



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

*Distributed
Computing*



DJ Roboto

Bachelor Thesis

David Eschbach

`esdavid@student.ethz.ch`

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:

Gino Brunner, Yuyi Wang
Prof. Dr. Roger Wattenhofer

July 23, 2017

Acknowledgements

First, I want to express my gratitude to Gino Brunner and Yuyi Wang for their support during this thesis and for their ever quick and helpful advice when I arrived with questions. I also thank my wife for supporting and encouraging me in times when I did not know how to proceed.

Abstract

In this project we look for a learning model that can reconstruct a given melody using Sonic Pi. In order to achieve this, we first look at instrument classification. We use Long Short-Term Memory (LSTM) networks to build a classifier that is able to decide which instrument was used to generate a given audio file.

Arguably the most important characteristics of a melody are the pitches it contains. Thus in the second part of the thesis we deal with pitch detection. We rely on LSTM networks again to implement a regression which approximates the pitch at a given point of time in an audio file.

In both parts of the thesis we use Sonic Pi to generate the respective train sets consisting of audio files.

Contents

Acknowledgements	i
Abstract	ii
1 Introduction	1
1.1 Motivation	1
1.2 Related Work	2
2 Background	3
2.1 Sonic Pi	3
2.2 Python Interface to Sonic Pi	4
2.2.1 Using OSC Protocol	4
2.2.2 Implementation	4
2.2.3 Sample Creation	5
2.3 Neural Networks	5
2.4 Keras	6
3 Instrument Classification	7
3.1 Motivation and Problem Statement	7
3.2 Implementation	7
3.2.1 One Instrument	8
3.2.2 Multiple Instruments	10
3.2.3 Discussion Of Results	12
4 Melody Recreation	16
4.1 Implementation	16
4.2 Results and Discussion	17

CONTENTS	iv
5 Conclusion and Outlook	19
5.1 Instrument Classification	19
5.2 Melody Recreation	20
Bibliography	21

Introduction

1.1 Motivation

Making music is easy. Everyone can sit in front of a piano and hit some keys. Of course the probability that this approach is going to result in pleasant sounds is not very high. The task of generating *good* music is much harder. Since Sonic Pi[1] came along, every programmer can use his coding skills to generate some music. This enables a programmer to skip the step of learning how to play an instrument. Although the transition from music to good music might happen faster using Sonic Pi, there is still work to do in order to reach a certain level of quality. The technical skill of an instrumentalist is not the only criterion for good music. Thus, the real question is: What *is* good music? Can a computer learn to distinguish between good and bad music?

In this thesis we try to answer this question by considering music reproduction. We look for a computer program that can reconstruct the Sonic Pi code used to produce an audio file while only having access to this audio file. To this end, a program first has to identify the instrument which plays the melody in the sound file, and then it has to reproduce the melody itself.

We start by using Sonic Pi to create a data set of melodies. We then implement an instrument classifier relying on the LSTM architecture, which is a special kind of recurrent neural network (RNN). We use the data set generated by Sonic Pi to train our classifier and then improve the LSTM network to perform better on sound files that were created using real instruments. Furthermore, we adapt our classifier to not only work reliably on melodies played on one instrument but also consider the more difficult case of multiple instruments per audio file.

In the second part of the thesis we use LSTM networks to extract the pitches of a melody. Again, the network is trained on a set of audio files generated by Sonic Pi. We compared different representations of the train data and discovered that the model provides the best results if it is supplied with a representation that combines data from both frequency domain *and* time domain.

1.2 Related Work

Both instrument classification and pitch detection are topics which have been researched before. Especially for pitch detection there are multiple known algorithms. Gerhard gives an overview of the history of pitch detection and lists the most widely used algorithms for both time domain and frequency domain[2].

Machine Learning techniques are used frequently to tackle the problem of pitch detection. For example Özbek et al. examine the use of a likelihood-frequency-time (LiFT) analysis designed for automatic transcription of music using Support Vector Machines (SVM)[3]. They use the information obtained by the LiFT analysis to extract features from the samples before relying on linear, polynomial and radial basis function kernels to extract the pitch. In contrast to us they use classification to solve this task. We use a regression approach.

Also the idea to use neural networks for pitch detection is not new. As early as 1989, Sano et al. used them to model the functionality of the human ear[4]. Their final model focuses on preprocessing the impulse arriving at the ear. As input data they only use a frequency spectrum, while we experiment with data from both time and frequency domain.

Böck et al. use LSTM cells in bidirectional neural networks to perform polyphonic piano note transcription[5]. Similar to us, they use a single regression output layer instead of the more frequently used one-versus-all classification. For the training data they use a format based on a frequency domain representation obtained by using Short-Time Fourier Transforms (STFT) with different window lengths. In contrast to that we use the fast Fourier transform (FFT).

Copeland et al. build on SVMs to approach the problem of instrument classification[6]. They manage to improve their results by not only relying on a pure frequency domain representation but introducing more advanced features like autocorrelation coefficients. In our thesis we use a pure time domain representation to tackle the task of instrument classification.

Graves et al. introduce Bidirectional Long Short-Term Memory (BLSTM) networks and use them to tackle the task of phoneme classification[7]. Additionally, they compare the performance of BLSTMs to the results achieved by other learning models like unidirectional LSTMs, RNNs and multilayer perceptrons (MLP). Discussing the results of these comparisons they state that for their task BLSTMs outperformed almost any other neural network found in literature. Additionally, they point out that LSTMs also beat other neural networks when the training times are compared, that is LSTMs converge faster than other neural networks. As suggested by this paper, we build on LSTMs for all tasks tackled in our thesis.

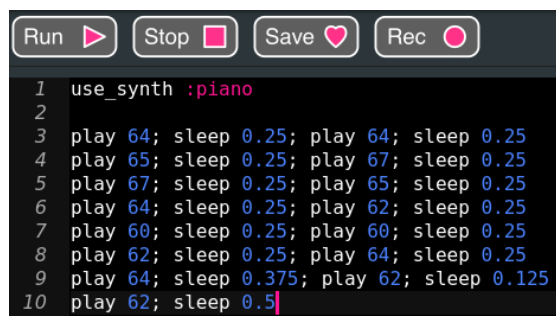
Background

2.1 Sonic Pi

Sonic Pi promotes itself as the “*live coding music synth for everyone*” and as a “*new kind of instrument*”[1]. It is a computer program that allows to code pieces of music and play or record them. Sonic Pi makes it possible that one programmer replaces a whole band or an orchestra.

Originally, Sonic Pi was designed to teach programming[8], because in order to achieve good sounds a user requires knowledge about many fundamental programming mechanisms like loops, modularization or threading. Sonic Pi is also very suitable for live performances, since sounds can be changed easily while they are playing.

Sonic Pi comes with a wide range of built in synthesizers, instruments and samples to be used for creating music. It offers a variety of commands to produce sounds and combine them. The possibilities range from replaying and combining recorded samples to providing melodies and tell Sonic Pi on which synthesizer to play them. Here we provide just one example of how to use Sonic Pi. The code shown in Figure 2.1 prompts Sonic Pi to produce the most famous tunes of Beethoven Symphony No. 9 as soon as the run button is hit.



```
Run ▶ Stop ■ Save ♥ Rec ●  
1 use_synth :piano  
2  
3 play 64; sleep 0.25; play 64; sleep 0.25  
4 play 65; sleep 0.25; play 67; sleep 0.25  
5 play 67; sleep 0.25; play 65; sleep 0.25  
6 play 64; sleep 0.25; play 62; sleep 0.25  
7 play 60; sleep 0.25; play 60; sleep 0.25  
8 play 62; sleep 0.25; play 64; sleep 0.25  
9 play 64; sleep 0.375; play 62; sleep 0.125  
10 play 62; sleep 0.5
```

Figure 2.1: Melody in Sonic Pi

Sonic Pi uses integers to address pitches. For example the command *play 64* prompts Sonic Pi to play the 64th tune of the piano. The time to sleep between two pitches is given in seconds.

Implementation

Behind the graphical user interface (GUI) of Sonic Pi there is a backend that runs the server responsible for executing the code[9]. This backend communicates to the GUI via the message-based network protocol Open Sound Control (OSC), which was designed for communication between multimedia devices. Some of the benefits of OSC are “interoperability, accuracy, flexibility and enhanced organization and documentation”[10].

2.2 Python Interface to Sonic Pi

2.2.1 Using OSC Protocol

At the beginning of the thesis we decided to use Python for the programming parts, since there are several powerful frameworks and libraries to deal with Machine Learning Problems available in Python. The first task is to control Sonic Pi from a Python program by developing a python wrapper. Here we can use the fact that Sonic Pi internally uses the OSC Protocol to communicate between GUI and server. Sonic Pi specifies a port on which the server listens for incoming messages from the GUI. Thus, our wrapper can just send messages to this port. While the application program interface (API) of this communication is not officially documented because it is subject to change, the author of Sonic Pi provides an informal description[11]. To play a melody we can now send an OSC message containing the corresponding code from our wrapper to the specified port. In this message we need to state */run-code* as target and Sonic Pi will play the melody just as if we hit the run button in the GUI.

2.2.2 Implementation

The most important part of the wrapper is the class `SonicPi`. It handles the setup of the connection to the Sonic Pi server in its constructor. The class provides several methods to add code to a string called `RUN_ARG` which contains the code to be sent to */run-code*. The method `commit-run` is the equivalent of hitting the run button in the GUI. Furthermore, the class includes functionality to save a played sound into a wave file.

Adding Instruments

One very useful property of Sonic Pi is that the user can specify the rate at which a sample is played. We use this to include instruments which are not provided by Sonic Pi in a similar fashion as in [12]. Virtual Playing Orchestra[13] provides several sets of instrument samples which contain just one pitch of an instrument each. Changing the rate at which such a sample is played changes the pitch. Since the frequency difference between two pitches that are next to each other is approximately constant, we can use one sample to produce several pitches. For example if we have a sample which contains the pitch a2 we can just play it at rate 1.059462323 ($\approx \sqrt[12]{2}$) to get a#2.

Because we want to be able to use the same functions for built-in and sampled instruments, we adopt the convention from Sonic Pi to address pitches using integers. For sampled instruments this is realized by introducing one dictionary per instrument. That dictionary uses the desired pitch as key and a tuple whose entries are the sample name and the rate as value.

2.2.3 Sample Creation

For the Machine Learning tasks we need a set of music samples played on different instruments. To get this we write a python program that creates such a set using the wrapper just described. Since we want to cover a range of samples as big as possible, we create those samples almost completely randomly. But since real music is not random either, we have to specify some rules. The algorithm has to stick to a certain scale and to a pitch range which is defined per instrument. Also, the samples have a minimal and a maximal length. Besides that, the program can choose many parameters randomly including the chords, how many pitches to play per chord and how long a chord is played. The samples are stored in wave files[14].

One problem we faced in the task of creating this sample set is that the Sonic Pi functionality of saving records is the digital analogue of holding a microphone next to an instrument. That means Sonic Pi records only in real time what it sends to the speaker and thus it takes one hour to produce one hour of music.

2.3 Neural Networks

The basic building blocks of a neural network are called neurons. This name was chosen because they are inspired by biological neurons also known as nerve cells[15]. Neurons are simple computational units that take inputs and produce outputs.

In a neural network neurons are combined into several layers, which can have an arbitrary number of neurons. Every neuron of layer $i - 1$ is connected with each neuron of layer i . These connections are given by weights which are multiplied with the outputs of the neurons on layer $i - 1$ to produce the inputs for the neurons on layer i . Training a neural network model is about finding the optimal values for these weights. A neural networks with n layers implements a function $F(x)$, where x is the input of the first layer, x_i the output of layer i and $F(x)$ the output of the last layer. In this basic case this can also be expressed as:

$$x \equiv x_0 \rightarrow x_1 \rightarrow \dots \rightarrow x_n \equiv F(x) \quad (2.1)$$

Such neural networks are also known as feedforward neural networks or MLPs.

Apart from feedforward neural networks there are also RNNs. In contrast to feedforward neural networks, RNNs allow data to flow forward *and* backward.

The problem of conventional RNNs is that the errors tend to either “blow up or vanish while flowing backward” [16]. To solve this problem Hochreiter et al. introduced the LSTM architecture [16]. LSTM networks are well suited for processing sequences with high correlation between subsequent elements. The audio files we are working with are wave files and store the audio signal simply as a sampled wave [14]. Thus we deal with sequences of subsequently correlated elements. For this reason we chose to rely on LSTM networks for this thesis.

2.4 Keras

Keras [17] is an API written in Python that was specially designed for working with neural networks and thus offers mechanisms to build them very fast. A user can specify the parameters of the network he wants and Keras builds that network using libraries like TensorFlow [18] or Theano [19]. Keras is user-friendly as it minimizes the time to get from a network sketched on a piece of paper to a working model. It is also easy to connect several models and to add new modules.

Instrument Classification

3.1 Motivation and Problem Statement

Humans can learn to distinguish the sounds of different instruments quite fast. A couple of examples are enough for most people to learn to keep piano and flute melodies apart. Thus, there must exist some attributes according to which humans can decide that. It should be possible for a computer to find such attributes and acquire the same capability of classifying different melodies according to the instrument which plays them.

The goal of this classification task is to find a model that learns a representation of a melody according to which it can decide what instrument plays the respective melody. We start by assuming that a melody consists of pitches of exactly one instrument. Later we relax this requirement and look for a model that can predict all instruments encountered in a audio file. Our sound files can contain pitches of the following set of instruments: piano, trumpet, clarinet, flute and viola. Piano is the only instrument of this set which is provided by Sonic Pi as built-in synthesizer. We added the other instruments using the method described in [2.2.2](#).

3.2 Implementation

To reuse as much code as possible we first implement a class `InstrumentsClassifierBasic` from which all other versions inherit. This class contains a collection of all learning models used, and functionality to choose, train and test a model. It further provides the methods used to document the performance of the models according to the metrics specified for the particular version of the classifier. We decided to use this structure because we want to allow that different versions can construct the train and test set in different ways. All the same, we want to have the models gathered at one place.

In spite of allowing different methods to construct the train and test set, all of them have to follow some guidelines. A user can specify the length of

the audio-snippets in the train and test set by setting the parameter *scope_size*. Additionally, the user can set the downsample factor as well as the batch size and the number of epochs used to train the model. Further, a user can specify which and how many instruments are used in his training set.

For the documentation of results we used a method that writes all parameters used to a CSV file. This includes the name of the model, the number of epochs, the size of the snippets and many more. Most importantly, it includes the train and test scores according to the metrics used in the respective version.

3.2.1 One Instrument

The first requirement in order to solve the first task is to have a train and a test set which only contain melodies played on exactly one instrument. To generate these we adapted our snippet producing routine presented in 2.2.3. In order to ensure that the model decides according to the instrument used, we build a part of the train set by defining melodies and generating one snippet per instrument for each of these melodies.

While generating the sets, we chose to include the names of all instruments used to generate a wave file into the filename. This enables us to use the filename to build our target matrix while reading the snippets. We define the target matrix M as follows:

$$M_{i,j} = \begin{cases} 1 & \text{if instrument } j \text{ appears in snippet } i \\ 0 & \text{otherwise} \end{cases} \quad (3.1)$$

Since we are working with only one instrument per snippet and there are no empty snippets, every row of M contains exactly one non-zero entry.

The first model we use for this task contains two LSTM Layers consisting of 64 LSTM cells each and a Dense Layer on top, which uses softmax[20] as activation function to perform the classification. Softmax is a function that operates on a vector z of length n as follows:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^n e^{z_k}} \quad (3.2)$$

Softmax computes a probability distribution, thus in our case every entry of the prediction vector represents the probability that the corresponding snippet was generated by the respective instrument. Classification can be achieved by taking the maximum entry of this vector.

We use cross entropy[21] as loss function. Cross entropy is a function computed upon two probability distributions. For two discrete probability distribu-

tions p and q it is defined as follows:

$$H(p, q) = - \sum_x p(x) \log q(x) \quad (3.3)$$

We train the model using the optimizer rmsprop[22]. Rmsprop is a gradient-based optimizer which normalizes the gradients by considering the magnitude of recent gradients[22].

To evaluate the performance of the model, we compute the classification accuracy which is defined as the fraction of the number of correct predictions over the total number of predictions. We train this model on a set that contains 1000 snippets and test it on a set which consists of 150 snippets. We obtain the following results:

	Train Set	Test Set
Accuracy	0.99	1.0

As expected, our model works well on the data produced by Sonic Pi. Now we also want to know how good our model performs on real music. Keras offers the possibility to save a trained model to use it again in another context. To build our real music test set consisting of 1884 snippets we take songs from YouTube[23], which are played only on the instruments we used and slice them into pieces. For this real music test set our saved model yielded a classification accuracy of 0.17. Since we use five instruments this result is as good as we would expect a random classifier to perform.

We conclude that our model is not able to generalize to real music data. Therefore we need to try other techniques. We start by using an autoencoder. An autoencoder is a model that consists of an encoder that transforms the original input array into an intermediate representation and a decoder that reconstructs the original from the intermediate representation. The goal of an autoencoder is to reconstruct the input. After training the autoencoder we extract the encoder and use it to transform the input data to the intermediate representation. This intermediate representation is used as input for the classifier. The idea behind this approach is to abstract away the unnecessary information which seems to separate music data produced by Sonic Pi and real music files.

Additionally to using an autoencoder, we increase the size of the train set to 70,221 snippets. Running the whole program, which includes training the autoencoder and training the classifier, yields the following only slightly improved results:

	Train Set	Test Set	Real Music Test Set
Accuracy	0.95	0.95	0.23

The problem about this approach seems to be that a simple autoencoder is not task-sensitive. The autoencoder does not know what we consider a good intermediate representation. We would favour an intermediate representation which amplifies the differences between distinct instruments. Thus, we need to couple the tasks of reconstruction and classification together by optimizing the two objectives of reconstruction and classification in parallel. To achieve this, we use the fact that layers can be shared by multiple models in Keras.

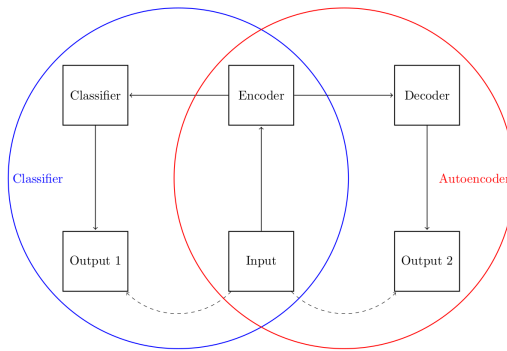


Figure 3.1: Final Classifier Model

Our final model consists of an autoencoder and a classifier which share the layers of the encoder as shown in Figure 3.1. Training the autoencoder and the classifier alternately ensures that we get good results for both reconstruction and classification. Running this final model yields the following results:

	Train Set	Test Set	Real Music Test Set
Accuracy	0.86	0.84	0.35

3.2.2 Multiple Instruments

The second task we tackle is to predict all instruments present in an audio file that contains sounds of multiple instruments. As in the previous task, the first step is to produce suitable train and test sets by adapting the method presented in 2.2.3. To generate snippets we use all combinations of instruments, and we determine that all instruments present in a snippet have to play at the same time. This way we ensure that it is always possible to extract all instruments contained in a snippet, no matter which scope is chosen.

In contrast to the previous task, we have to adapt the format of the target matrix and the way a prediction is taken. The target vector of a snippet in the train set is still a binary vector containing ones at the column indices of instruments that were used to generate a snippet. But since now there might be multiple ones in this vector, it is no probability distribution anymore. We have

to fix that in order to still be able to use cross entropy as loss function. Thus, we need to normalize all rows of the target matrix before training.

We use a model that consists of two LSTM layers with 64 cells per layer and one Dense layer on top, which performs the classification step. In contrast to the first task we have to change the activation function of this Dense layer from softmax to sigmoid[24] because softmax provides a probability distribution. This is not favourable anymore, since we want multiple entries of our prediction to be high in the cases where multiple instruments are present. The sigmoid function is given by:

$$S(x) = \frac{1}{1 + e^{-x}} \quad (3.4)$$

We first consider the simplified subtask to perform classification on a set where each snippet contains tunes of exactly two instruments. The idea behind this is to see how much the accuracy decreases as soon as we release this restriction. This indicates how much the difficulty of the task increases. To document the performance we introduce three new measures:

1. We define *exact matches score* to be the percentage of snippets for which prediction and target are identical.
2. We define *partial matches score* to be the percentage of snippets for which the prediction and the target overlap.
3. As third measure we use the average hamming distance[25]. The hamming distance of two vectors corresponds to the number of entries in which the two vectors are different. This measure is more expressive than the partial matches score. Optimizing partial matches score can be achieved by always predicting all instruments, whereas for the average hamming distance there is no such cheat.

For the more restricted problem of two instruments per snippet we train the model on 3998 snippets and test on 1309 snippets. Training the model for 50 epochs results in the following classification accuracies:

	Train Set	Test set
Exact matches score	0.95	0.92
Partial matches score	1.0	0.99
Average hamming distance	0.06	0.13

As expected, the results get worse when we relax the restriction and allow an arbitrary number of instruments per snippet. We increase the size of the model to 128 LSTM cells per layer, train it on 2983 and test it on 166 snippets. This process yields the following classification accuracies:

	Train Set	Test set
Exact matches score	0.47	0.23
Partial matches score	1.0	1.0
Average hamming distance	0.75	1.26

To do better, we further increase the size of the model to 256 LSTM cells per layer. While this step results in an increased train accuracy, the performance on the test set is almost unchanged:

	Train Set	Test set
Exact matches score	0.73	0.25
Partial matches score	1.0	1.0
Average hamming distance	0.3	1.21

3.2.3 Discussion Of Results

One Instrument

Although in the final version we manage to double the classification accuracy on the real music data set compared to our baseline model, the results are disappointing. They suggest that the data sets produced by Sonic Pi are not general enough to represent real music data and that we overfit to the Sonic Pi sound representation. To establish this we use the real music data set as validation set in training the classifier on the Sonic Pi data and compare the training loss and the validation loss. After the first epoch training loss and validation loss are similar but as the training loss decreases, the validation loss increases as shown in Figure 3.2.

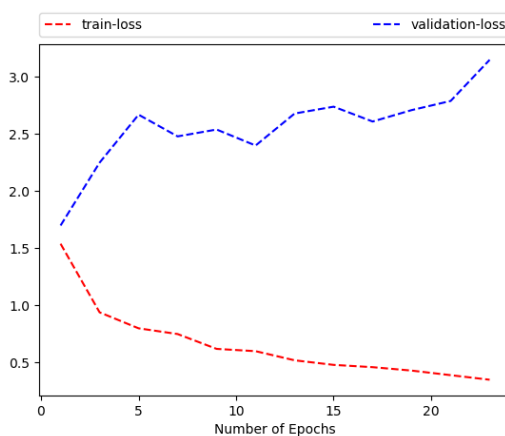


Figure 3.2: Development of training and validation loss

In order to check the sanity of our real music data set, we split it into a train set and a test set consisting of 1,663 and 221 snippets respectively. We train and test our final model of the classifier on these newly created sets. Additionally, we create another train set which contains the snippets from the real music train set and 500 snippets generated by Sonic Pi. After training on this one, we test the model on the same real music test set we use for the real music train set. These two models result in the following accuracy scores:

	Train Accuracy	Test Accuracy
Using real music train set	0.74	0.44
Using combined train set	0.65	0.52

These results indicate a potential overfit to the train set. This does not come surprisingly given the size of the sets and considering that the real music data contains noise in contrast to the Sonic Pi data. We further observe that the train accuracy is roughly ten percent higher when we train on the Sonic Pi train set compared to training on the real music train set. This difference supports our assumption that the good results for the Sonic Pi data presented in 3.2.1 arise from an overfit to the Sonic Pi sound representation. It also points out that classifying the real music data is a harder task than classifying the Sonic Pi snippets.

The occurrence of the overfit to the Sonic Pi sound representation could be an indicator that our snippet producing routine introduced in 2.2.3 is not randomized enough. Thus, we would like to know how many different snippets our routine can produce. Based on this we would like to estimate how probable it is that our train set contains many duplicates.

For the final model we considered snippets with a length of 0.66 seconds. The snippets were extracted from the audio files such that they contain the maximal amplitude value occurring in the respective audio file. The number of melodies of this length that our routine can produce is given by the following formula:

$$\sum_{i=2}^7 (f \cdot r_i \cdot c_i) = \sum_{i=2}^7 (f \cdot r_i \cdot 78^i) = 8.39 \cdot 10^{15} \quad (3.5)$$

f corresponds to a constant factor, which is defined by the number of possible combinations of instrument, scale and range for a particular melody. r_i denotes the number of different rhythm patterns of length i that can be built. c_i corresponds to the number of possibilities to occupy a given rhythm pattern of length i with pitches or chords. The bounds of the sum result from the fact that rhythm patterns consist of sleep durations between 0.1 and 0.4 seconds. These sleep durations are provided as arguments to the Sonic Pi command *sleep* as explained in 2.1.

Although it is possible that we have duplicates in our train set, we conclude that it is unlikely that we have many of them. But since the set of melodies

that could be generated by our routine is roughly 10^{10} times bigger than the set we actually produced, it is possible that we could improve the performance on our real music data set by further increasing our train set. To estimate whether this would be worthwhile, we run the final model again on smaller data sets and compare the results:

Size of Train Set	Train Set	Test Set	Real Music Test Set
500	0.98	0.96	0.31
1982	0.94	0.89	0.34
70221	0.86	0.84	0.35

As the results almost not improve while we add more snippets to the data set, we stop generating more snippets.

We conclude that there is a general difference between wave files generated by Sonic Pi and our real music wave files. To establish this we look at the mean[26] and the standard deviation[27] of our sound files. We first want to know if it is possible to distinguish sound files generated by Sonic Pi and real music sound files only considering their mean and standard deviation. In order to figure this out, we implement a simple classifier using one LSTM layer consisting of 16 cells and one Dense layer with softmax as activation function. We generate a train set consisting of 700 Sonic Pi and 1,663 real music wave files and a test set which contains 490 Sonic Pi and 221 real music snippets. Each wave file is transformed to a vector that only contains the mean and the standard deviation of this wave file before it is provided to the classifier. Training this classifier results in the following accuracies:

	Train Set	Test Set
Accuracy	0.76	1.0

Training for one epoch is enough to achieve a perfect test score. Thus, deciding if a wave file was generated by Sonic Pi or is part of our real music data set is possible, even if we only consider its mean and standard deviation. This finding confirms our assumption that our real music wave files and our Sonic Pi wave files are different.

Additionally, we would like to know if it is possible to classify the snippets according to their instrument only based on their mean and standard deviation. For this task we use a classifier consisting of two LSTM layers with 128 cells per layer and a single Dense layer with softmax as activation. We use the same data as in the task of distinguishing between Sonic Pi and real music wave files and get the following accuracies:

	Train Set	Sonic Pi Test Set	Real Music Test Set	Combined Test Set
Sonic Pi Train Set	0.45	0.44	0.02	0.31
Real Music Train Set	0.48	0.19	0.55	0.30
Combined Train Set	0.40	0.19	0.52	0.29

Training on the set that includes both Sonic Pi and real music snippets yields a much better test score on the real music test set than on the Sonic Pi test set. This gap indicates that the distribution of sounds generated by our snippet producing routine (2.2.3) using a particular instrument in Sonic Pi does not match the distribution of sounds produced by the corresponding real instrument.

Multiple Instruments

While we increase the complexity of our problem, we observe that the performance of our classifier steadily decreases although we increase the size of the model. In the following table we compare the accuracies of the different models on the respective Sonic Pi test set according to the metrics exact matches score (ExactMS), partial matches score (PartMS) and average hamming distance (AHD) defined in 3.2.2

	ExactMS	PartMS	AHD
One instrument	0.99	0.99	0.02
Two instruments	0.92	0.99	0.13
Five instruments	0.25	1.0	1.21

This decline of performance is no surprise. In 3.1 we state that humans can learn to reliably classify instruments according to their sound characteristics quickly. But the sound of an instrument is harder to identify if it does not come isolated. For five available instruments the task of assigning a sound file to the set of instruments used to generate it is more challenging for humans than recognizing a single instrument. Thus, we expect that the task is also more difficult for a computer and the results confirm this assumption.

To establish that the task of classification gets harder for humans as soon as more instruments are involved we perform an experiment. We try how reliably we can classify the different sounds with our own ears. For every task we take 50 snippets, listen to the sound once and try to assign it to the correct set of instruments. This results in the following classification accuracies:

	ExactMS	PartMS	AHD
One Instrument	1.0	1.0	0.0
Two Instruments	0.91	1.0	0.18
Five Instruments	0.45	1.0	0.79

Melody Recreation

The sequence of pitches is one of the most characteristic features of a sound as a melody can be made completely unrecognizable by changing only some of its pitches. Therefore, one step on the journey to reproducing a music file is to decide which pitch is played by an instrument at a certain point in time. We extract the pitches using a regression model.

To keep it simple we consider only snippets consisting of exactly one pitch. The goal is to extract the pitch only considering a small time interval in the wave file. If this is possible, we can recreate a melody by slicing it into short fractions and predicting the pitch independently for each of these fractions.

4.1 Implementation

Already during the development we equipped our wrapper with the functionality to save the code that is sent to the Sonic Pi server to a text file. In order to extract the melody from a text file saved like this, we wrote a parser that reads two vectors. The first vector contains all pitches of a melody in time order and the second vector consists of the lengths of those pitches. In our case of one pitch per snippet, this process simplifies to extracting a single pitch and its length from the text file.

The train and test set are created with a method similar to the one already used for Instrument Classification. The only difference is that we can split up the snippets further. With our five instruments we can produce exactly 227 distinct tunes in Sonic Pi, which equals the number of distinct programs we can use to generate snippets containing only one pitch. Since Sonic Pi produces exactly the same audio file for the same code, there is no point in producing many snippets. The only possible difference is a phase shift caused by different delays in the communication over the OSC Protocol. Therefore, splitting up a snippet into smaller parts achieves the same result as producing more snippets. The target of the model is a vector of pitches in their integer representation as shown in 2.1.

The model we use to solve this task is a three layer LSTM network which uses mean absolute error (MAE)[28] as loss function and sigmoid as activation. The MAE of two vectors x and y of length n is given by:

$$\text{MAE}(x, y) = \frac{1}{n} \left(\sum_{i=1}^n |x_i - y_i| \right) \quad (4.1)$$

Because sigmoid yields a result in the open interval $(0,1)$, we have to ensure that the target values are in this interval too. Since the integer representations of the pitches used range from ten to 99, this can be achieved by dividing the target by 100. Later we have to multiply the predicted value for a snippet by 100 again and round that result to an integer to get the final predicted pitch.

An important question is in which form we should provide the data to the learning model. Wave files contain the sampled wave of a sound, that is they store the information represented in time domain. But since the pitch of a sound is related to the *frequency* of the sound wave, we might be more interested in a frequency domain representation. To obtain such a representation we can apply FFT on the data set. We try three different formats for the data in the train set. The first contains the original sampled wave in the time domain, for the second we use the FFT of the sound wave and as a last version we provided a combination of both. To obtain that combination we use the property that the FFT function of numpy[29] yields a result array of the same size as the input array. Thus, we can use a matrix with the time domain representation in the first row and the frequency domain representation in the second row to describe a snippet.

4.2 Results and Discussion

To measure the performance of the models we use the percentage of pitches for which the distance between prediction and target is less than one octave. The train set contains 21,777 and the test set 5,449 snippets. Each snippet has a length of 2,000 time steps, which is enough to include at least one whole period of all sound waves that can be produced.

Training the model separately with the three different input formats yields the following results, measured with the metric just described:

	Train Set	Test set
Time domain	0.85	0.84
Frequency domain	0.84	0.84
Combination	0.90	0.88

These results suggest that time domain representation and frequency domain representation are equally suitable to extract the pitch. This does not come

surprisingly since there are multiple known pitch detection algorithms for both time domain and frequency domain[2]. Apart from that, the results indicate that we get most information about the pitch if we consider both time and frequency domain representation of the audio files.

In 5.2 we sketch how we would try to increase the performance of our model, if we had more time to work on the topic.

Conclusion and Outlook

While we have been able to reliably solve the task of instrument classification for the audio files created using Sonic Pi, we failed to achieve good results on the real music data. For the task of pitch detection we only achieved partial success. We were able to approximate the correct pitch, but we failed to confidently predict it. In this chapter we provide an outlook on what could be done to tackle these problems.

5.1 Instrument Classification

We located the problem responsible for the bad results on the real music data in the insufficient generality of the train set we generated with Sonic Pi. Therefore the first step on the way to better results would be to obtain a more general train set.

As shown in 3.2.3 our snippet generating method introduced in 2.2.3 is able to produce a high number of varying melodies. All the same, we could still increase this number by introducing even more randomization into the method. For example we could put our focus on the dynamics of the melodies. This could be achieved by considering more of the parameters that Sonic Pi offers. For instance Sonic Pi offers thirteen parameters for the command *play*. Most of them help to control the dynamics of the melodies. We only used four of these parameters and adopted the default values for the rest. Introducing more randomization at this stage could result in a better generalization to real music data.

Based on this new set we could then try some techniques of transfer learning in order to overcome the performance decline on the real music data. Transfer learning deals with transferring knowledge from a related task which has already been learned[30].

Additionally, we could try other formats for the train data. For example in [6] a couple of interesting methods of feature extraction are used to obtain the format for the data to train the classifier on.

5.2 Melody Recreation

Our model for pitch detection can only be used reliably for approximations of the pitch. Böck et al. achieved better results for the more challenging polyphonic case[5]. They use an accuracy-metric, which was defined by Dixon[31]:

$$Score = \frac{N}{FP + FN + N} \quad (5.1)$$

N denotes the number of correctly identified notes, FP stands for the number of notes reported by the system that were not played and FN counts the number of notes played that were not reported by the system. Evaluating their model relying on this metric they present results of up to 90 percent. Since their setting differs from ours, we can not directly apply this to our case. But it indicates that there are probably ways in which we could improve our model.

We could proceed by following an approach similar to the one we finally used for the instrument classification. A network that combines autoencoding and the task of pitch detection by alternately training these models could improve the results.

Another possibility would be to try additional formats for the training data. For example we could try some preprocessing techniques like the use of an autocorrelation functions as it is done in some known frequency domain based algorithms for pitch detection [2].

We could also try how a bidirectional LSTM network performs. Böck et al. used this approach with great success[5].

Bibliography

- [1] : Sonic Pi - The Live Coding Music Synth For Everyone. <http://sonic-pi.net/> Accessed: 2017-06-29.
- [2] Gerhard, D.: Pitch extraction and fundamental frequency: History and current techniques (2003)
- [3] Özbek, M.E., Delpha, C., Duhamel, P.: Musical Note And Instrument Classification With Likelihood-Frequency-Time Analysis And Support Vector Machines. 15th European Signal Processing Conference (EUSIPCO 2007), Poznan, Poland, September 3-7, 2007, copyright by EURASIP
- [4] Sano, H., Jenkins, B.K.: Long short-term memory. *Computer Music Journal* **13**(3) (1989) 41–48
- [5] Böck, S., Schedl, M.: Polyphonic piano note transcription with recurrent neural networks. In: Acoustics, speech and signal processing (ICASSP), 2012 IEEE International Conference on, IEEE (2012) 121–124
- [6] Copeland, C.N., Mehrotra, S.: Musical Instrument Modeling and Classification. CS 229: Machine Learning, Stanford University (2013)
- [7] Graves, A., Schmidhuber, J.: Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks* **18**(5) (2005) 602–610
- [8] : Raspberry Pi Learning Resources - Sonic Pi Lessons. <https://www.raspberrypi.org/learning/sonic-pi-lessons/> Accessed: 2017-07-03.
- [9] : Sonic Pi Internals. <https://github.com/samaaron/sonic-pi/wiki/Sonic-Pi-Internals> Accessed: 2017-07-21.
- [10] : Open Sound Control. <http://opensoundcontrol.org/introduction-osc> Accessed: 2017-07-03.
- [11] : Sonic Pi Internals - GUI Ruby API. <https://github.com/samaaron/sonic-pi/wiki/Sonic-Pi-Internals----GUI-Ruby-API> Accessed: 2017-07-03.
- [12] : rbnrpi - Adding a new sample based flute voice for Sonic Pi. <https://rbnrpi.wordpress.com/project-list/adding-a-new-sample-based-flute-voice-for-sonic-pi/> Accessed: 2017-07-04.

- [13] : Virtual Playing Orchestra. <http://virtualplaying.com/virtual-playing-orchestra/> Accessed: 2017-07-03.
- [14] : Wave File Format. <http://www.piclist.com/techref/io/serial/midi/wave.html> Accessed: 2017-07-17.
- [15] Rosenblatt, F.: The Perceptron, a Perceiving and Recognizing Automaton Project Para. Report: Cornell Aeronautical Laboratory. Cornell Aeronautical Laboratory (1957)
- [16] Hochreiter, S., Schmidhuber, J.: Long short-term memory. *Neural computation* **9**(8) (1997) 1735–1780
- [17] : Keras: The Python Deep Learning library. <https://keras.io/> Accessed: 2017-07-03.
- [18] : TensorFlow. <https://www.tensorflow.org/> Accessed: 2017-07-21.
- [19] : Theano. <http://deeplearning.net/software/theano/> Accessed: 2017-07-21.
- [20] : Softmax. https://en.wikipedia.org/wiki/Softmax_function Accessed: 2017-07-13.
- [21] : Cross Entropy. https://en.wikipedia.org/wiki/Cross_entropy Accessed: 2017-07-13.
- [22] : Rmsprop. <http://climin.readthedocs.io/en/latest/rmsprop.html> Accessed: 2017-07-15.
- [23] : YouTube. <https://www.youtube.com/> Accessed: 2017-07-20.
- [24] : Sigmoid Function. https://en.wikipedia.org/wiki/Sigmoid_function Accessed: 2017-07-13.
- [25] : Hamming distance. https://en.wikipedia.org/wiki/Hamming_distance Accessed: 2017-07-21.
- [26] : Mean. https://en.wikipedia.org/wiki/Arithmetic_mean Accessed: 2017-07-20.
- [27] : Standard Deviation. https://en.wikipedia.org/wiki/Standard_deviation Accessed: 2017-07-20.
- [28] : Mean Absolute Error. https://en.wikipedia.org/wiki/Mean_absolute_error Accessed: 2017-07-13.
- [29] : NumPy. <http://www.numpy.org/> Accessed: 2017-07-21.

- [30] Torrey, L., Shavlik, J.: Transfer learning. *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques* **1** (2009) 242
- [31] Dixon, S.: On the computer recognition of solo piano music. In: *Proceedings of Australasian computer music conference*. (2000) 31–37