DLA Project Report - “Music Synthesis from MIDI with RNN/LSTM”

Henrik Erpenbach
Institut für Nachrichtentechnik
Technische Hochschule Köln
Cologne, Germany
Henrik.Erpenbach@smail.th-koeln.de

Philipp Kalytta
Institut für Nachrichtentechnik
Technische Hochschule Köln
Cologne, Germany
Philipp.Kalytta@smail.th-koeln.de

Victor Kerscht
Institut für Nachrichtentechnik
Technische Hochschule Köln
Cologne, Germany
Victor.Kerscht@smail.th-koeln.de

Bastian Schetter
Institut für Nachrichtentechnik
Technische Hochschule Köln
Cologne, Germany
Bastian.Schetter@smail.th-koeln.de

***Abstract*—This paper gives an overview on music synthesis based on recurrent neural networks (RNNs), especially Long-Short-Term-Memory based RNNs, that will be trained with data extracted from MIDI files and can generate MIDI data on the output side.**

Keywords—music generation, LSTM, recurrent neural networks, MIDI

I. Introduction

This student project explores the use of machine learning in the generation of music. Neural networks will be used to learn from real composed music. These same networks can be used to write their own music after a period of training. The project was a group effort of four students that worked together to discuss and implement the contents of this report. This report discusses the way the project was assembled, from the tools that were used to the way data is converted from an industry standard to a more training friendly format. The different designs and iterations of the network itself will be described. Finally, the results and conclusion that stem from these experiments will be presented.

II. Objective

The objective of our project is to generate music in the form of MIDI files using neural networks. To achieve this objective, we want to train a Recurrent Neural Network (RNN) with large quantities of MIDI files. The result should be a playable MIDI file that is pleasant to listen to by a human and that should reflect the style of music that was fed into the network. Directly using notes in a file instead of transcribing the music from a sound file helps training the network for correct dependencies on music notes. If successfully trained with LSTMs, checking other RNN variants (GRUs or bidirectional LSTMs) and comparing their performance to LSTM-based neural networks might be additional work in this project. We might also consider neural network types that were not tested for music generation in other research, like multiplicative LSTM. The project also includes creating a robust way to convert MIDI data to a format that can be ingested by the neural network, as well as the conversion back to MIDI from the predictions the neural network creates.

III. Technical Outline

A. Tensorflow

TensorFlow is an open-source framework for machine learning (ML) applications based on the Python programming language. TensorFlow originated from one of Googles internal projects and provides a stable Python API that supports training ML models using the GPU as an accelerator [1].

B. MIDI

The MIDI format or “Musical Instrument Digital Interface” (file ending .mid) is an industry standard that is in principle very similar to sheet music [2]. The information is encoded as pitches that should be played and the instrument that should play them. The real sound is not included in a MIDI file, the device reading the file must access separate tone samples to generate an audio signal. In the case of this project, MIDI files are a good match, because the notes and timestamps are already included. This allows the neural network to work with precise data, that exactly tells it which note is played when.

MIDI is a message-based format, meaning information is saved as messages for single notes or chords which are sent at a specific point in time.

C. Music21

Music21 is a set of tools for Python that allows for computer-aided music conversion, analysis, and generation [3]. Music in Music21 consists of music21.note objects that are organized in music21.streams. Streams are hierarchical objects that either contain notes (or other control objects like metronome marks) or can themselves contain other stream objects: i.e., a score (which is a stream) can contain different parts (which are also streams) which contain measures (also streams), and these contain notes [4]. We will traverse this hierarchy when parsing music from MIDI files, as MIDI also has the hierarchical structure, but based on so called tracks. When converting back from the generated data to MIDI files, we will need to recreate the hierarchical structure, at least so far that the MIDI contains the notes in a stream object. Other limits apply regarding easier conversion (see “Conversion Back to Midi”).

D. Neural Networks

A neural network is a network of neurons that can be used for different tasks such as regression analysis, classification of patterns and objects or data processing. Plain neural networks cannot detect sequential or timeseries

©2021 TH Köln

features. The problem of music generation is a multi-label classification task based on timeseries. Keys/notes can be pressed at once, most of them are not pressed at the same time. One important parameter, that also needs to be considered but is not a classification problem is the metronome, it is the one other important value of music regarding how good it sounds subjectively. The velocity of notes has shown to be less important in our testing [5]. The metronome does not need to be classified, rather it is a numerical prediction.

E. RNN

RNNs or Recurrent Neural Networks are a type of neural networks that specialize in recognizing and extending temporal sequences. Examples of such sequences are words in a text, movement of an object in a movie or the notes in a piece of music. To achieve this RNNs use data from a previous slice of data in a sequence as an additional input to the current slice, essentially creating a memory.

In our project we can use RNNs to train and then generate MIDI data that we can save to a file that is then playable by a computer.

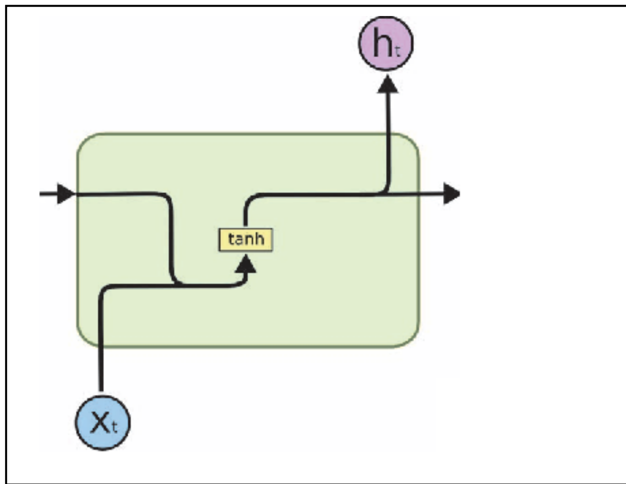

Figure 1: Simple RNN Cell [6].

F. Long Short-Term Memory

Based on the idea of an RNN using previous data slices for a better prediction, LSTMs are specialized on learning long-term dependencies. They consist of three different sigmoid gates to update the internal cell state. The input gate is responsible to decide what new information shall be stored in the cell state. The forget gate is used to forget already remembered information that is not necessary to keep while the output gate provides a filtered version of the cell state combined with an input as the output of that cell.

Regarding this project, LSTMs were chosen as the RNN architecture to be able to recognize dependencies between musical notes. Music consists of sequences of notes that can be played consecutively as well as chords when played at the same time. These sequences and chords form a melody that sounds harmonious to the human ear. A normal neural network is basically unable to decide whether a sequence of notes sounds harmonious or not.

Instead, based on a set of pieces of music as input data, patterns between notes shall be recognized. Each LSTM cell decides whether a note fits in the context of the previous played notes. In the next step, these patterns should then be reproduced by the network to create a composition by itself.

G. Gated Recurrent Unit

GRUs or Gated Recurrent Units are a type of RNN cell that introduces a gate to the RNN scheme. This gate, which is a simple mathematical operation attached to additional weights. The gate gives the cell the ability to decide whether the data from previous cells should be passed to the next cell. As such it introduces the ability to 'forget' to the network [6].

H. Data Sets

Since the RNN needs a large quantity of MIDI files to train, a lot of similarly formed MIDI files are needed. The MIDI format allows for multiple different types of saving music data. Removing metadata and bringing every MIDI file in the same format (same type and same temporal space) is important to correctly train the network with the data. Ideally, the neural network will be able to produce MIDI data that can directly be processed by a music application to play the generated sounds.

IV. Data

As mentioned above, MIDI can be compared to a sheet of music. Likewise, the music can be played by any instrument. Since it is basically possible to strike several notes on a piano at the same time, the focus in this project was set on piano pieces from classical music.

For this purpose, a collection of pieces of music by various famous composers could be found online, which was used to train the neural network [7].

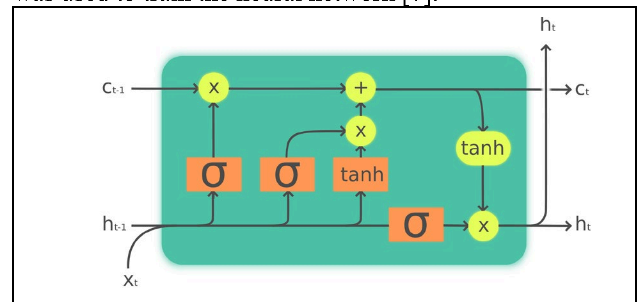

Figure 2: "Structure of a LSTM (Long Short-term Memory) cell. Orange boxes are activation functions (like sigmoid and tanh), yellow circles are pointwise operations. A linear transformation is used when two arrows merge. When one arrow splits, this is a copy operation" [11] [12].

A. Data From Midi

The MIDI file format is used because the format is, in essence, much easier to work with. Most audio file formats save the actual sound signal in some form. This way, details like specific instrumental sounds, effects, intonation, and volume are preserved. MIDI can save things like the intensity and tempo of a played sound, but further details are lost. This is because, as mentioned earlier, a MIDI-file merely consists of messages (information on which instrument to play at what pitch, at which point in time) – the actual sound itself is synthesised by the device interpreting the messages using external instrument sample data. In this way, MIDI is more like a sheet of music instead of the music itself.

©2021 TH Köln

In the case of training a neural network actual sounds are a more difficult dataset to work with. For example, an audio-cd works with a sample rate of 44.1 kHz. Considering the Nyquist-Theorem this leads to a possible signal spectrum between nearly zero and about 22kHz, which would need to be analysed for at least probably half as short as a 32nd note in the tempo of the music. This would amount in a massive amount of work analysing and parsing the data before it could be used to train a neuronal network. Considering that this project is limited in the amount of time for the research, this is unfeasible.

Using MIDI instead, we can limit the amount of input neurons for a single instrument to the different states a note can take times the amount of 88 possible notes.

Additionally, MIDI files are widely available on the internet. Some collections of music are even free to download.

B. Internal Data Format

Since MIDI is a message-based format with different standards, it is not that useful to feed into a neural network directly. Instead, we used an internal format to hold the input data. Our goal was to create a discrete-time representation of individual notes quantised over a finite range, namely the 88 keys of a standard piano. To achieve this, we structured the data like a so-called "piano roll", a method of music storage used in player-pianos and barrel organs since the 19th century. A piano roll is a long roll of paper with slotted perforations that moves over a read head while the song is played. Each slot represents a note, the lateral position of the slot encodes which note to play, while the length of the slot determines how long it should play.

Analogously, our input data is represented in a two-dimensional array-form, where each row contains the current state of the 88 keys during a single point in time. We defined three possible note-states, "silent", "start" and "hold". This is necessary to discern multiple notes played back-to-back from a single, continuous note (compare attacking a piano key four times in row to attacking it once and letting the string vibrate freely).

C. Conversion From MIDI

To generate the data, we use the aforementioned Python library 'Music21'. Music21 allows parsing of MIDI-files and other music storage formats by quantising the MIDI-messages into a stream of note-objects as they would appear on sheet music. It uses the pitch of the message to determine the note on a scale from A0 to C8 including accidentals (i.e., sharps and flats) and the offset between messages to estimate the bar and tempo in beats per minute (bpm) followed by the duration of the tone (half, eighth, quarter etc.).

The values are parsed for each track of the MIDI (different tracks can correspond to different instruments, different hands, voices, etc.) to generate the array. Because classical compositions often contain changes in tempo, the current tempo in each time-step is also added to each row. The step-size was set to a 16th note, meaning that 32nd notes and triplets are rounded to the nearest 16th, which improved the size-efficiency of the input arrays, without losing much musical information. Note that the information about which instrument a note is played by is discarded; in its current form the data structure is only suitable for solo pieces. The result is a systematic data structure that retains pitch and duration while still supporting polyphony (multiple notes played at once).

D. Conversion Back to MIDI

In reverse, we can convert generated data from the neuronal network back into MIDI. This is needed to listen to the generated music.

The algorithm works concurrently with one thread for each possible note using the python thread pool executor. Every thread works through the 16th parts, if a note is attacked, the algorithm will read how many 'hold's follow afterwards to determine how long the note is.

After that, the node will be converted into a string consisting of the note name and the octave. The arrays for the single notes are than converted into music21 streams and merged into a single stream. The resulting MIDI-file, when viewed in something like the midi visualization software 'MuseScore', will look as if 88 pianos are playing together, with every piano only ever playing a single distinct tone.

The use of multithreading reduces the time needed for the conversion if it is run on a CPU that supports simultaneous multithreading on a single core. However, multithreading will only run on a single physical core of the CPU. To use more physical cores, the use of multiprocessing is required, which splits the execution into multiple child processes, which has the potential to divide the time needed for conversion by the number of processor cores used.

Sadly, the use of multiprocessing will lead to errors in Music21 when the streams for the single tones are merged. Because of this, multiprocessing is not used in this project.

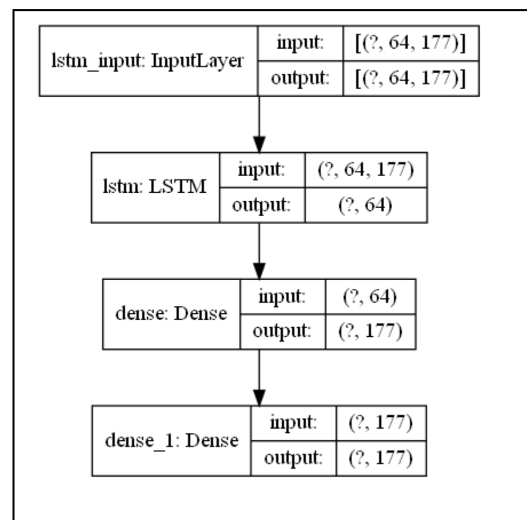

Figure 3: Base Sequential Model of the neural network used in this paper for the prediction of new piano music.

V. Neural Network Model

As the neural network will work on timeseries data - notes in a score are dependent on a time basis as well as other pressed notes at the same time, the main part of our model will be one or several RNN-type layers.

A. *Base Model*

The neural network model is a sequential model, meaning the different layers are wired sequentially after each other.

The first layer is an LSTM layer with 64 internal layers. The input of the layer is comprised of 177 neurons. This number results from the 88 possible notes that can take 3 states. The silence state is ignored because it seems to have negative influence on the output quality. Therefore the 88 notes are multiplicated by 2 to account for starting and holding a note. An additional neuron is added to pass the tempo. The LSTM layer is added to analyse the 16th samples in correlation to the past. Following the LSTM layer are two dense layers with the same 177 input and output neurons. The second dense layer functions as the output layer, from where the prediction can be read back into the internal format.

VI. Methods

As mentioned, the data consists of two important parts: The actual keys pressed at a given time and the tempo of that time. The former is a categorical probability problem: The model must predict probabilities for a keypress for the given time. The latter is a real numerical problem: There is always a tempo in the score, it might change by a numerical value. This imposes a problem regarding Tensorflow using Keras: The simple sequential model can only be used with one loss function, which can therefore only optimise towards one of the two named problems. Either reducing loss on the probability of the keypresses or reducing loss on the numerical values of the tempo. This will further be discussed in the following section A.

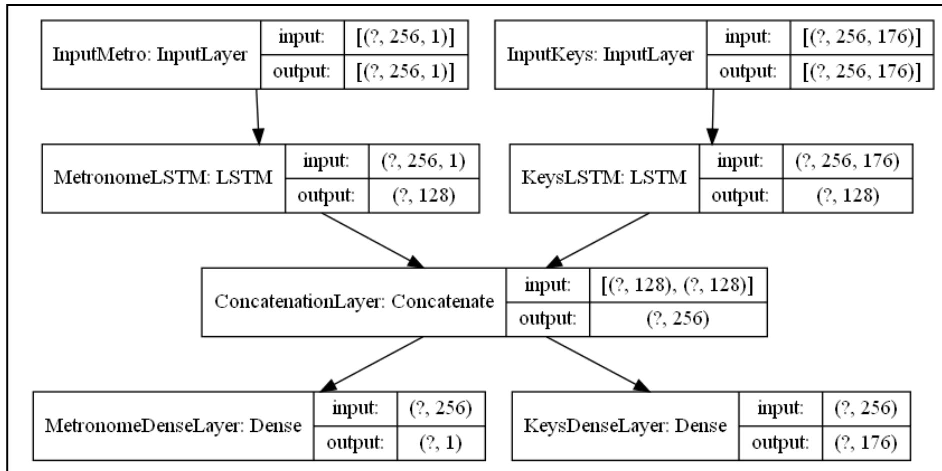

Figure 4: RNN Split Model using the Keras Functional API

A. *Solution Approach*

The base model, as stated, consists of one large LSTM layer with 61952 parameters, which is followed by two dense layers with 177 neurons each, which results in a total of 104,963 trainable parameters. The model performed slightly better with two dense layers, about 0.1 % higher accuracy than with only one. Deeper LSTMs, however, did not further increase the accuracy. This is mainly because the loss correction is flawed as mentioned above.

1) *RNN Split Model*

The issue of the two different problems that arise with the music data can be circumvented with a split model. Tensorflow and Keras allow to do this with the Functional API. The Functional API is more flexible regarding the design of the neural network, particularly, it allows for multiple input and outputs. This is necessary as we can define a different loss function on each of the output vectors. This enables us to use a cross entropy loss function for the pressed keys output and mean squared error for the numerical loss on the metronome (or tempo).

2) *Activation Functions*

The LSTM layer uses tanh as the activation function, as this is the only activation function for LSTMs that allows the use of cuDNN (Training on Nvidia GPUs) [8]. The first dense layer uses the default linear activation function. The last dense layer shows the best results with “softplus” activation. ReLU and sigmoid will yield slightly worse results.

3) *Loss Functions*

We chose mean-squared-error for the loss function for the non-split model over the full output vector. Using a cross entropy loss would be beneficial for the probability-dependent outputs of the pressed keys but would completely invalidate the output for the tempo. To preserve meaningful output, mean squared error will result in good output for the tempo value, and will try to estimate a numerical value for the pressed keys: This will result in very small numbers less that one for the key outputs (Info: those would go in as either 0 or 1) which cannot be directly interpreted as probabilities. Those output predictions must be interpreted by a decision algorithm to evaluate if a keypress needs to be put into the output MIDI file.

4) *Optimizers*

When trying to achieve better results optimizers can be used to dynamically adjust the weights on the nodes. The first versions of the LSTM used Adam, a stochastic gradient descent method as an optimizer that is a combination of RMSprop and momentum. Therefore, it includes the exponentially decaying average of past squared gradients as well as past non-squared gradients to compute an adaptive learning rate.

However, Liu and Ramakrishnan [II] stated RProp provides better results since it only uses the sign of the derivative to adjust the learning rate and does not take the size of the partial derivatives into account. First tests of the LSTM confirmed that better result could be achieved. However, since RProp has difficulties to work with large datasets and its similarities to it RMSProp has been set as the optimizer of choice.

B. *Alternative Approaches*

GRU cells (Gated Recurrent Unit) can be used as an alternative to LSTM and were evaluated in comparison to the LSTM by replacing the LSTM layer with a GRU layer with the same number of units. This would reduce the training time by about 10-15%, as it also reduced parameter size to 89,667. But it did not yield better results in terms of the generated music.

Figure 5: Output from training on a Bach Score, this shows the problem with too many rests between notes.

VII. Experiments

A. Musical Rests

First experiments showed an interesting behaviour of the RNN regarding musical rests.

In many classical compositions, the intro begins with a rest that can last several seconds until the first note is played. The same applies to the outro, which may contain multiple rests after the last musical note played until the composition is finished. Musical pauses are also often used during a piece of music, which results in a high number of pauses used in total within one composition.

Because of this high number, the neural network interpreted rests as something positive. Also, because the loss decreased when the RNN played rests, the first outputs of the network only included breaks.

To avoid this behaviour in the first step the rests in the beginning and at the end of a composition have been removed from the input data.

B. Hyperparameter Manipulation

The Hyperparameters we chose to change throughout different runs were the following: Learning rate, epochs, and batch size, to compare them with different settings. Best results were achieved with 20 to 25 epochs with a learning rate of 0.002. This was determined with a batch size of 64 and a deepness of 64 for the LSTM layer. Training with more epochs would slightly reduce the loss on the training data, but nearly no change on the validation set. This can also be seen in Figure 5. Training with more than 100 epochs will then result in a quick spike in categorical cross entropy loss, after dropping rapidly to near zero categorical cross entropy loss. This indicated that the network learned to optimize against either zero or one for the key presses – which it did: after 150 epochs, the prediction output would only contain zeroes. We then settled with the conservative value of 25 epochs to further test on batch size. Batch size did not at all influence the output of the neural network, mini-batches performed in the same 0.001 loss interval around 0.015 as large batches (See Figure 6).

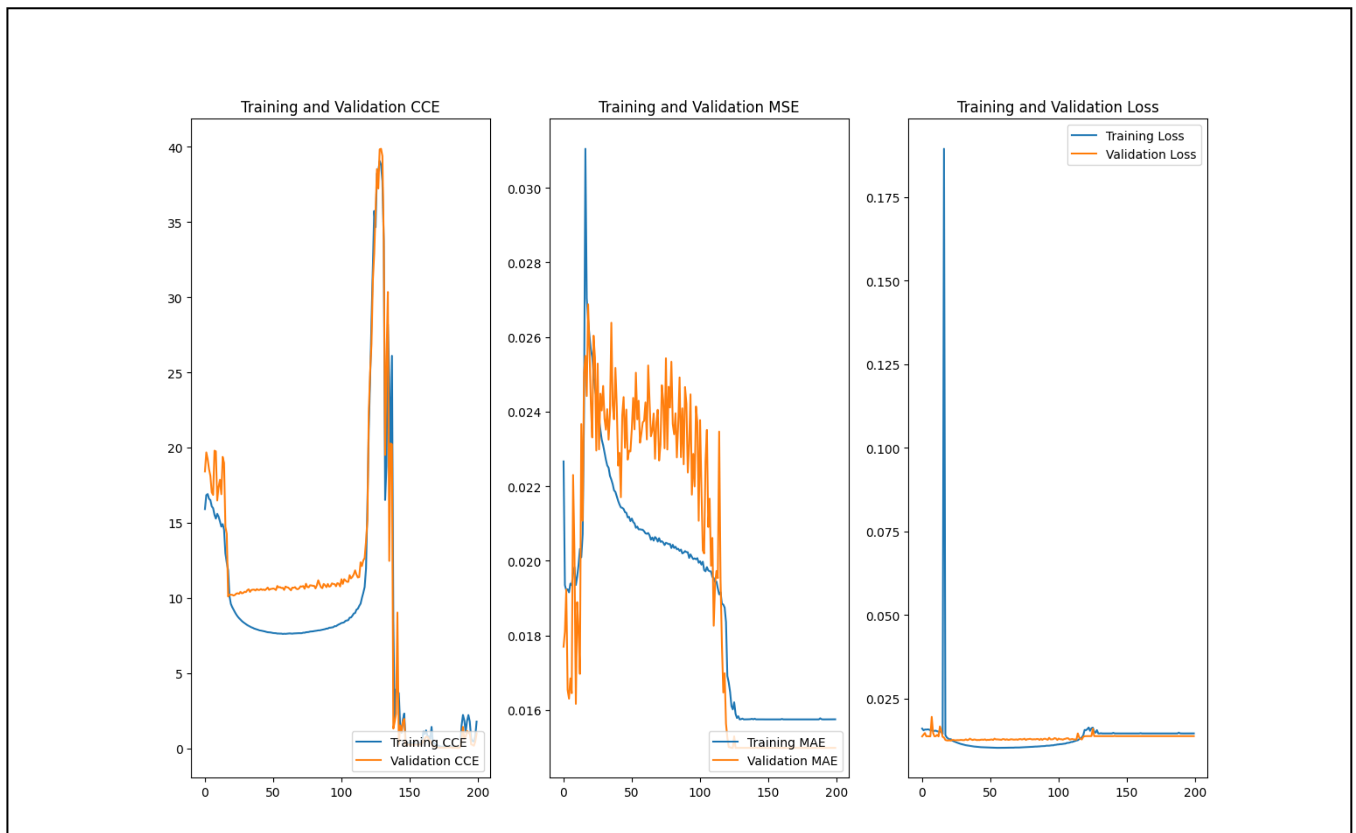

Figure 6: Training and validation categorical cross entropy, mean absolute error and loss for training with batch size 64 and LSTM layer size 64 for 200 epochs on the base model.

©2021 TH Köln

C. Regarding the Architecture (Split/Non-Split)

All the shown experiments were done using the base model. We were not able to work with the split model architecture from the Keras Functional API due to restrictions with the Tensorflow Implementation and the way our dataset was formed. We would need to completely rework the data ingestion code to be able to train the split model [9]. This was not doable in the given amount of time. As our data is asymmetrical – consisting of only one value for the tempo and 176 values for the keys per timestamp. This would make NumPy arrays inappropriate for the input and output, as we would have 175 empty values for one of the input vectors, producing unexpected input and output. Also batching these inputs correctly needs to be done as well as creating tensors for each input for the training and validation data. Doing these steps on each dataset itself would result in Tensorflow failing on running the first epoch. Ingesting the whole dataset at once would result in Tensorflow not being able to create Tensors from the arrays. Also, this made using timeseries data generators unusable as for generating the data for the whole (very large) dataset.

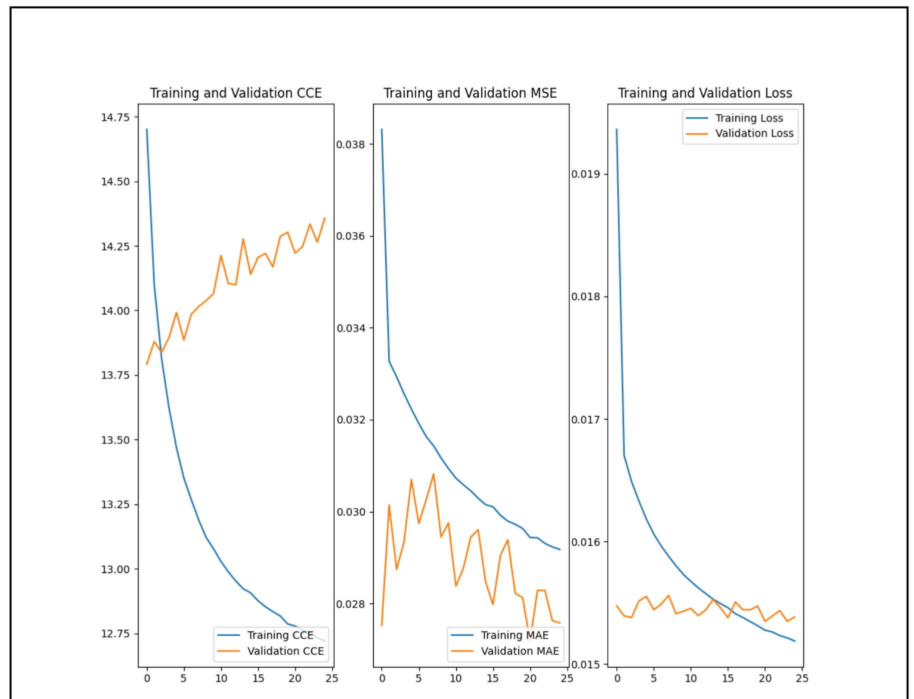

Figure 7: Training and validation categorical cross entropy, mean absolute error and loss with batch size 128 and LSTM layer size 64 for 15 epochs on the base model. All batch sizes performed like this one, if the other hyperparameters were not changed.

D. Data Generator for Input Data

Training on a single composer was possible with a standard 16 Gigabytes of system memory. Ingesting the whole music dataset at once and windowing the data for the model training results in requiring over 200 Gigabytes of system memory. This is not feasible for training, but we wanted to have to option to train on the whole dataset. This required us to use data generators. Tensorflow Keras provides the TimeseriesGenerator class for working with timeseries data, which fits our music data. In Tensorflow 2.0, directly calling model.fit() with generators is possible (instead of using fit_generator()) which further reduces the effort for training with the whole dataset. With a Nvidia RTX 2070, training on GPU was done with a batch size of 256, which took about 5 minutes per epoch to train with the whole dataset. Loss remained about the same 0.013 that was achieved with single composers. The predicted music subjectively has the same complexity and errors like training with a single composer.

VIII. Results

A. Music Generation

Regarding the goal of the generation of music, there are promising results. The resulting pieces of music tend to have the problem of long pauses and few variations of notes played, depending on the threshold used to analyse the prediction. In other cases, most notes will be played in perpetual 16th all the time.

Nevertheless, in some of the generated pieces, accords are played in succession which will sound good. The results of course do not compare with the complexity of the music fed into the network. The GRU network compared to the LSTM performed better regarding an overall theme for the score, but used significantly more notes, which results in an oversaturation effect for the listener. The GRU overall has the same problem: The network cannot be perfectly optimized regarding the loss, as it the input and output is treated in the wrong way (only one input/output vector). This effect seems to prevent the network to train any further after a given point.

IX. Conclusion

We can conclude that even though we did get music with some accords played, out network would need to be much larger and more complex to learn to imitate a classic composer. The corresponding time and resources the network would need were too much for the scope of this project.

In general, our dataset would need to be restructured. Most of the problems came from two facts. First, the metronome was included in the input vector. This input had a different 'encoding' from the note neurons and was never zero. We theorize that this adds an adverse effect on the quality of the prediction. One possible solution to this problem would be the introduction of a second network that calculates only the tempo for the slices, whereas the first network would be devoid of any tempo inputs. If Tensorflow would be able to work with multiple input generators for timeseries data, this would also enable the split RNN model described before to work with the given data.

The second severe problem was the number of zeroes fed into the network. As discussed, this problem arose from the fact that a single note in a piece of music will be silent more often than it being played. As a result, the training data leads the neuronal network to believe that zeroes are the best possible output with the least amount of error generated during training. This might be mitigated by using a bias on

©2021 TH Köln

these inputs to reduce the impact, alternatively, this would also be approached by using a different loss function. LSTMs should be able to ignore parts of the input if they do not contribute to the output over time.

A more complex network with a more complicated input tensor and more complex model with more layers and neurons is probably the solution to this problem (See the Functional API of Tensorflow/Keras for reference [10]).

Another factor to the solution of this may be the introduction of a dynamic interpretation of the data. Whereas in the project a static threshold was used to decide if a note is played or not, changing this to a more intelligent algorithm may lead to better results in the generated music, ideally, the network would be deep enough to be able to decide this for itself, only outputting correct values that can be directly translated to a working MIDI file.

X. Related Works

Other research focuses on LSTM performance for music generation and recognition as well as gated recurrent units (GRUs), which is a gating mechanism for recurrent neural networks, especially LSTM.

In Liu and Ramakrishnan [II] a LSTM network has been used to learn structure and rhythm of musical compositions. The results were similar pieces of music composed by the network itself. As stated in previous chapters, this project used a similar data structure regarding the musical notes represented as an 88-value input vector for the whole range of a piano from A0 to C8.

Nayebi and Vitelli [III] used raw audio samples extracted from songs encoded in WAV format as input for a LSTM network instead of MIDI files as used in this project. However, they discovered performance impact as an effect of adding layers of recurrent units into the network.

In other works, there are also variants like Peephole LSTM, hyperLSTM, feedback LSTM and the new multiplicative LSTM (mLSTM) discussed.

[I] J. Nam, J. Ngiam, H. Lee, M. Slaney, “A Classification-based Polyphonic Piano Transcription Approach Using Learned Feature Representations”, Ismir, 2011

[II] I. Liu, B. Ramakrishnan, “Bach in 2014: Music Composition with Recurrent Neural Network”, ICLR, 2014

[III] A. Nayebi, M. Vitelli, “GRUV - Algorithmic Music Generation using Recurrent Neural Networks”, Stanford, 2015

[IV] Z. C. Lipton, J. Berkowitz, “A Critical Review of Recurrent Neural Networks for Sequence Learning”, UCSD, 2015

[V] B. Krause, L. Lu, I. Murray, S. Renals, “Multiplicative LSTM for sequence modelling”, 2016

[VI] D. Ha, A. Dai, Q. Le “HyperNetworks”, 2016

XI. References

[1] vrv, claynerobison, caisq, Rishit-dagli, case540, ashahba, andrewharp, gelembungsahabat, MarkDaoust, keveman, lamberta, VersatileVishal, martinwicke, wdirons, Nayana-ibm, jhseu, bzhaoopenstack, benoitsteiner, 8bitmp3 und mihaimaruseac, „tensorflow/README.md,“ TensorFlow, 16 10 2020. [Online]. Available: https://github.com/tensorflow/tensorflow/blob/v2.4.1/README.md. [Zugriff am 04 02 2021].

[2] MIDI Association, „The MIDI Association - Home,“ The MIDI Association, 2021. [Online]. Available: https://www.midi.org/specifications. [Zugriff am 04 02 2021].

[3] M. S. Cuthbert, „music21: a Toolkit for Computer-Aided Musicology,“ [Online]. Available: http://web.mit.edu/music21/. [Zugriff am 04 02 2021].

[4] M. S. Cuthbert, „User’s Guide, Chapter 6: Streams (II): Hierarchies, Recursion, and Flattening — music21 Documentation,“ 03 02 2021. [Online]. Available: http://web.mit.edu/music21/doc/usersGuide/usersGuide_06_stream2.html. [Zugriff am 04 02 2021].

[5] V. Kerscht, H. Erpenbach, B. Schetter und P. Kalytta, „Sciebo MIDI Examples Download,“ [Online]. Available: https://th-koeln.sciebo.de/s/To8uIJaHB1z0384. [Zugriff am 04 02 2021].

[6] saurabh.rathor092, „medium.com,“ 02 06 2018. [Online]. Available: https://medium.com/@saurabh.rathor092/simple-rnn-vs-gru-vs-lstm-difference-lies-in-more-flexible-control-5f33e07b1e57. [Zugriff am 10 02 2020].

[7] piano-midi.de, „piano-midi.de,“ [Online]. Available: www.piano-midi.de. [Zugriff am 11 2020].

[8] The TensorFlow Authors, *Long Short-Term Memory layer - Hochreiter 1997 - recuurrent_v2.py:912,* 1997.

[9] L. Geiger, „tf.keras multi input models don't work when using tf.data.Dataset,“ 11 07 2018. [Online]. Available: https://github.com/tensorflow/tensorflow/issues/20698. [Zugriff am 01 02 2021].

[10] The TensorFlow Authors, „The Functional API,“ 20 01 2021. [Online]. Available: https://www.tensorflow.org/guide/keras/functional. [Zugriff am 10 02 2021].

[11] G. Chevalier, „Structure of a LSTM (Long Short-term Memory) cell,“ 2020.

[12] G. Chevalier, „LSTM cell - Long short-term memory,“ Wikimedia, 23 09 2020. [Online]. Available: https://commons.wikimedia.org/wiki/File:The_LSTM_cell.png. [Zugriff am 04 02 2021].

©2021 TH Köln