University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Marcin Papierzyński
Student no. 345782

Music Style Transfer

Master's thesis
in COMPUTER SCIENCE

Supervisor:
dr hab. Marek Cygan
Instytut Informatyki

October 2019

Supervisor's statement

Hereby I confirm that the presented thesis was prepared under my supervision and that it fulfils the requirements for the degree of Master of Computer Science.

Date Supervisor's signature

Author's statement

Hereby I declare that the presented thesis was prepared by me and none of its contents was obtained by means that are against the law.

The thesis has never before been a subject of any procedure of obtaining an academic degree.

Moreover, I declare that the present version of the thesis is identical to the attached electronic version.

Date Author's signature

Abstract

TODO

Keywords

style transfer, music, midi, neural networks

Thesis domain (Socrates-Erasmus subject area codes)

11.4 Artificial Intelligence

Subject classification

Applied computing
Arts and humanities
Sound and music computing

Tytuł pracy w języku polskim

Transfer Stylu Muzycznego

Contents

Chapter 1

Introduction

Transfer of style between images is a well-known problem. Currently it can be realized between any two images by using convolutional neural networks. The analogous style transfer done for music is a much less explored topic and it is challenging even to define what exactly a musical style is.

In this paper the goal is to define some sensible notion of musical style and create a model that can create a different arrangement for any song based on style extracted from any other song. The music used for training is assumed to be in the MIDI format and can use any range of instruments. The instruments supported by the model must be set during training, but the supported styles do not – we want the trained model to be able to extract style from any song.

Chapter 2

Other works

There exist previous works doing music style transfer to some extent. A few examples are shortly described below. All of them, however, are performing a simpler kind of a style transfer.

2.1. Neural Translation of Musical Style

In Neural Translation of Musical Style [1] authors created a model than can learn to perform any given piano piece in either jazz or classical style. The model does this by regression of notes' velocities (loudness), so it's effectively creating an interpretation of a given piano piece. Therefore it's much simpler, because it only supports one instrument (piano) and cannot create a completely different arrangement.

2.2. Symbolic Music Genre Transfer with CycleGAN

In Symbolic Music Genre Transfer with CycleGAN [2] the described model can create new arrangements but it can still only generate piano and the styles need to be set during training (in this case it's jazz, classic and pop).

2.3. Play as You Like: Timbre-enhanced Multi-modal Music Style Transfer

In Timbre-enhanced Multi-modal Music Style Transfer [3] the model can operate on different instruments, but the styles still need to be set during training (the styles are actually instruments, the authors are using guitar, piano and string quartet). Like in the previous examples, it doesn't allow to transfer style between arbitrary songs.

Chapter 3

Overview and results

3.1. Musical style and composition

For the musical style transfer, the two main notions concerning songs are composition and style. I start by giving a rough definition of them.

Composition is any information about a song that is related to a specific moment in time, e.g. if the music gets louder or some instrument starts playing. The most important part of a composition is the melody (played by one or more instruments).

The style, on the other hand, is information not related to any specific moment in time, e.g. the general mood of the song or the way specific instruments are used (for example, which instruments play the main melodic line and which ones create a backing).

Since music is a form of art, even though it can be understood and formally described, it will always be at least partly subjective. This means that the exact way to split songs into composition and style is not unique (for example, used instruments may naturally be considered part of style, but they can also change in time). The main assumption, however, is that composition-style splitting can be performed in *some* fashion which can be learned by a complex enough model.

Since any song can be naturally interpreted as a time sequence, I will use a recurrent neural network as a model for splitting the input song into composition and style. Composition, just like the input, is a time series, but simpler, with less features. The style, on the other hand, is a single vector representing the whole song. During training, the model splits the input song into composition and style, and then combines them to recreate the original song. Because composition is much smaller than the input song, the model must include useful information in the style vector as well.

After training, I can use the model to extract style from some song and combine it with the composition extracted from other song. The hypothesis is that this approach will allow for a sensible music style transfer.

3.2. Results

TODO

Chapter 4

Framework

4.1. Tools

The whole project is implemented in Python. The machine learning framework I use is PyTorch, along with mido – a Python library for working with MIDI files.

4.2. Data

The dataset I use is Lakh MIDI Dataset[1]. It contains over 100,000 songs in the MIDI format and covers various genres, including pop, rock or classical.

[1] https://colinraffel.com/projects/lmd/

Chapter 5

Data flow

The data flow of the whole architecture is composed of 3 stages: style extraction, predicting song information, and style application.

5.1. Style extraction

The input in the first stage is:

- song information
  - instruments used in a song
  - mode of a song (major or minor)
  - tempo (in beats per minute)
- song content (information about which notes are played in any moment in time for any instrument used in a song)

Mode and used instruments are represented using one-hot encoding. I use 41 most popular instruments in the dataset (percussion and 40 pitched instruments) covering 90% of all the sounds in the dataset. The tempo is a numerical value between 50 and 200 (and is clamped to that interval, if necessary).

The output is composed of 3 parts: melody, rhythm and style. Each of them contains features as learnable embeddings. I will refer to melody and rhythm as composition.

Melody should encode song content (like the input) but with no information about specific instruments (it must combine them). It encodes all pitched instruments used in a song.

Rhythm contains additional information about the melody (but not explicitly related to specific notes) and features of percussion if it is present in the input song. The main reason for introducing rhythm alongside melody is to be able to represent percussion (the only unpitched instrument) but at the same time do not force it if it's not present in the original song (the model should be able to add percussion to a song that originally didn't have it).

I require that there is always at least one pitched instrument used in a song, while the percussion is optional. That way both melody and rhythm are always well-definied (there is at least one pitched instrument they can encode).

Melody and rhythm, along with the style, should enable to recreate all the instruments used in a song.

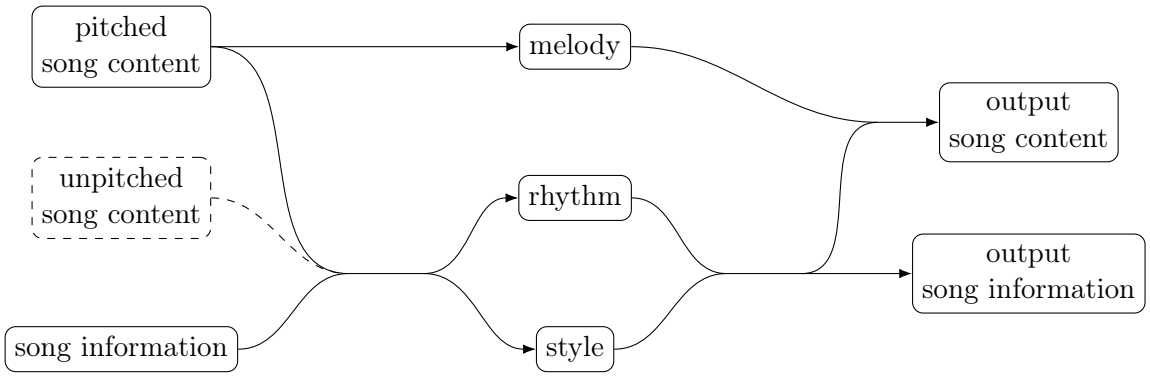


Figure 5.1: Data flow of the model (dashed part is optional).

5.2. Prediction of song information

The input in this stage is style and rhythm. The output is basic information about a song (as in the first stage):

- used instruments (classification)
- mode (classification)
- tempo (regression)

All those outputs should coincide with the respective inputs from the first stage. Note that the set of supported instruments can be arbitrarily large but needs to be specified during training.

5.3. Style application

At this stage the inputs are composition (i.e. melody and rhythm) and style from the first stage. The output is the song content for each of the used instruments. Generated song content should conincide with the input from the first stage. In other words, the model must reconstruct the original song. Figure 5.1 depicts the data flow in all three stages.

Chapter 6

Data representation

In MIDI format, the song is encoded as a sequence of events messages. Messages can contain instructions like setting the tempo or playing a note. Note events are assigned to one of the 16 channels – each of those channels can represent a different instrument.

6.1. MIDI encoding

The MIDI file is encoded as a tensor in the format:

(*batch, channel, bar, beat, beat fraction, note, note features*).

The *channel* dimension refers to a MIDI channel (an instrument). The *bar*, *beat* and *beat fraction* dimensions point to the exact moment in a song when the note is being played.

6.1.1. Specifying time

I assume that the time signature cannot change, so that the number of the beats is constant throughout the song (usually 3 or 4). The *bar* and *beat* dimensions denote in which bar and at which beat the note should play. The *beat fraction* dimension is specifying the exact moment during the beat.

I divide the beat into 8 parts, which gives 8 possible fractions: $0, \frac{1}{8}, \frac{2}{8}, \ldots, \frac{7}{8}$. Independently, I also divide the beat into 3 parts: $0, \frac{1}{3}, \frac{2}{3}$. Combining it, symplifying and sorting, we get 10 different fractions:

$$0, \frac{1}{8}, \frac{1}{4}, \frac{1}{3}, \frac{3}{8}, \frac{1}{2}, \frac{5}{8}, \frac{3}{4}, \frac{2}{3}, \frac{7}{8}.$$

Each *note fraction* coordinate represents one of those fractions. A single beat most commonly has length of a quarter note, which means that fractions with denominator 8 allow to represent up to 32th notes. While division into powers of 2 is the most common in music, many songs also use division into three parts, called triplets. Inclusion of fractions with denominator 3 allows for the exact representation of them, which otherwise would need to be approximated using fractions with denominator 8. More fractions could be added to allow for representing faster melodies, but these 10 are enough for most songs.

The typical way of splitting the beat in neural music generation is to simply divide it into timesteps of equal length, e.g. into 12 parts [4]. The downside of this approach is that it creates many "unnecessary" fractions (like $\frac{1}{12}$), which are not typically used, so they mainly only increase the size of the MIDI file representation. The increase in the size is quite significant, since being able to represent 32th notes using this approach would require

denominator $24 = 3 \cdot 8$, resulting in the whole input being over two times bigger. Here I only introduce the fractions that are necessary to represent triplets ($\frac{1}{3}$ and $\frac{2}{3}$), omitting the rest.

The proposed approach also scales well with addition of more tuplets. For example, adding support for quintuplets (division into 5 parts) would only require adding 4 more fractions ($\frac{1}{5}, \frac{2}{5}, \frac{3}{5}$ and $\frac{4}{5}$), resulting in a constant increase in size ($14 = 10 + 4$ timesteps), while the traditional approach would mean a 5-fold increase in the total size ($120 = 3 \cdot 5 \cdot 8$ timesteps). More complex tuplets are however much less common (and virtually never used in popular music), so I don't use them here.

It's worth noting that in my approach resulting fractions are not equidistant, meaning that it's not justified to apply convolution along the *note fraction* dimension. However, the length of this dimension is not substantial and it's still possible to apply convolution along *bar* and *beat* dimensions, so it's not a big limitation.

6.1.2. Note features

The above format is used for both percussion and pitched instruments. However, the number of notes and note features (lengths of dimensions *note* and *note features* respectively) is different for percussion and for pitched instruments, so two tensors must be used: one for pitched instruments and one (possibly empty) for percussion.

Note features for pitched instruments are:

- velocity (loudness)
- duration
- accidentals (raise or lower the note one semitone)

For percussion the only note features are velocity and duration.

6.2. Notes representation

I use note's span of 8 octaves, each with 12 sounds, which gives 96 possible notes in total. The notes are represented as one-hot encoded vectors with a slight modification. In the standard one-hot encoding, each coordinate in a *note* dimension would correspond to a specific note (so *note* dimension would have length 96). However, this way of encoding would pose a problem for style transfer. Namely, depending on the mode of the song (major or minor), a different set of notes is used. This means that the same song in a different mode would use different sounds in its melody. Hence, with the typical way of encoding notes, the melody representation would partially impose style, which is not desired, since we want to split them.

To remedy that, I encode the notes relative to the scale the song is in. Effectively, what is encoded are not the actual notes, but rather the scale degrees they represent; so the note C in a song in C major would be encoded the same way as the note G in a song in G minor. That way, changing the scale of the song will not affect the melody representation.

This way of encoding only allows to encode notes contained in a given scale (which covers 7 out of 12 notes in each octave). For example, if the song is in C major, we could only encode "white keys". To allow for encoding all the remaining notes as well, each note has additional features (called accidentals) that can raise it or lower it one semitone. Then we can represent all 12 notes in each octave.

The downside of this solution is that it requires the scale of the song to be known – and finding it is in itself not a trivial problem. I use fairly simple heuristics based on the

Krumhansl-Schmuckler key-finding algorithm[1], which predicts the scale of the song based on the frequency of notes used.

6.3. Style and composition representation

Melody is a tensor of shape

$$(\textit{batch, channel, bar, beat, beat fraction, note, features}),$$

which is the same as the input shape but with no *instrument* dimension. The number of note features may also be different.

Rhythm is a tensor of shape

$$(\textit{batch, channel, bar, beat, beat fraction, features}).$$

It's also similar to the input but this time with no *instrument* and no *note* dimensions.

Style is encoded as a regular vector, so it has shape

$$(\textit{batch, features}).$$

The number of features in a melody should be low enough so that the model cannot simply remember all the instruments in the input. Instead, the model will need to learn some compressed high-level representation of the melody, from which it will later be able to reconstruct the input instruments, using the style vector.

[1] http://rnhart.net/articles/key-finding/

Chapter 7

Model

7.1. Notation

The nodes represent tensors, while the edges represent neural network layers – standard fully-connected and activation layers, unless specified otherwise.

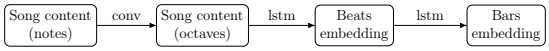
The $\oplus$ symbol is used to indicate elementwise addition (with broadcasting), and the $++$ symbol indicates concatenation of tensors. Since the concatenated tensors often have different shapes, the concatenation is also performed with broadcasting.

7.2. Model description

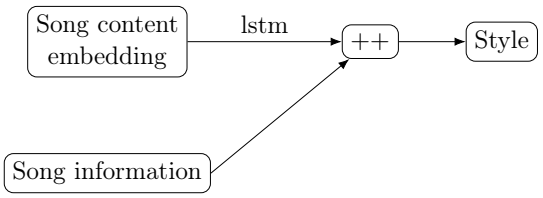
Figure 7.1 shows the architecture of the model components in the style extraction stage. The first step is encoding the input song content using convolutional and LSTM layers (figure 7.1a). The convolution is applied along the *note* dimension, with stride equal to 7 (the number of scale degrees in a single octave). This way, the output of the convolutional layer contains information about specific octaves in the song. Afterwards, two hierarchical LSTM layers are used for further processing. The first LSTM layer is applied along the *beat* dimension (for each bar independently, with beats as timesteps), and the second one along the *bar* dimension. They produce the embedding of the song content which is passed to further submodules.

Style encoder uses an LSTM layer to process the generated embedding into a single time-independent vector (figure 7.1b). Melody and rhythm encoders use both the original content and the generated embedding to produce the output (figures 7.1c and 7.1d). The two-headed arrow represents a note generating submodule, shown in figure 7.2.

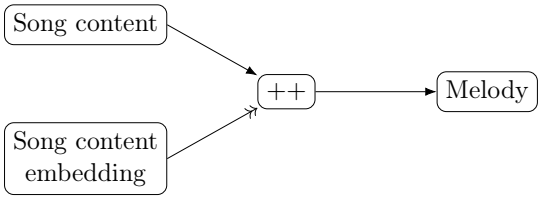
Figure 7.3 shows model architecture in the final two stages.



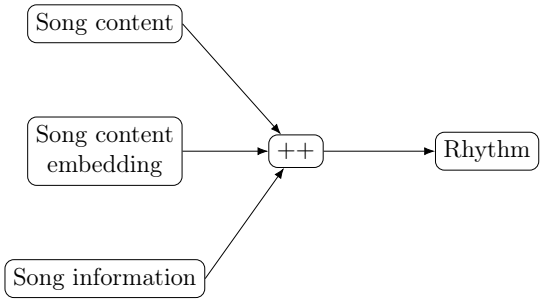
(a) MIDI channels encoder. The convolution is applied along the *note* dimension, independently for every beat. The first LSTM is applied for beats and the second one for bars.



(b) Style encoder.



(c) Melody encoder.



(d) Rhythm encoder.

Figure 7.1: Style extraction stage.

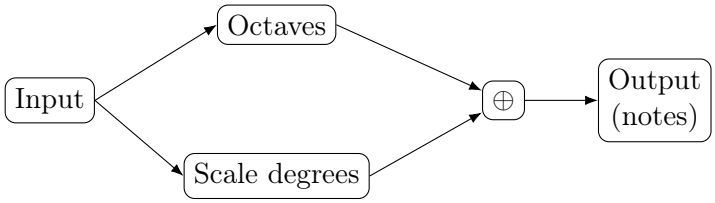
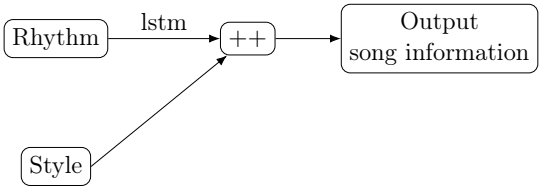
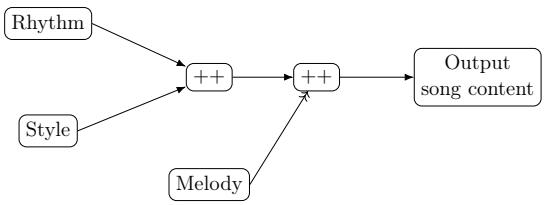


Figure 7.2: A submodule used for generating notes. It generates octave-specific and scale degree-specific information separately, before merging them. Such architecture allows the generated notes to depend on the exact octave, while still preserving correlation between different octaves.



(a) Song information model.



(b) Style applier.

Figure 7.3: Song information prediction (7.3a) and style application (7.3b) stages.

Chapter 8

Experiments

8.1. Loss function

The loss function measures if the song generated by the model is the same as the input song. It is defined as a combination of many components, with the following structure:

- total loss
  - channels loss
    - pitched channels loss
      - notes loss (complement of smooth F1 score)
      - velocity loss (MSE)
      - duration loss (MSE)
      - accidentals loss (cross-entropy)
    - unpitched channels loss
      - notes loss (complement of smooth F1 score)
      - velocity loss (MSE)
      - duration loss (MSE)
  - song information loss
    - instruments loss (cross-entropy)
    - tempo loss (MSE)
    - mode loss (cross-entropy)

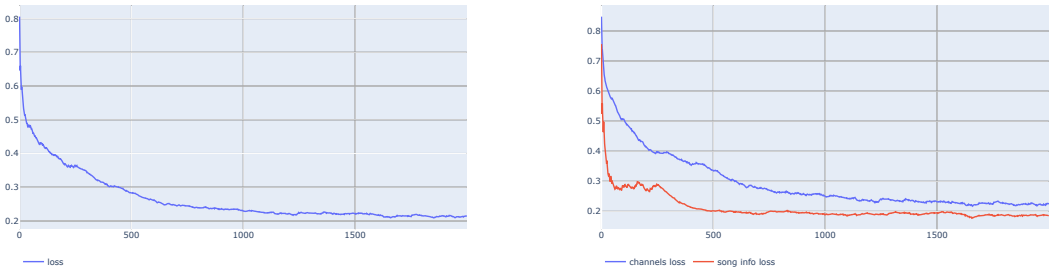
This hierarchy signifies that the total loss is the average of channels loss and song information loss, channels loss is the average of pitched and unpitched channels loss, etc.

Notes loss is measuring if the model is generating the right sounds. Since the desired output is greatly unbalanced (most of all the possible notes will not be played), I'm using a complement of smooth F1 score to measure notes loss:

$$\text{notes loss} = 1 - F_1. \tag{8.1}$$

Smooth F_1 score is definied in a standard way, using

$$\text{true positive} = \min(p, t), \tag{8.2}$$
$$\text{false positive} = \max(0, p - t), \tag{8.3}$$
$$\text{false negative} = \max(0, t - p), \tag{8.4}$$



(a) Total loss.

(b) Loss components.

Figure 8.1: Total loss and its components during training. As can be seen on the second plot, predicting basic song information is easier than generating music, especially at the beginning of training.

where $p, t \in [0, 1]$ are prediction and ground truth respectively, interpreted as fuzzy logical values. In case when p and t can only take discrete values of 0 or 1, this gives a standard definition of the F_1 score. Thus, it is a generalization of the F_1 score compatible with gradient descent. In our case the notes' fuzzy logical values are defined from velocity (1 meaning that the note is played with maximum possible velocity, and 0 meaning that the note is not played at all).

Velocity loss measures if the model is generating notes with the right loudness. Velocity is already measured by the notes loss (acquiring maximum F_1 score requires playing with exactly right velocities), so this loss function provides additional, more explicit component measuring the velocity error. It only takes into account the notes that should be played to avoid problems with unbalanced output (the same is true for duration loss).

Since different kinds of loss functions have different magnitudes, before averaging, all the components are normalized to take values in the $[0, 1]$ interval. Then, the normalized components are averaged using quadratic mean to yield the total loss. The use of quadratic mean instead of arithmetic promotes minimizing the components that have the greatest value, resulting in a more balanced training.

8.2. Training

The model was trained for 2000 iterations using the Adam optimizer [5]. In each iteration the input is a random song from the dataset. Figure 8.1 shows total loss during training, while figures 8.2 and 8.3 show specific components of the loss function. All the plots are smoothed using exponential moving average with momentum parameter equal to .99 and initial bias equal to 1 (maximum possible error).

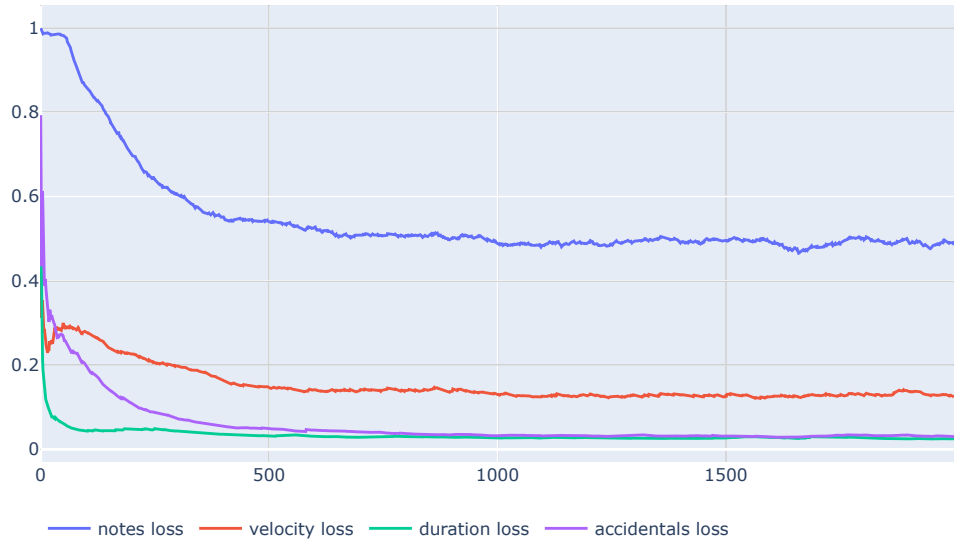
Figure 8.2b shows how pitched and unpitched notes losses compare during trainig. In general percussion is easier for the model, as indicated by lower error value for the most time of training. Interestingly however, approximately during iterations 200-800 the unpitched notes loss is significantly higher (around .7) than the pitched notes loss (around .55). Possible explanaition is that it's easy to achieve this level of unpitched notes loss by mostly repeating the most common drum patterns. This hypothesis can be confirmed by listening to the music generated by the snapshot of the model at iteration 300 (before unpitched notes loss started to decrease again). As it turns out, the generated percussion is indeed very basic, only featuring

some typical drum patters found in most of the pop/rock music. To achieve lower unpitched notes loss the model needed to start encoding some useful information about percussion, but could not do so explicitely per note (like for pitched instruments), which explains why for some time percussion posed a main challenge for the model.

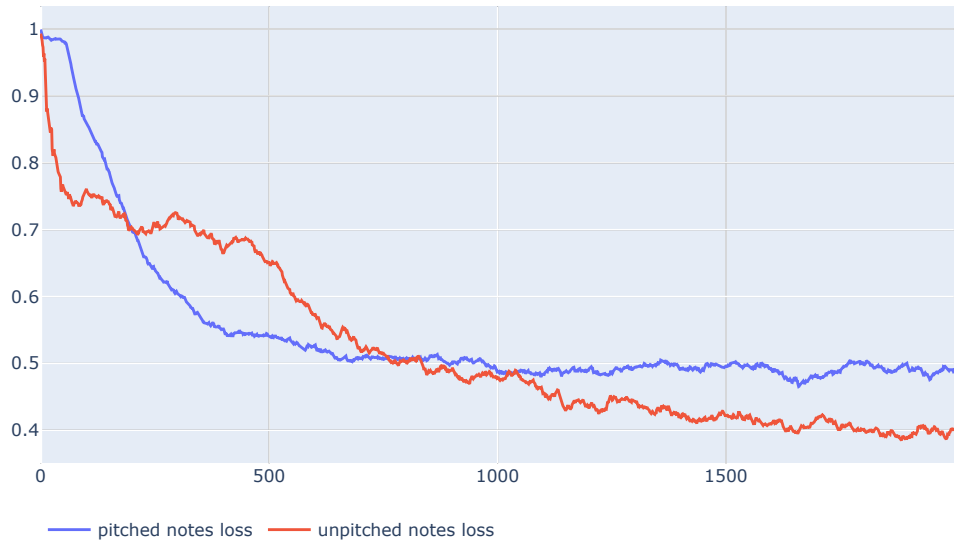
8.3. Style transfer

TODO

After training, the model is used to extract style and composition from two songs and then generate songs from these compositions but with switched styles.



(a) Pitched loss components.



(b) Comparision of pitched and unpitched notes loss. The red line represents how well the model is playing the percussion, while the blue line represents all the other instruments. We can see that in general pitched instruments are harder than percussion – at the beginning the model needs some time to start learning to play pitched instruments, and pitched notes loss eventually saturates at higher level than unpitched notes loss. There are however a few hundred iterations during training when the model is actually better at playing pitched instruments than percussion.

Figure 8.2: Channels loss during training.

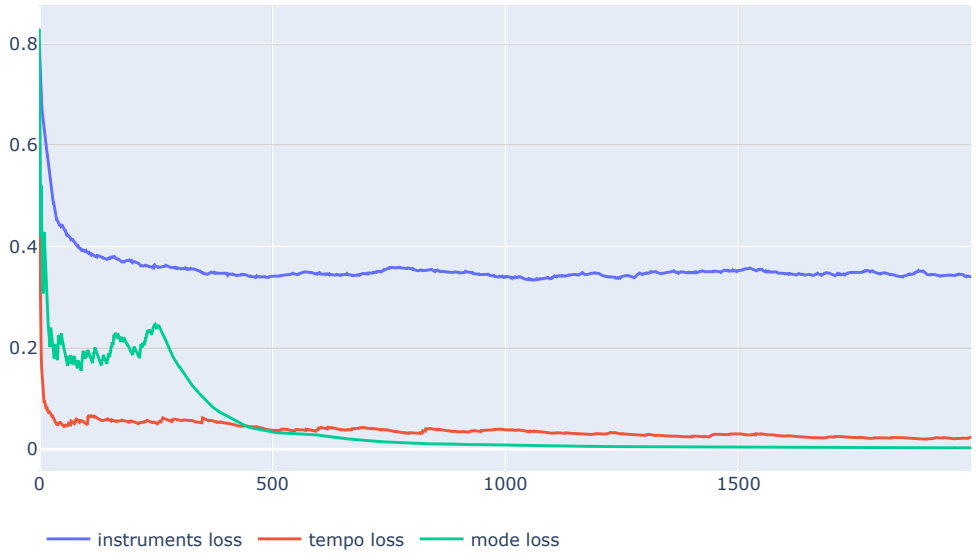


Figure 8.3: Song information loss components during training.

Bibliography

[1] Iman Malik, Carl Henrik Ek, *Neural Translation of Musical Style*, https://arxiv.org/abs/1708.03535.

[2] Gino Brunner, Yuyi Wang, Roger Wattenhofer, Sumu Zhao, *Symbolic Music Genre Transfer with CycleGAN*, https://arxiv.org/abs/1809.07575.

[3] Chien-Yu Lu, Min-Xin Xue, Chia-Che Chang, Che-Rung Lee, Li Su, *Play as You Like: Timbre-enhanced Multi-modal Music Style Transfer*, https://arxiv.org/abs/1811.12214.

[4] Christine Payne, OpenAI Scholars Program, *Clara: Generating Polyphonic and Multi-Instrument Music Using an AWD-LSTM Architecture*, http://www.christinemcleavey.com/files/clara-musical-lstm.pdf.

[5] Diederik P. Kingma, Jimmy Ba, *Adam: A Method for Stochastic Optimization*, https://arxiv.org/abs/1412.6980.