed by Hochreiter and Schmidhuber [12], is a type of RNN architecture that remedies this issue. As Olah explains [13], LSTMs have an internal cell state, which makes use of three gates (pictured on Figure 1):

- forget gates that decide which information the cell state forgets or retains
- input gates that decide which information updates the cell state
- output gates, which determines the output based on the cell state

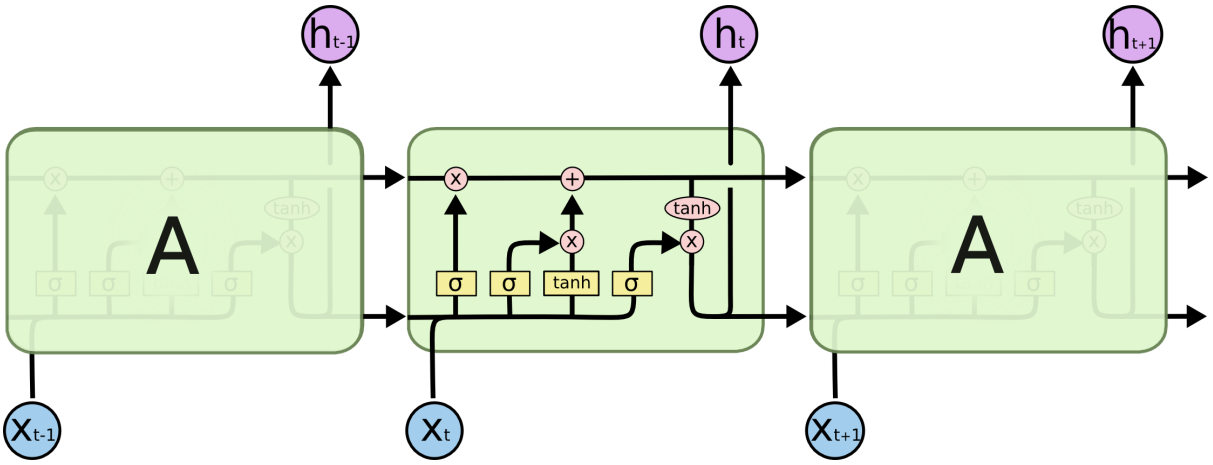


Figure 1. An unfolded LSTM showcasing its gates. σ (left to right) denotes the forget, input and output gates [13].

The information stored and regulated by these gates enhances the ability to attend to the context previously seen, enabling music to be coherent.

1.5 Related work

Hutchings and McCormack [14] point out that while a lot of commercial video games use music that adapts to the actions and surrounding world of the player to some degree, they are rarely generated in real time: rather, layers of prerecorded soundtracks are used and played depending on a programmatically stated set of rules. There are, however, some games with real time music generation and even integratable tools that aim to supply such music. This subchapter overviews two works of that kind.

1.5.1 Escape Point

Escape Point, a 3D science fiction horror game developed by Anthony Prechtl, has the player navigating a dark maze while also evading enemies [15]. During gametime, a local UDP connection is made to communicate with a music generator program [15]. The generator is a Markov model, which, on the basis of its current state and input provided by the game, calculates the next most probable set of notes to play [15]. The Markov model is also capable of accounting for the emotion needed to be conveyed during gameplay: during more tense moments of the game, like the proximity of an enemy, the music gets more tense [15].

However, it should be mentioned that the construction of the given Markov model requires a significant amount of domain knowledge as the probabilistic weights of chord progressions within the model are calculated with music theory in mind [15].

1.5.2 Adaptive Music System (AMS)

The Adaptive Music System, a novel tool produced by Hutchings and McCormack, proposes using a spreading activation model [14]. The activation model maps relationships (edges) between a discrete set of emotions and different objects and environments of a game (vertices), all of which are preferably put in place during game development [14]. Real time communication between the game and the model enables the model to understand the emotional context of a game, which in turn is passed on to RNNs, which develop pre-composed melodies [14].

The tool was tested by implementing it on two games that allow capturing their inner game states – *Starcraft II* and *Zelda Mystery of Solarus* – and conducting a study among the implementation's players [14]. As a result, players reported higher immersion in the game [14].

2. Generating music

This chapter presents the knowledge necessary to generate music automatically.

First, the retrieval of MIDI files is discussed. Further on, the importance of a thought-out textual encoding of said files is shown, alongside an example of such encoding. The end of the chapter covers the training process and use of the LSTM model.

2.1 Dataset

All the MIDI data used in this thesis is from vgmusic.com[1], specifically from the piano only section. Many similar datasets can be found on the internet with various composers, categories and genres.

2.2 Downloading MIDI files

It may turn out that downloading files from an internet database is not straightforward. Machine learning tasks, much like the one undertaken here, require hundreds of files, but websites might not have a separate link to download multiple files at once. This proves to be a problem with the current data set source, too.

A solution also used in previous music-generation works, for example by Kalingeri and Grandhe [16], is to use a web scraper.

A web scraper is a script in which it is possible to iterate over and interact with certain elements of a webpage. This approach of automatically clicking through only the dictated parts of a page makes downloading MIDI files quick.

For writing the scraper, the author used Selenium with Python[2]. Navigation for the sought-after links can be achieved without Selenium, but using it simplifies the process considerably, as there are specific methods aimed at doing this[3].

2.3 Pre-processing MIDI data

Music generation with neural networks, on a more specified level, means modelling a statistical understanding of music: given a sequence of musical events, the model should predict the next most probable event. Training an adequate statistical model like this would be impossible if the training data consist solely of plain MIDI files, since, as Back's paper on

[1] https://www.vgmusic.com/
[2] https://selenium-python.readthedocs.io/
[3] https://selenium-python.readthedocs.io/locating-elements.html

the MIDI specification [17] shows, they also contain a lot of information unrelated to the task at hand. Thus, pre-processing is necessary, as it will help the neural network generalize.

A toolset for the Python language - Music21[4] - provides useful functions for manipulating and pre-processing .mid files. Using these functions, the dataset of 721 MIDIs is encoded into text-based sequential data. This encoding, similar to the one used by Musenet [18], arranges performance-related events into tokens, where each token can be arranged into one of seven categories. The categories and the number of unique values in each category are as follows:

- 95 note-on events[5]
- 95 corresponding note-off events
- 80 pause (wait) events, each with a different duration
- 25 tempo events
- 8 velocity (loudness) defining events
- 4 time-signature defining events
- a start and end event

These tokens are extracted by iterating over all events in every MIDI file based on time (Music21 keeps track of time by "offset", which is the time passed since the start of the given piece) and checking which Music21Object[6] corresponds to the current event. If any events of interest, such as the ones previously listed, are met, it is added as a token. However, Music21 does not explicitly state note-off events. These are instead extracted by remembering the duration of a note in a note-on event and declaring a note-off event when the duration has passed.

After encoding, an example token-sequence describing a piece of music looks like this (a sheet music equivalent is shown on Figure 2):

```
start tempo:120.0 timesig:4/4 velocity:112 note:c4 wait:2.0
 note:c4:off note:d4 wait:2.0 note:d4:off note:e4 wait:2.0
 note:e4:off note:f4 wait:2.0 note:f4:off note:g4 wait:2.0
 note:g4:off note:a4 wait:2.0 note:a4:off note:b4 wait:2.0
       note:b4:off note:c5 wait:2.0 note:c5:off end
```

[4] What is Music21? http://web.mit.edu/music21/doc/about/what.html
[5] It turns out that the piano-only dataset contained a few notes outside the normal range of a piano (88 notes).
[6] Music21Object https://web.mit.edu/music21/doc/moduleReference/moduleBase.html#music21object

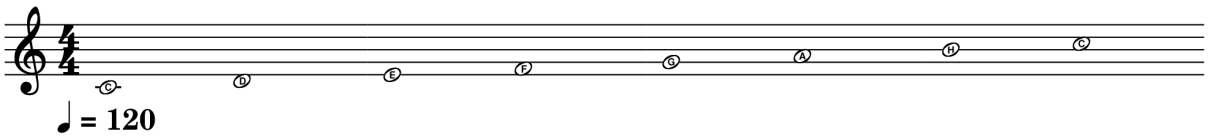


Figure 2. Sheet music corresponding to tokens

Other encodings could be used, but the efficiency of this one is ensured by the fact that there is a small number of unique tokens. For an example, consider the following variant:

```
start tempo:150.0 timesig:4/4 note:B-2:v127:d0.25 wait:1.0
note:B-2:v127:d0.25 wait:1.0 note:B-2:v127:d0.25 wait:1.0
note:B-2:v127:d0.25 ...
```

Here, the wait tokens only keep track of moments of complete silence, minus the note duration before it, and note-on tokens include their velocity and duration. Clearly, the number of unique tokens increases by thousands. The more unique tokens there are, however, the harder it is for the resulting statistical model to generalize, because it considers two of the same notes with different durations and/or velocities as two completely different event types.

Next, all the sequences of tokens are transformed into sequences of integers, such that each unique token corresponds to a unique integer from 1 to 311 (one for 309 unique tokens listed before, including one out-of-sequence token).

Finally, a sliding window algorithm is applied to all integer sequences to form model inputs consisting of 50 integers and model outputs, which are single integers. Thus, the model is trained to predict the next token in a sequence. The inputs and outputs are written to storage as a .txt file.

After preprocessing, the model can use pre-processed data as input.

2.4 Training

The LSTM implementation used in this thesis is provided by TensorFlow's keras.layers module[7]. The full model structure along with the shapes used is provided in Appendix I. The architecture is a modified version of Skúli's work [19].

The model was trained for 24 hours on a Nvidia GeForce 1060 6GB graphics processing unit (GPU) to a softmax loss of ~0.50. Training on a GPU with TensorFlow requires a CUDA-enabled video card and specific CUDA drivers as detailed on TensorFlow's website[8].

[7] LSTM layer - Hochreiter 1997 https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTM
[8] https://www.tensorflow.org/install/gpu

The setup is also mandatory to get real time predictions during gameplay. This work uses CUDA 11.

During training, the input along with its target, is one-hotted and served to the model by a batch generator with a batch size of 512. The generator is necessary because the entire dataset is too big to be read into memory all at once, especially when it is one-hotted.

To avoid overfitting, which would teach the model to essentially play the same music it trained on note-for-note, dropout layers are used[9].

After training, the model is saved as a .h5 binary file – this file represents the trained model that will be loaded during the game and used for real time music generation.

2.5 Continuous music generation

It is also important to consider how the trained model is actually used to generate a continuous sequence of tokens.

As described before, the model predicts one musical event for 50 preceding musical events. In order to make the next prediction, the previous prediction is to be added to the end of the input sequence. To account for the new length of the input sequence, which is 51 tokens, one prediction is removed from the beginning of the input sequence in a sliding window fashion. To keep the music changing and the input sequence at a constant length of 50, this action, also illustrated on Figure 3, is done upon every prediction except the first.

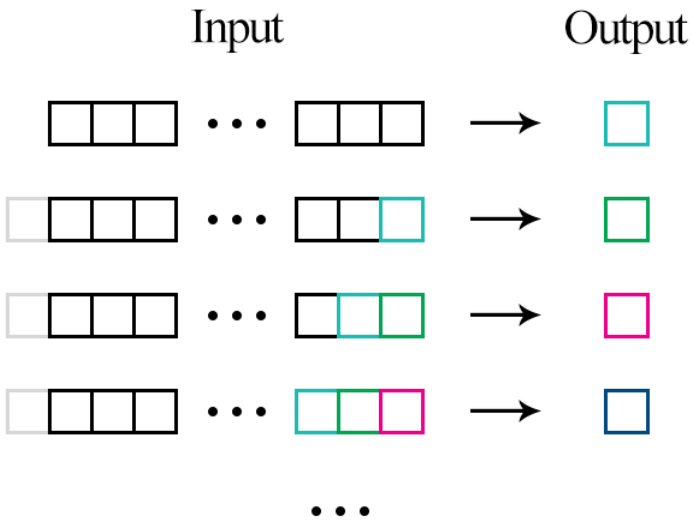


Figure 3. Illustration depicting the appending and removal of tokens in the input sequence

[9] https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dropout

Despite this, it can happen that the model starts predicting the same note-on and note-off tokens constantly, with little to no rhythmic variation, due to getting stuck in a local minimum. A solution to this is found by, before prediction, checking if the input sequence contains more than 15 unique tokens. If it does, the model can continue to predict in the aforementioned way. If the opposite is true, then the model is instead asked for the top three most probable tokens. By choosing a random one from these three, the likelihood of single-note repetitions is made minimal and the model is nudged to move along the possible space of solutions.

3. Creating a game

This chapter covers the mechanics and development of the demo-game that dictates the parameters of music generation concurrent with gameplay.

3.1 Tools

The game was developed using Unity, a game engine which has been the building block of several commercially successful games[10]. Unity provides tools for faster game development: features such as graphics rendering and physics handling.

All of the visual art assets in the game are original, drawn in Aseprite: a pixel art drawing program[11].

3.2 Gameplay and objective

While game thematics that could make use of various music generating parameters are essentially infinite, the author makes the decision based on simplicity: a 2D side-scroller platform game. The simplicity proves useful in the player feedback study in Chapter 5, as even players that are not familiar with video games quickly get used to the simple gameplay.

On launch, the player is shown the main menu, pictured on Figure 4. Here the player has three interactable elements: either start or quit the game, and a music type dropdown. The dropdown will be discussed further in the next chapter.

Figure 4. Main menu of the game

[10] A list of notable games made with Unity: https://unity.com/madewith
[11] Pixel art tool: Aseprite https://www.aseprite.org/

Once the game has started, the objective is to get as far as possible in the game-world's horizontal space, where the terrain is procedurally generated. During gameplay, the view of the game-world is constantly sliding to the right and the target for the player is to be quick enough to not fall out of sight of this panning scene. If the player fails to do so, their position is reset to the beginning. The objective is made harder by restricting the width of the game-world, thus narrowing the space the player can move in.

The further the player gets, the higher is their score. The score is kept track of by two scoreboards: a smaller one, which shows the current score and a larger one, which shows the best score during the current playing session. The scoreboard, along with the rest of the game, can be seen on Figure 5.

Figure 5. Gameplay depiction

The player can be controlled on the keyboard by either the left and right arrow keys or by pressing A and D, correspondingly moving the player left or right. The player can also jump by pressing the space key and sprint faster by pressing the shift key.

3.3 Scripting

All of the programming controlling the game, as in most Unity projects, is done in the C# language. Using Unity components, rigidbodies[12] and box colliders[13], the game engine

[12] https://docs.unity3d.com/ScriptReference/Rigidbody2D.html
[13] https://docs.unity3d.com/ScriptReference/BoxCollider2D.html

handles all the player collisions and gravity. Similarly, Unity components[14] are used to manipulate UI elements, such as the buttons in the main menu and scoreboards during gameplay. However, some mechanics need custom scripts. Scripts are written for: horizontally panning the camera, moving the character, procedurally generating the terrain and managing the parallax movement of the background. These scripts, along with all other game files can be found in the author's GitHub repository[15].

[14] https://docs.unity3d.com/Manual/UIToolkits.html
[15] https://github.com/IngvarBaranin/music-unity

4. Combining music and gameplay

Playing music that adapts to gameplay in real time requires game states to adapt to, and a communication interface which transfers the change of these game states to the generator. This chapter covers these problems along with detailing how the music is performed.

4.1 Communication between Python and C#

The game mechanics are written in C# and music generation is done in Python. Due to the programming language difference, the author used an existing socket communication solution[16].

When the game is started, a Python file acting as a server running the music generator is also automatically launched. The Python server opens a port and starts listening for a client connection. A C# script running in the background of the game as the client connects to the port and lets the server know that it is ready to receive string-encoded musical data. The server does so by sending a sequence of 50 tokens for every Unity request. Once received, Unity holds these sequences in a queue, playing them in a first-in, first-out order.

4.2 Unity synthesizer

In order to play the music sent by the server, a synthesizer that understands MIDI events is required. This is handled by the MidiStreamPlayer[17] of Midi Player Tool Kit (MPTK), a third-party Unity asset.

MPTK takes in a list of MPTKEvents, where each event, similarly to a MIDI event, is defined by a pitch value, duration, velocity and delay. However, as hinted by the delay, MPTK does not account for time in the same way as Music21's offset does. While the offset in Music21 is based on note durations, MPTK offset – the delay – is set by milliseconds.

Fortunately, Music21 is able to convert its offset to seconds, making the conversion to milliseconds simple. The tokens arriving to Unity thus look like this:

```
76:0.0:0.5:69 52:0.0:0.25:57 64:0.25:0.25:52 71:0.5:0.25:59
52:0.5:0.25:59 72:0.75:0.25:58 64:0.75:0.25:58
```

Each token defines four colon-separated numbers required to initialize a MPTKEvent: the pitch, offset in seconds, duration in seconds and velocity.

[16] Two-way communication between Python 3 and Unity (C#) - Y. T. Elashry
https://github.com/Siliconifier/Python-Unity-Socket-Communication
[17] https://paxstellar.fr/midi-file-player-detailed-view-2-2/

4.3 Music defining game components

In order for the game to demonstrate the adaptability of the music generator, it is best for the game components to consider all of the musical parameters that can be modified. The modifiable parameters are:

- tempo
- pitch height
- instruments
- pauses

The gameplay elements that are chosen to control said parameters, in the same order, are as follows:

- player's horizontal movement speed
- player's vertical game world position
- the type of platform the player is currently standing on
- player movement inaction

The player's horizontal movement speed has two states: a running and a faster, sprinting state. When the player is running, the music sent from the generator to the synthesizer is played at its generated speed. When the player is sprinting, the music is played twice as fast by multiplying every MPTKEvent's delay and duration with the inverse of the player's speed. Because this can be calculated by the game itself, the player's speed is not communicated to the server.

The player's vertical position is kept track of in three ranges: top, middle and bottom, as seen on Figure 6. Each of the three ranges has an unique ID, which is communicated to the server. When the player is in the highest range, the server is told to prefer higher notes. This means that from the top three most likely following notes, the server predicts the highest one. The bottom range works similarly, the only difference being the server preferring the lowest note. If the player is in the middle range, then the music generator predicts as it usually would, outputting notes most likely to occur next.

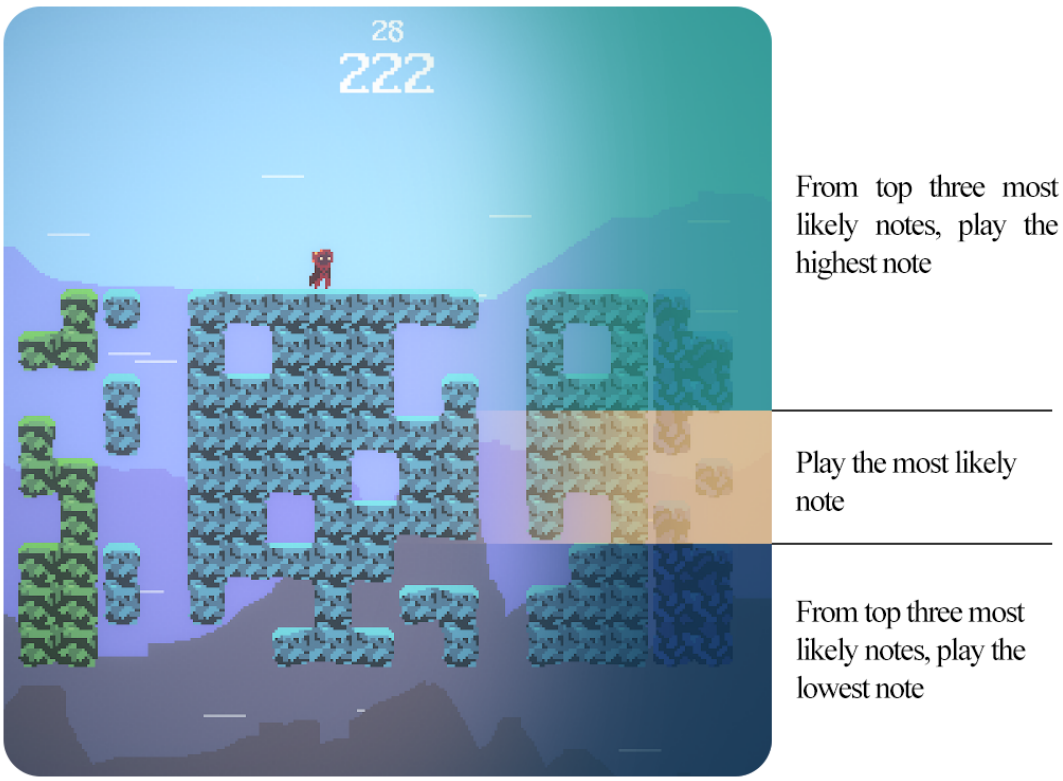


Figure 6. The three ranges tracking the player's vertical position with an explanation of what each range communicates to the music generating server

As first seen on Figure 5, the game features three platforms with different colors. When the player makes contact with these platforms, instruments change. By default, the music is synthesized by an electric piano, which corresponds to deep blue platforms, but when the player steps on a green platform, the instrument changes to a music box. Lastly, light blue platforms switch to a softly-plucked guitar. Changing instruments is also a feature of the Midi Player Tool Kit, meaning that the change does not take place in the server.

The last adaptive method is the pausing of music when the player is not moving. When the player continues moving, so does the music.

4.4 Types of music generation

In order to form a frame of reference to the adaptive real time generated music, the demo game also features a static music generator. Static music is also generated by the server, just as adaptive (dynamic) music, however, it does not control any of the aforementioned parameters. Irrespective of the player's movement or position, the generator does not, at any point, prefer higher or lower notes nor does the game play the music faster or slower.

Additionally, there is the option to turn off the music altogether. The three different options – dynamic, static and no music – can be chosen from the main menu and are used until the player restarts the game.

The next chapter uses these options to analyze which method, if any, the players enjoy the most.

5. Results

Evaluating the quality of the generated music alongside the game, player feedback was gathered based on a short questionnaire. Here, the answers are analyzed and possible future improvements are discussed. Links to the source code can be found in Appendix II.

5.1 Player feedback

After having played the game with three different music generation types – dynamic, static, and no music at all –, five University of Tartu students, aged 21 to 23, were asked to compare all three pairs of different types. The pairs are: dynamic and static music, dynamic and no music, static and no music.

For each pair, similar to Prechtl's work [15], the comparative questions asked were:

- In terms of music or lack thereof, which did you like more?
- Which version of the game felt more exciting?
- Which version of the game felt more challenging?
- Which version of the game was more fun?

During comparisons, if the player did not find any significant differences between the two, they were given the possibility of a third option: answering "I did not notice a difference".

After the comparative questions, other questions follow:

- Do you like video games?
- Have you played similar 2D platformer games before?
- In the game's dynamic music condition, which elements did you like the most?
- Did anything about the game bother you? (Open-ended)
- Do you have any additional feedback? (Open-ended)

The answers to the comparative questions can be analyzed by ranking, a solution proposed by Prechtl [15]. If a certain type of generation was preferred, it gets one point and the unfavored type gets zero points. In case of a draw due to the player not noticing a difference, both of the compared types get half a point.

Using this ranking, the average rank score was calculated for each questioned category, as seen on Table 1.

	Preference	Excitement	Challenge	Fun
Dynamic	1.8	1.9	1.4	1.9
Static	1.1	1.1	1	0.8
No music	0.3	0	0.6	0.3

Table 1. Average ranking (range 0-2) per category for each music generation type

Dynamic music fared better in each of the four categories with static music being in constant second place. No music was considered the worst in all categories. Interestingly, the different types seem to influence the challenge aspect of the game the least, as in this category the rankings are the most similar.

The subsequent questions covered the players' past experiences with video games and gave them the ability to give more specific opinions. These will be discussed next.

Most of the students reported liking video games, with two out of five being indifferent. Four out of five had played similar 2D-side scroller games before.

The most liked music defining game mechanic was the changing of tempo according to the player's speed. This was followed by instrument changes, leaving the preference of higher or lower notes based on the player's vertical position as the least liked aspect. The author believes that this order is mainly influenced by the noticeability of a given mechanic: while music changing its speed is highly perceptible, the generation of higher/lower notes might not occur at all, especially if the player is in the corresponding vertical position for a short amount of time.

In the open-ended questions there was a suggestion that there could be standard pre-programmed game sound effects, for example for jumping and running. This could be considered in a future version, however, currently the focus was on generated music only, meaning that sound effects were intentionally not implemented.

Other suggestions consisted of adding additional inputs for player movement or making the game camera move faster along with the player's movement speed and as such are not key in determining how the generated music was perceived.

5.2 Future work

The biggest shortcoming of the music generator is the fact that for the music to be generated fast enough to be real time, it needs substantial computational power. This means that the machine that the generator is running on requires a CUDA-enabled graphics processing unit. In order to remove this requirement, the generating model needs to be simplified, thus reducing computational needs, therefore enabling the model to be also fast enough on central processing units (CPU). Another option is to try and train the model on a smaller set of possible output tokens, as this may also facilitate computation. Yet another idea is to move the generator to a web server that has access to a GPU, but this may introduce other limitations.

Another improvement which would make the game more accessible is to remove its dependency on Python. Currently, generating the music and encoding it for Unity requires Python's TensorFlow and Music21 libraries, not to mention installing Python itself. To remove the dependencies, more research is needed to find out which C# libraries could cover the functionalities of the aforementioned Python libraries. It would be optimal if these substitutive libraries could be compiled along with the rest of the game's code, as this would make the installation process equal to downloading a single compressed folder.

The adaptiveness of the generated music itself could be improved, too. One idea would be to train two different models, one for music written in major keys and another for music written in minor keys. As a result, the game could control a parameter of emotion by switching between the two models, as major scale music tends to sound happy, while minor scale music sounds sad.

To make music generated during gameplay more accessible, a useful addition would be to add a setting, which, upon enabling, would save the music made during gameplay to storage, as a MIDI file.

Finally, some gameplay ideas provided by players during feedback deserve implementing as well. Generally, these intend to improve player input. These, however, serve a purpose once the previous improvements are solved.

Conclusion

The aim of this thesis was to train a machine learning model that can generate video game music, and to develop a video game that exhibits the ability of the music generator by controlling its musical parameters.

As a result, hundreds of MIDI files containing performances of video game music were encoded and pre-processed to use them as training data for a LSTM-based model structure. The trained model was used in a 2D side-scroller game developed with Unity, where the movement of the player directly influences the composition and dynamics of the generated music.

To evaluate the generated music, players were asked to play the game in separate adaptive, non-adaptive and silent settings, afterwards comparing each setting. Using a ranking system, it was found that the adaptive music condition was consistently reported as the best in each compared category, thus supporting the quality of the generated music.

References

[1] H.H. Britan. The Power of Music. *The Journal of Philosophy, Psychology and Scientific Methods* 1908; 5: 352–357, https://www.jstor.org/stable/2011756 (accessed 30 March 2021).

[2] C. Klimmt, D. Possler, N. May, et al. Effects of soundtrack music on the video game experience. *Media Psychology* 2019; 22: 689–713, https://doi.org/10.1080/15213269.2018.1507827 (accessed 29 March 2021).

[3] A. Gershenfeld, M. Loparco, C. Barajas. *Game Plan: The Insider's Guide to Breaking In and Succeeding in the Computer and Video Game Business*. St. Martin's Publishing Group, 2007.

[4] What is an adequate time frame for writing the score? *Robin Hoffmann*, https://www.robin-hoffmann.com/tutorials/what-is-an-adequate-time-frame-for-writing-the-score/ (accessed 30 March 2021).

[5] C. Plut, P. Pasquier. Generative music in video games: State of the art, challenges, and prospects. *Entertainment Computing* 2020; 33: 100337, https://www.sciencedirect.com/science/article/pii/S1875952119300795 (accessed 4 December 2020).

[6] A. Kania. The Philosophy of Music, https://plato.stanford.edu/archives/spr2014/entries/music/ (accessed 24 March 2021).

[7] K. Keps. Optiline noodituvastus, https://comserv.cs.ut.ee/home/files/Keps_informaatika_2020.pdf?study=ATILoputoo&reference=D7B589C0024028EAC9F62721572609D5430211B8 (accessed 5 December 2020).

[8] Generating music in the waveform domain. *Sander Dieleman*, https://benanne.github.io/2020/03/24/audio-generation.html (accessed 24 March 2021).

[9] B. Moog. Musical Instrument Digital Interface. 1986; 11.

[10] J. Brownlee. When to Use MLP, CNN, and RNN Neural Networks. *Machine Learning Mastery*, https://machinelearningmastery.com/when-to-use-mlp-cnn-and-rnn-neural-networks/ (accessed 28 March 2021).

[11] What are Recurrent Neural Networks?, https://www.ibm.com/cloud/learn/recurrent-neural-networks (accessed 28 March 2021).

[12] S. Hochreiter, J. Schmidhuber. Long Short-term Memory. *Neural computation* 1997; 9:

1735–80.

[13] Understanding LSTM Networks -- colah's blog, https://colah.github.io/posts/2015-08-Understanding-LSTMs/ (accessed 30 March 2021).

[14] P. Hutchings, J. McCormack. Adaptive Music Composition for Games. *IEEE Trans Games* 2020; 12: 270–280, http://arxiv.org/abs/1907.01154 (accessed 31 March 2021).

[15] A. Prechtl. *Adaptive Music Generation for Computer Games*. Phd, The Open University, http://oro.open.ac.uk/45340/ (accessed 31 March 2021).

[16] V. Kalingeri, S. Grandhe. Music Generation with Deep Learning. *arXiv:161204928 [cs]*, http://arxiv.org/abs/1612.04928 (accessed 16 November 2020).

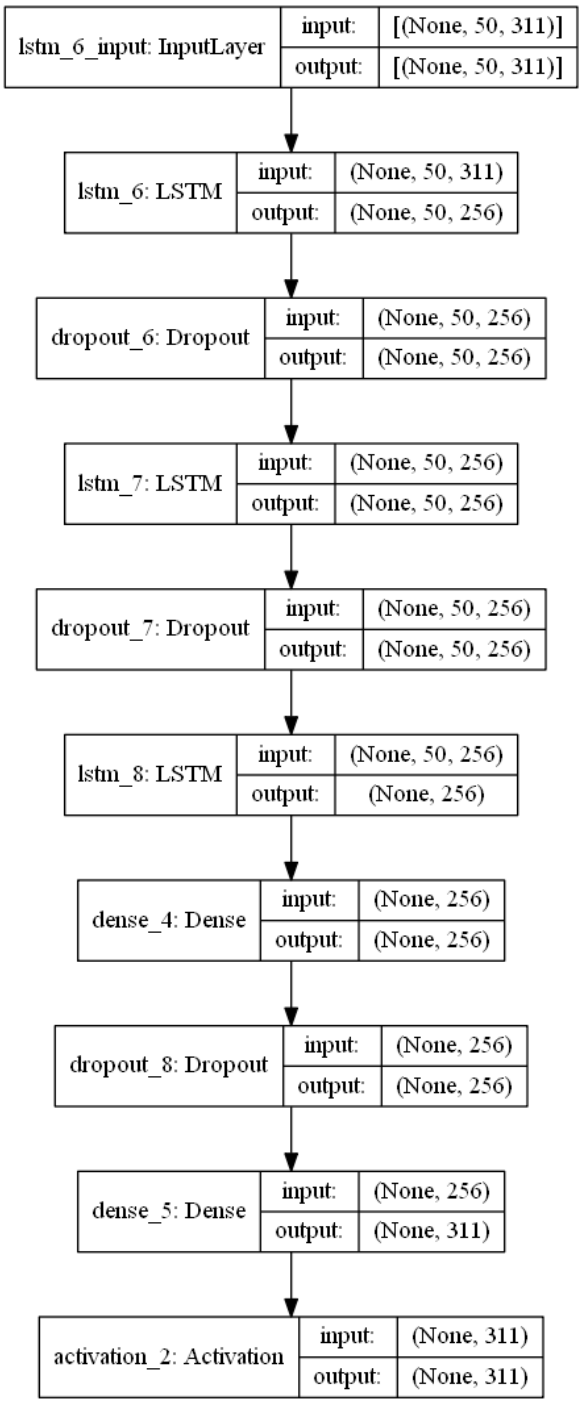
[17] B. David. Standard MIDI file format, updated, http://www.music.mcgill.ca/~ich/classes/mumt306/StandardMIDIfileformat.html (accessed 26 March 2021).

[18] C. Payne. MuseNet. *OpenAI*, https://openai.com/blog/musenet/ (accessed 27 March 2021).

[19] S. Skúli. How to Generate Music using a LSTM Neural Network in Keras. *Medium*, https://towardsdatascience.com/how-to-generate-music-using-a-lstm-neural-network-in-keras-68786834d4c5 (accessed 20 March 2021).

Appendix

I. The full model architecture



II. Installation guide

The game can be downloaded from the author's itch.io page, https://bingvar.itch.io/musicgen, and works on Windows. As mentioned in previous sections, the generator requires Python (version 3.7 or higher) with TensorFlow 2 and Music21 installed.

Additionally, due to slow inference on CPUs, real time music generation requires a CUDA-enabled GPU, alongside CUDA drivers.

The source code can be found on the author's GitHub webpage, where the training process and game resources are in separate repositories:

https://github.com/IngvarBaranin/ai_music_generation
https://github.com/IngvarBaranin/music-unity

III. License

Non-exclusive licence to reproduce thesis and make thesis public

I, Ingvar Baranin,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

Generating Real Time Adaptive Game Music,
supervised by Anna Aljanaki.

2. grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.
3. am aware of the fact that the author retains the rights specified in p. 1 and 2.
4. certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Ingvar Baranin
26/04/2021