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

The goal of this work is to create a neural network architecture enabling to create a new arrangement of any song in the MIDI format based on the style extracted from another song. I'm using an architecture based on LSTM layers as time-processing units, also utilizing convolutions. I define the notion of musical style and composition and train a model that is able to split and later merge them. This results in a symbolic music style transfer that supports a wide range of musical instruments and does not require the musical styles to be predefined during training, which makes it much less restrictive than many other solutions.

Keywords

style transfer, music generation, midi, neural networks, recurrent neural networks, convolutional 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

List of Figures

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 [1]. 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.

1.1. Goal statement

In this paper the goal is to define a 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.

In summary, I aim for the model that:

- Supports multiple instruments and is able to change them, if necessary

- Supports various time signatures (most importantly $\frac{4}{4}$ and $\frac{3}{4}$)

- Is able to change the tempo and the mode of the song (major or minor)

- Is able to extract the style from any song (in particular the supported styles are not hard-set during training)

Note that other common approach to music generation is to perform it on raw waveform, most typically working with spectrograms and utilizing Fourier Transform, as in [2]. In this work however, I only consider the problem of music style transfer using the MIDI format. Using the MIDI format, as opposed to the raw audio, has two main advantages:

1. MIDI format provides more structured data than raw sound – the input is divided into channels serving as separate instruments, and for each instrument we have the exact information on which notes are played and when. Thus, working with MIDI files I can focus solely on the style transfer, which makes the task much easier.

2. Direct generating of realistically sounding music is in itself a hard problem. Here I'm assuming that the goal is to generate a MIDI file, which can later be converted into audio using any available synthesizer.

1.2. 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 or have some significant limitations.

1.2.1. Neural Translation of Musical Style

In Neural Translation of Musical Style [3] the 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. The main limitations of this solution is that it only supports one instrument (piano) and cannot create a completely different arrangement.

1.2.2. Symbolic Music Genre Transfer with CycleGAN

In Symbolic Music Genre Transfer with CycleGAN [4] the described model can create new arrangements but the styles need to be set during training (in this case it's jazz, classic and pop). Also, similarly as in the previous work, it can still only generate piano, and does not support multiple instruments.

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

In Timbre-enhanced Multi-modal Music Style Transfer [5] the model can operate on different instruments, but here the main limitation is that the styles need to be set during training (the styles are actually instruments; the authors are using guitar, piano and string quartet). Therefore, like in the previous examples, this solution doesn't allow to transfer style between arbitrary songs.

Chapter 2

Overview and results

2.1. 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 a song and combine it with the composition extracted from another song. My hypothesis is that this approach will allow for a sensible music style transfer.

2.2. Results

Using this approach I've been able to achieve interesting results, obtaining a model that can create the new arrangements of given songs, as expected. The exact characteristics of the generated music varies, from very simple changes in used instruments, to generating a song that is almost unrecognizable from the original, but always sounding like an actual music. The quality of the generated music also varies. Sometimes it doesn't sound completely natural, but altogether I consider the results to be satisfying.

Chapter 3

Framework

3.1. 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.

3.2. Data

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

---

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

Chapter 4

Data flow

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

4.1. 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 which notes are played at 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. Therefore, it must combine and encode 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-defined (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.
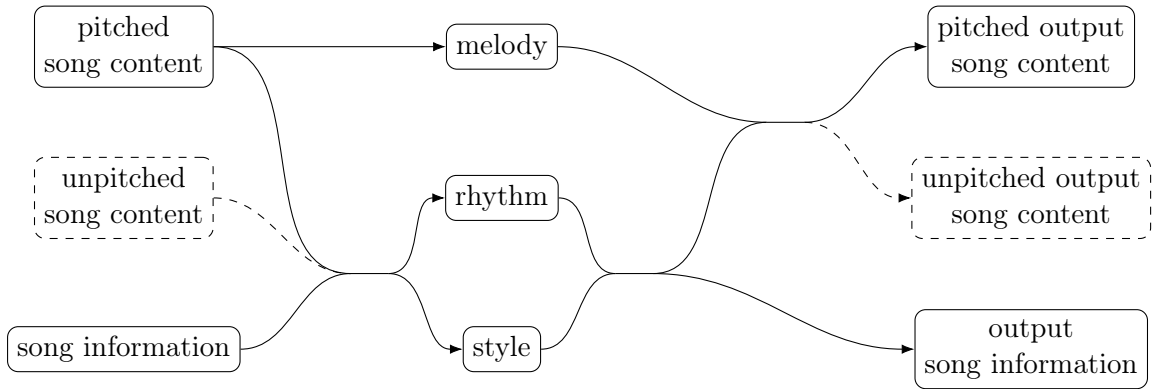
Figure 4.1: Data flow of the model (dashed parts are optional).

4.2. 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.

4.3. 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. The generated song content should coincide with the input from the first stage. In other words, the model must reconstruct the original song. Figure 4.1 depicts the data flow in all three stages.

Chapter 5

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.

5.1. MIDI encoding

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

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

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.

5.1.1. Specifying time

I assume that the time signature cannot change, i.e. the number of the beats in a bar 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, simplifying 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 (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 [6]. 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 the

denominator of $24 = 3 \cdot 8$, 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.

5.1.2. 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.

5.1.3. Note features

The same format is used for both percussion and pitched instruments. However, the number of notes and note features (i.e. 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

---

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

- accidentals (raise or lower the note one semitone)

Accidentals are represented using one-hot encoding with vector of length 3 (raise the note, lower the note, or play the note unmodified), which gives a total of 5 features per note for pitched instruments. For percussion the only note features are velocity and duration, which gives 2 features per note.

5.2. Style and composition representation

Melody is a tensor of shape

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

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

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

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

$$(batch,\ features).$$

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.

Chapter 6

Model

6.1. Notation

In the following diagrams the nodes represent tensors, while the edges represent neural network layers – standard fully-connected and activation layers, unless specified otherwise. Leaky ReLUs are used as activation functions throughout the model, except for the output activations, which are standard sigmoid.

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.

6.2. Model description

Figure 6.1 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 6.1a). 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 then the second one along the *bar* dimension. They produce an embedding of the song content which is then passed to the other submodules.

Style encoder uses an LSTM layer to process the generated embedding into a single time-independent vector (figure 6.1b). Melody and rhythm encoders use both the original content and the generated embedding to produce the output (figures 6.1c and 6.1d). Note that the use of LSTM layers as time-processing units allows for any number of bars and beats per bar in the input song. In particular, any time signature is supported.

Figure 6.3 shows model architecture in the final two stages.

6.3. Melody and rhythm size

The most important hyperparameters are melody and rhythm embedding sizes, which were set to values 8 and 32 respectively. Recall that the number of features per note for pitched instruments is equal to 5, which means that the melody has a slightly bigger size than a single pitched instrument (but needs to encode *all* pitched instruments used in a song). The

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

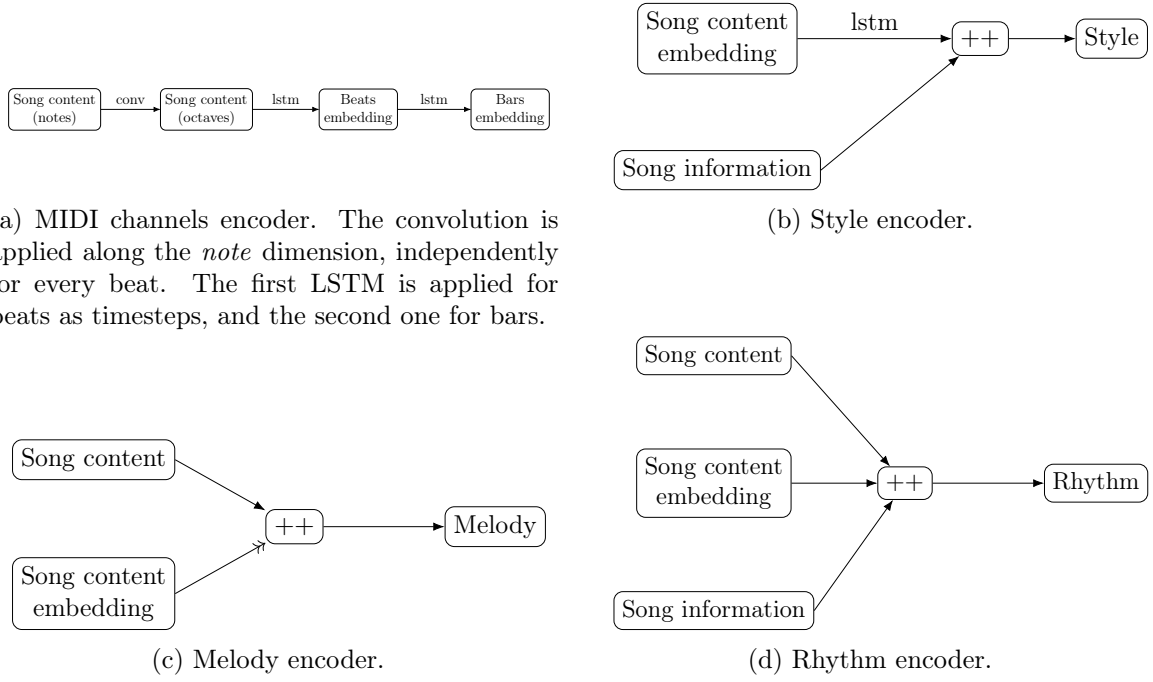(b) Style encoder.

(c) Melody encoder.

(d) Rhythm encoder.

Figure 6.1: Style extraction stage. The two-headed arrow represents a note generating submodule shown in figure 6.2.
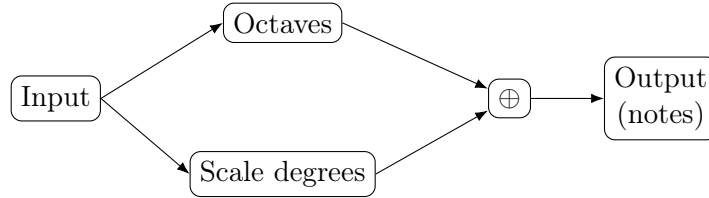

Figure 6.2: A submodule used for generating notes. It generates octave-specific and scale degree-specific information separately, before merging them. Such architecture allows the generated notes to depend on the exact octave, while still preserving correlation between different octaves.

(a) Song information model.
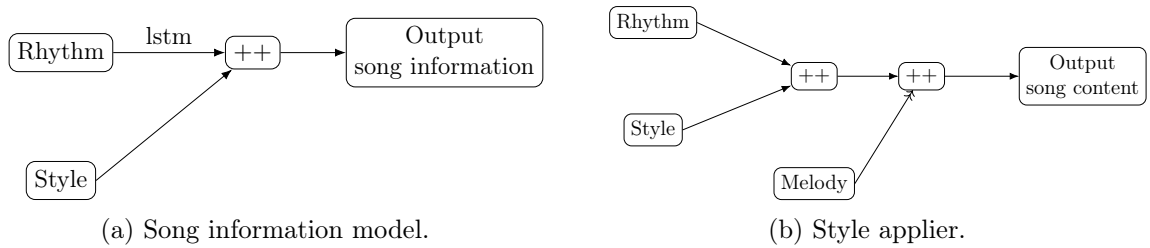
(b) Style applier.

Figure 6.3: Song information prediction (6.3a) and style application (6.3b) stages.

melody contains embedding for each note, as opposed to the rhythm, so the size of the rhythm embedding can be set to a higher value than the melody.

Increasing the size of the melody and/or rhythm embeddings would lead to a better reconstruction of songs but a potentially worse style transfer, since the model could then easily overfit by remembering more features in the composition, effectively ignoring the style vector. Therefore, composition embeddings should be kept rather small for this architecture.

Chapter 7

Experiments

7.1. 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 smooth F1 score to measure notes loss. Smooth F_1 score is defined in a standard way, using

$$\text{true positive} = \min(p, t), \tag{7.1}$$

$$\text{false positive} = \max(0, p - t), \tag{7.2}$$

$$\text{false negative} = \max(0, t - p), \tag{7.3}$$

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

(a) Total loss.   (b) Loss components.

Figure 7.1: Total loss and its components during training.

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 if the note is played with maximum possible velocity, and 0 if the note is not played at all).

The notes loss is then defined as the complement of the smooth F_1 score:

$$\text{notes loss} = 1 - F_1. \tag{7.4}$$

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. To avoid problems with unbalanced output, it only takes into account the notes that should be played (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.

7.2. Training

The model was trained for 5000 iterations using the Adam optimizer [7]. In each iteration the input is a random song from the dataset. Figure 7.1 shows total loss during training, while figures 7.2 and 7.3 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 (i.e. the maximum possible error).

Figure 7.2b shows how pitched and unpitched notes losses compare during training. We can see that the model needs around 100 iterations to start learning pitched instruments (with the pitched loss barely below 1), during which time the unpitched loss is already significantly lower (around .7). This can be explained by the fact that percussion is in general more repetitive than pitched instruments, so it's possible to achieve that level of the unpitched 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 near the beginning of the training (for example at the iteration 300, when the unpitched notes loss shortly flattens at the level of .7). As it turns out, the generated percussion is indeed very basic, featuring only

24

some typical drum patters found in most of the pop/rock music. The fully-trained model can nevertheless generate percussion that is relatively complex.

7.3. Reconstructing the song

The trained neural network can fairly easily reconstruct any song in a way that is immediately recognizable, although often with some differences. It's very common for the model to change the exact instrument that is playing some part, for example changing electric guitar to acoustic, or even piano to guitar, especially if there are many instruments used in the song. These are not a big problems for the style transfer, however, since modifying the original song is our goal.

7.4. Style transfer

The trained model can be used to transfer style between any songs. To do this, the model needs to extract the composition from one song and the style from the other. Then, the resulting composition and style are used to predict song information (i.e. mode, tempo and used instruments) and generate the content of the predicted instruments.

The first, most obvious application is to do the exact same thing as for the image style transfer, i.e. having two songs transfer the style from one to the other. It can be done both ways at once ("switching styles") – generating song A in a style of song B, and song B in a style of song A. In practice however it's much more informative to apply different styles to the same song, to see how they compare.

I used various groups of usually four songs (one for composition and three as style sources) to test the style transfer. The quality of the results varies depending on the exact songs. In some cases the generated music sounds unnatural, with a sudden changes in loudness, but in other instances the results are quite naturally-sounding. For some songs the style transfer is mostly switching tempo and using different instruments for melody, but there are cases when it creates a completely differently sounding arrangement. The differences are most notable when considering the songs of significantly different moods (happy vs. sad) since it this case style transfer often involves switching mode and changing tempo, both of which are immediately recognizable changes.

Another typical scenario is changing the number of used instruments from one to many and vice versa. This includes, for example, creating orchestration of a solo piano piece, or creating a piano arrangement of a given orchestral song. The results are usually not near human-level performance, but are nevertheless sounding decently.
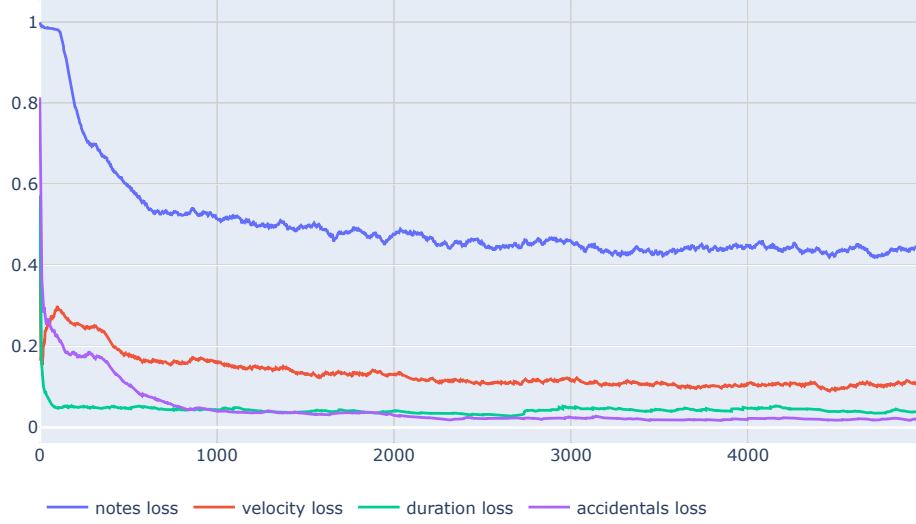
(a) Pitched loss components. The notes loss measures if the model is playing the right notes, while other components measure the specific features of the played notes. The notes loss eventually saturates at around .45 (giving F_1 score of .65).
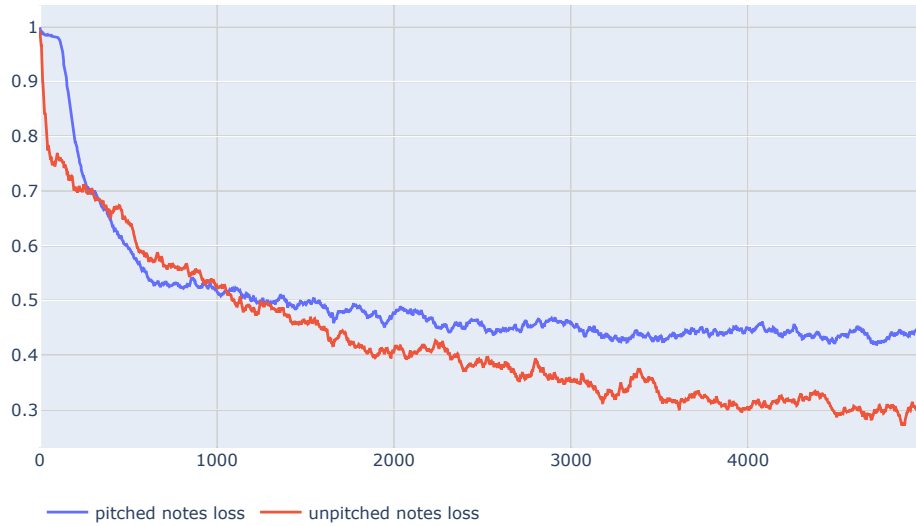
(b) Comparision of pitched and unpitched notes loss. The red line represents how well the model is playing the percussion, while the blue line represents all the other instruments. We can see that in general pitched instruments are harder than drums. There are however a few hundred iterations during training when the model is actually slightly better at playing pitched instruments than percussion.
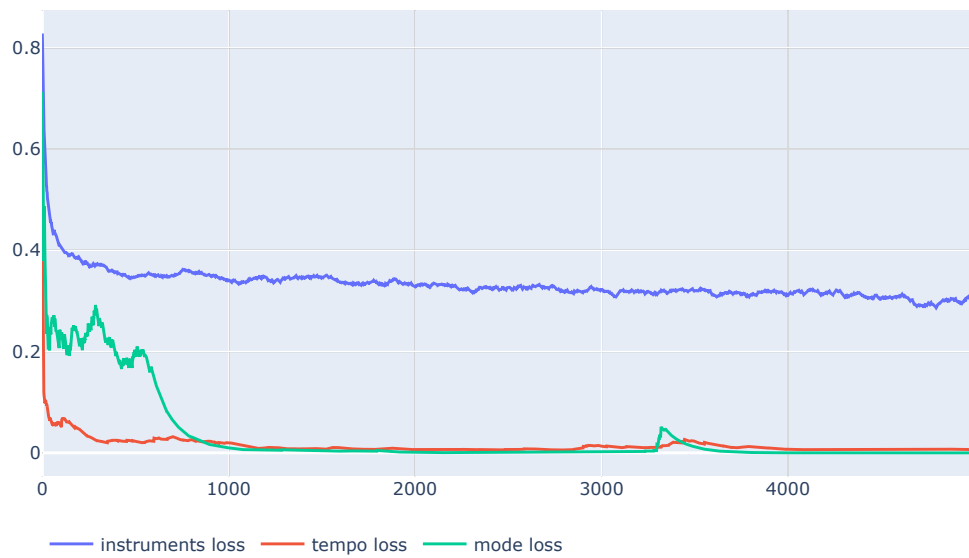
Figure 7.2: Channels loss during training.

Figure 7.3: Song information loss components during training.

Chapter 8

Conclusion

My goal was to create a versatile deep learning model capable of transferring musical style between any two songs in the MIDI format. The model I designed supports various instruments and includes the tempo and the mode of the song as its input and output. It is able to generate a good-sounding music, introducing changes to the arrangement of the given song. In some cases the changes are quite significant, resulting in a notably different arrangement, while in other cases the changes are very basic, like changing the instruments playing the main parts. The model uses fairly basic deep learning solutions, so it can effectively serve as a baseline for further, more advanced architectures.

Bibliography

[1] Leon A. Gatys, Alexander S. Ecker, Matthias Bethge, *Image Style Transfer Using Convolutional Neural Networks*, `https://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/Gatys_Image_Style_Transfer_CVPR_2016_paper.pdf`.

[2] Piotr Kozakowski, Bartosz Michalak, *Music generation using artificial neural networks*.

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

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

[5] 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`.

[6] 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`.

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