A recurrent neural network for classical music generation

Fredrik Lundkvist
KTH Royal Institute of Technology
flundkvi@kth.se

Emil Clemendsson
KTH Royal Institute of Technology
emilcl@kth.se

Abstract

Recurrent neural networks are well suited for learning from data with a temporal structure, such as speech or music. In this paper, we build and train an RNN with long short-term memory (LSTM) to generate classical piano music, using over 160 hours of training data in MIDI format. While the final network failed to generate what we would classify as music, the results of our experiments do present some interesting discussion topics about performance plateaus and the importance of preparing your data in an optimal way.

1 Introduction

Recurrent neural networks (RNNs) are well suited for problems which have some form of sequential or temporal factor to consider; often related to fields such as speech, text, or music. In this paper, we present our process for training an RNN with long short-term memory (LSTM) to generate classical piano music from a single starting note. The network was trained on over 160 hours of piano performances from the MAESTRO dataset (Hawthorne et al. [2019]), in the form of MIDI files. While research has been conducted on RNNs and music generation, these studies have used waveform data, rather than MIDI, which encodes musical information as a series of symbols instead of a continuous signal. Research using midi files have used other architectures, for example CNNs. The aim of this project is to investigate the generative capabilites of RNNs in the music domain when trained on MIDI data.

2 Related Work

Using deep networks for music generation is not a novel idea, as quite a lot of research has been conducted within the field. Some notable examples that inspired us for this project are OpenAI's MuseNET (Payne [2019]), and Google's CocoNet, which was used to create Bach-like compositions (Huang et al. [2017]). However, none of these models were RNNs, with MuseNet being a transformer network, and Coconet being a CNN.

For recurrent neural networks, the long short-term memory cell (LSTM), is an extension of a standard hidden layer for RNNs, including so-called "memory gates", that decides how much of the data at a time step t should be remembered in a longer term than the next hidden state (Hochreiter and Schmidhuber [1997]). This allows RNNs with LSTM modules to remember longer-term context, which is a limitation found in vanilla RNNs. Another hidden layer concept used in RNNs is the gated recurrent unit (GRU). Since we do not use GRUs in this project, we will not explain the concept further; we do however want out that different approaches for longer-term memory exist.
In 2015, Nayebi and Vitelli compared the performance of GRU and LSTM RNNs for the task of generating music based on waveform data, and found that *"the generated outputs of the LSTM network*

were significantly more musically plausible than those of the GRU" (Nayebi and Vitelli [2015]). Our decision to use long short-term memory in our network was based on their findings.

3 Data

3.1 Dataset

For all experiments described in this paper, we used the MAESTRO data set (Hawthorne et al. [2019]), which consists of over 200 hours of virtuoso piano performances, split into training, validation, and test sets. All data is available in both waveform and midi format. We opted for using midi data for two reasons, the first being that we have not found research on RNNs and MIDI data, and the second being storage space restrictions. The waveform data requires many gigabytes of storage space, while the MIDI versions require less than a gigabyte of storage in total. Regardless of file ofrm, this dataset provided more than 160 hours of training data across 967 performances, which we deemed to be enough for this project.

3.2 Data processing

All midi files were processed for use by network by parsing using the python library music21[1]. Note pitches were extracted, and saved in a JSON file on disk. This was done once for the entire dataset; throughout the experiments, data was loaded directly from the JSON file. In order to reduce complexity in the input for the network, we opted to only work with single notes and chords, omitting factors such as intervals, strike and release times, and strike force. A special character was added to the pitch alphabet in order to allow the construction of chords without the alphabet becoming too large to handle.

Mappings were created between symbols in the musical alphabet and integer values, in order to create numerical encodings for network input and output. To ensure consistent conversions between pitch and integer between different machines and dates, these mappings were generated once, and saved in separate files that were loaded by the network when started.

Input data for the network consisted of one-hot vectors, with the index related to the specific note being set as one. Due to peculiarities with pytorch, each sequence of data was represented as a

$$n \times 1 \times d$$

tensor.

4 Methods

In this section, we will explain the network used in this project, and our methods for optimization and evaluation.

4.1 The network

For this project, we decided to use a recurrent neural network, since they are built to handle sequential data. Since MIDI data can be seen a series of musical symbols, we believe that an RNN would be the architecture best suited for the task, since they are good for problems concerning sequential data, for example speech or text. However, a vanilla RNN would struggle with generating believable music, since tit lacks the ability to account for greater contexts than a few of the most recently encountered symbols. Thus, we decided to use a long short-term memory cell (Hochreiter and Schmidhuber [1997]) (LSTM) for our hidden layer, to allow the network to handle longer contexts, which we believe to be crucial in music.

The network had an input and output dimensionality of 90 (the amount of symbols in our processed data), and a hidden state and memory size of 64. While larger hidden states are common, we decided to use 64, since it is a good number for GPU computations (being a power of 2), and is not bigger than our input. All code used to implement the network can be found on github[2].

[1] http://web.mit.edu/music21/

[2] https://github.com/Lunkers/DD2424-lstm-music-generator

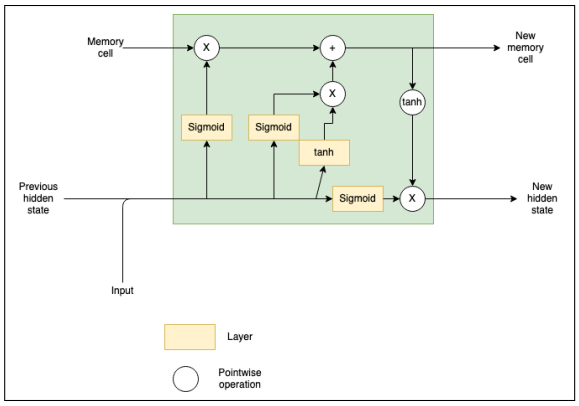

Figure 2: composition of an LSTM cell.

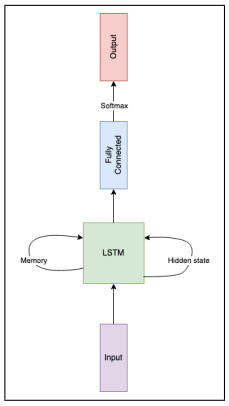

Figure 1: Computational graph for our network.

The network was implemented in python 3.7, using pytorch[3]. While pytorch has a built-in LSTM layer, we decided to implement it ourselves, as a learning exercise (author's note: we did not implement gradient calculations ourselves, these were calculated using pytorch's implementation of autograd).The network consisted of an input layer, an LSTM layer, and a fully connected layer, see figure for a drawing. We used cross entropy loss as our loss function, using the built-in functions in pytorch to calculate it. Data from the LSTM layer was then passed through a fully connected layer using softmax to normalize symbol probabilites.

Before training the final version of our network, we conducted a series of experiments to find optimal settings for certain hyperparameters (optimizer choice, sequence length, and learning rate). These experiments will be described later in the text.

4.2 Hardware

All experiments were conducted on the Google cloud platform, using two CPUs, 16GB of ram, and one Nvidia T4 GPU.

4.3 Optimization

Before training the final version of the network, we conducted three optimization experiments in order to find suitable settings for the network optimizer, learning rate, and sequence length. For each update step, training data was selected by random draw, meaning that a a random index was selected, and a the following n samples were used as training data. This method was used for all experiments described in the report.

[3] pytorch.org

4.3.1 Optimizer

The first experiment conducted involved finding the best optimizer (update algorithm) for the network. We decided to compare Stochastic gradient descent (SGD), AdaGrad (Duchi et al. [2011]), and ADAM (Kingma and Ba [2014]). For each optimizer, we trained a new version of the network for 1 epoch, with a sequence length of 25 and a learning rate of 0.001. all optimizer values except for the learning rates were set to the default values provided by pytorch. After training, the performance of each network was evaluated by calculating the predictive accuracy on a validation set consisting of roughly 250 000 symbols. The optimizer with the highest accuracy was chosen (see 5.1.1 for tables), and used for all other experiments described in this report.

4.3.2 Learning rate

For this experiment, we investigated whether using cyclical learning rates (Smith [2017]) would improve network performance. We trained two networks, one using a set learning rate of 0.001, and one using cyclical learning rates with an upper bound $\eta = 10^{-2}$, and lower bound $\eta = 10^{-4}$, with the best-performing optimizer from the previous experiment, comparing their accuracy on the validation set. The two networks where trained for 1 epoch each. Just as in the previous experiment, the setting used in the best performance was used for all future experiments.

4.3.3 Sequence length

The final optimization experiment was designed to find the optimal sequence length for the network. We trained 4 different networks for 1 epoch, with the best-performing learning rate and optimizer found in the previous experiments. Once again, we evaluated network performance by comparing network accuracy on the validation set, choosing the sequence length with the highest accuracy.

4.4 Final network

Here we present how we evaluated the performance of the final network. After finding suitable settings for optimizer, learning rate, and sequence length, we moved on to training the final network. This network was trained for three epochs. After three epochs, the network with the lowest loss encountered during the training process was selected for evaluation, and tested its accuracy on the test data, which consisted of roughly 1 million symbols.

Aside from testing predictive accuracy, we also used the final version to generate a number of MIDI files, which we listened to and evaluated subjectively. While this is not a quantitative evaluation, it is still useful, since it allows us to evaluate how "human" the output sounds, which is very hard to do quantitatively.

5 Experiment results

5.1 Optimization experiments

5.1.1 Optimizer

The results of the optimizer experiment can be seen in the table below:

Optimizer	Accuracy
SGD	42.4%
AdaGrad	42.4%
ADAM	46.1%

Based on these results, we decided on using ADAM as our optimizer going forward. One interesting thing to note is that AdaGrad gave no noticeable improvement on performance in our scenario compared to vanilla SGD. There are many possible reasons for this; perhaps the randomly selected training data was not as varied for the AdaGrad network as it was for the SGD one, or perhaps we would only have a difference if we had trained the network for longer. Since the amount of data was very large, we believed that one epoch should be enough, and therefore went ahead with ADAM.

5.1.2 Learning rate

The results of the learning rate experiment can be seen in the table below:

Method	Accuracy
Set learning rate	42.4%
Cyclical learning rate	42.4%

As shown in the table, both methods got the exact same result; in our data, they were completely equal on accuracy, as far as python's numerical precision computed it. Thus, we had to make our own judgement on which method to proceed with, and decided to use cyclical learning rates, as we believed that adjusting the upper bound could improve results but a set learning rate would require more work for the same theoretical gains, or might not improve at all.

5.1.3 Sequence length

Based on the results from the previous 2 experiments, we trained 4 different networks for epoch each, with varying sequence lengths. All networks used ADAM and cyclic learning rates, with $\eta_{max} = 10^{-3}$ and $\eta_{min} = 10^{-4}$. The results of the sequence experiment can be seen in the table below:

Sequence length	Accuracy
25	42.4%
50	42.4%
100	43.9%
200	42.5%

With the result of this experiment, we had the hyperparameters for our final network.

5.2 Final network

The final network was trained for three epochs, with a cyclic learning rate ranging from $\eta_{max} = 10^{-3}$ to $\eta_{min} = 10^{-4}$, ADAM optimizer, and a sequence length of 100 symbols.

5.2.1 Test accuracy

The best-performing network from the three epochs of training achieved a final accuracy of 41.7% on the test data set. This is actually lower than any network achieved in the optimization experiments. However, this might be due to the fact that our test set was roughly four times larger (1million symbols) than our validation set, with greater variance in sequence structure as a result.

5.2.2 Generated music

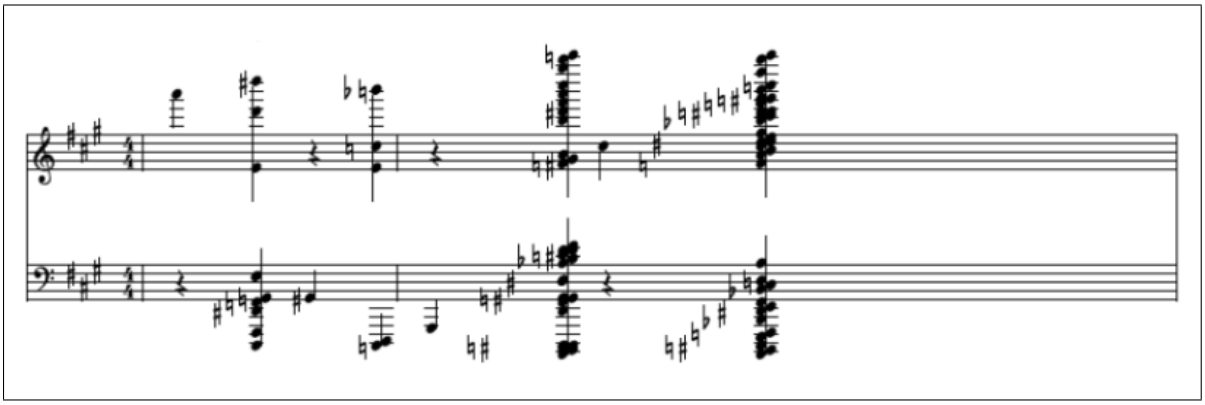

Figure 3: Sheet Music generated by our network, a sequence of 100 symbols

As one can see from reading the notes, the output of our network is not at all close to the intricate compositions it was trained on; the network seems to favor generating chords made out of 10+ notes, rather than constructing something resembling a melody.

One thing to note about the generated music is that many of the symbols generated by the network are not seen in the sheet music, or heard in the audio files. This is due to the fact that we constructed chords out of individual notes, rather than giving each possible chord its own symbol, which would have led to an unmanageable amount of dimensions in our input data. Because of this, we had to introduce a `<next>` symbol in order to interpret and construct the data correctly. Because of this, many of the generated symbols are `<next>`, and are not noticeable in the output. We also believe this to be the reason for the network constructing the unreasonable chords seen in figure 3.

5.2.3 Discussion

We believe that we might have hit a limit for how accurate this model can be in these circumstances, since the accuracy in the final model is not noticeably higher (it is in fact lower) after 3 epochs of training than any network was after one epoch, though this could be explained by the amount of test data as mentioned earlier. Thus, we believe that one will face diminishing returns of training the network for many epochs. although we cannot prove this conclusively, our experiment data indicates that this is the case.
There are many factors that could have led to these results, besides the architecture. The way we processed our data might have been a factor, as mentioned in the previous section. Continuing on the topic of data used, we also theorize that the MIDI format might have been too complicated for this project. Each note has quite a few variables, such as pitch, timing, and strike force to name a few. For the same reasons as our chord implementation (input data complexity), we chose to ignore all but pitch. This could explain why we were not able to find much work on music generation using MIDI; waveform data is easier to process since it does not have the same amount of complexity, and thus leads to more realistic results. Perhaps picking random sequences for training was a bad idea, and simply looping over the data set would have led to better results. The performance plateau might even have been the result of a bug in our implementation of the LSTM layer.

There are many factors that can be discussed when it comes to the design of our network. The first one is depth: we only had one hidden layer in our network, and could have added more. Since we were implementing the LSTM module ourselves, we ran into issues with the forward and backward passes in pytorch when we tried to go deeper. In order to produce results and finish the project in time, we decided to only use one hidden layer.

For tasks such as this, evaluating a network based on predictive accuracy might not be the most optimal strategy, since there is a large variance in both training and validation data. There is a large variance in sequence structure, chord structure, and piece structure, meaning that it is a very hard task for the network to predict what will come next. While there are some formal rules for musical structure, they are not as strict as those for other problem spaced RNNs are used in, such as language processing. Furthermore, many composers make a statement in going *against* these rules, making the task even harder.

5.3 Ideas for future research

To build upon the investigations presented in this paper, we believe it would be interesting to make the network able to handle multiple variables in the MIDI format, such as interval times and strike force, since we believe that these parameters can make the output of the network sound more human. We are also of the belief that it would be interesting to conduct the same experiment on a deeper version of the network, to investigate how the amount of recurrent layers affects the final output. Perhaps this would also lead to better performance in quantitative tests such as predictive accuracy ,as we seem to have reached a limit for the implementation presented in this report.

Another related topic we think would be interesting to investigate is is sequence length scheduling. Would an approach similar to cyclic learning rates affect the output of the network in some way, and if so, how? As shown in our optimization experiments, choosing the right sequence length impacts network accuracy, which leads us to believe that a schedule approach might be useful here, just as it is with learning rates.

6 Conclusion

In this project, we built a recurrent neural network with long short-term memory with the purpose of generating music. While the network does not really generate what we as humans would call "music" (although some may call it avant-garde), we still believe that our results are interesting, since we seem to have found a performance plateau for our implementation. We cannot claim what exactly caused this, but it sets the stage for interesting and valuable discussions concerning how different parts of an implementation affect final performance.

References

J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(Jul):2121–2159, 2011.

C. Hawthorne, A. Stasyuk, A. Roberts, I. Simon, C.-Z. A. Huang, S. Dieleman, E. Elsen, J. Engel, and D. Eck. Enabling factorized piano music modeling and generation with the MAESTRO dataset. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=r1lYRjC9F7`.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. doi: 10.1162/neco.1997.9.8.1735. URL `https://doi.org/10.1162/neco.1997.9.8.1735`.

C.-Z. A. Huang, T. Cooijmans, A. Roberts, A. Courville, and D. Eck. Counterpoint by convolution. In *International Society for Music Information Retrieval (ISMIR)*, 2017.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

A. Nayebi and M. Vitelli. Gruv: Algorithmic music generation using recurrent neural networks. *Course CS224D: Deep Learning for Natural Language Processing (Stanford)*, 2015.

C. Payne. Musenet, 2019. URL `openai.com/blog/musenet`.

L. N. Smith. Cyclical learning rates for training neural networks. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 464–472. IEEE, 2017.