Generating Music with Deep Learning

Spoken and Written Language Processing

Alex Carrillo
Pau Magariño
Guillermo Mora
Robert Tura

June 19, 2020

Contents

Abstract

This paper presents various approaches of music generation based on deep learning techniques. First, after a short introduction to the topic, the article analyses three early proposals: a LSTM, a Variational Autoencoder and a GAN, for symbolic music generation on MIDI, and we gently illustrate the variety of concerns they entail. Second, we leverage the power of a novel architecture called MidiNet by conducting our own refinements, and give examples of how this model, equipped with such a conditional mechanism on melody, can achieve very promising results on music generation.

1 Aim and motivation

The goal of this paper is twofold. On the one hand, the principal aim of this project is to study the capacity of modern deep learning techniques to automatically generate creative content: in particular, musical samples. The focus in this task is on the analysis of various network architectures and on its application. In addition, through it, we put into practice and consolidate the knowledge learned during the course and the information acquired when consulting papers about music generation.

On the other hand, the main motivation of this project is the fact of working with music. Since we have never worked with it before, we are quite excited about creating new songs. Also, it will be challenging to analyze results that are not only an accuracy percentage or a loss function. In addition, we are thrilled to face the problem of providing royalty-free commercial music for multiple purposes and content creators.

2 Introduction

In the following we introduce the main concepts of the elements that lead to the design of different deep learning-based music generation systems. That is, the nature of the musical context to be generated (the representation), the data on which we rely (the datasets) and the strategy on manipulating this data (the preprocessing). It is clear that before even start processing a dataset, a proper format must be set for the data, *i.e.* the nature of the representation of a piece of music to be interpreted by a computer. Its choice and encoding is tightly connected to the configuration of the input and the output of the architecture, but in wide terms the main choice is between audio versus symbolic representation.

As stated, this project could use two different approaches when it comes to preprocessing and data transformation: the first one is to use file audios and the second one is the usage of MIDI files (although some other approaches can be considered). When it comes to audio files, there are some pros and cons about them. On the one hand, some of the advantages are the amount of data that can be fed into the network, the possibility of transforming the sound to a spectogram and other transformations to extract feature information. On the other hand, there are also setbacks when using audio files. For instance, the amount of computational power needed to train a model with audio files is huge and the task behind differentiating parts of a song that happen at the same time can become very challenging.

2.1 MIDI

MIDI (*Musical Instrument Digital Interface*) is a standard that allows instruments such as a piano to communicate with a computer in a discrete and non-continuous way, *i.e.* note by note. This type of files can be used for any musical instrument and therefore our method of generating music could be extrapolated to some other instrument, not without some changes in the chord and melody preprocessing.

Moreover, another positive feature of MIDI is the possibility to decide the sampling frequency of the song, a particularity which allows us to adjust it like an hyperparameter, depending on our needs.

2.2 Piano roll

From those MIDI files described above we can extract what is known as a *piano roll*. This piano roll constitutes the base of our models, as the note events of a MIDI channel can be represented by it (synthesizing the notes and its velocity that are played at each time instant). This means that notes, which are the most primal elements, are also fed with a velocity that indicates the amount of pressure with which a key would be held in a real piano, *i.e.* the strength of the note. Both notes and velocity, along with the time stamps, are present in these piano rolls.

With all this information that we can extract directly from piano rolls, we can represent a song as a h-by-w matrix $\mathbf{X}$, where h denotes the number of MIDI notes we consider (actually MIDI uses 128 distinct notes) and w represents the number of time steps we use per window (also 128 in this case), emulating the shape of a piano roll. Now, the entries of the matrix contain the velocity of each note: from 0, meaning a note key has been released, to 127, meaning the key is being pressed with maximum intensity. So $\mathbf{X} \in \{0, \cdots, 127\}^{h \times w}$ is the data we use. We leave a link[1] to a website that might help understand how piano rolls work and how can be used on any instrument or sound.

2.3 Dataset and Preprocessing

In the span of the project we have used two different datasets, both based on MIDI files, but with different information in the core of the MIDI.

The dataset we used in early models is called MAESTRO (MIDI and Audio Edited for Synchronous Tracks and Organization), which is composed of over 200 hours of virtuous piano in which the creators joined forces with a musical competition in order to get proper samples to work with. The dataset can be found[2] in audio format or MIDI format. From this dataset we can extract piano rolls that can be used to know which notes are played at every time instant. But there is an issue with this format: we can not divide these notes between the ones from the melody and the ones from the chord, because there is no additional information and therefore we can not distinguish when two notes should be played at the same time because they are part of a chord.

[1] https://musiclab.chromeexperiments.com/Piano-Roll/

[2] https://magenta.tensorflow.org/datasets/maestro

3 First approach

3.1 First models

As said, for the first models we only had the piano roll information to feed them. Note this will hugely impact the quality of this outputs when compared with the last ones we got.

In order to check the results provided by each model, you can access this link[3], while the implementation of these models and the ones at section 5 are available at this Github repository[4].

3.1.1 LSTM

The first model we tried was a Recurrent Neural Network, more exactly a Long-Short-Term-Memory RNN. The main reason that led us to using a LSTM is the knowledge we have on music. We know that a song is divided into several parts that are repeated over the song, so we thought about the idea of using a RNN that could store information in both the long term (the repetition of the known parts) and the shorter term (correlation with respect to the last notes played).

As this was our first model, we did not explore many changes with respect to the baseline. As soon as we made it work, we tried different train set sizes, epochs and batch size, but there was no significant change in the outputs that the model created. One parameter that had a visible impact in the output was the sampling frequency of the preprocessing, since the sampling frequency of the input will be the same at the output. We observed that when it was higher, the songs were "crazier". That is due to the fact that the LSTM was generating single notes, so the higher sampling frecuency, the higher number of notes in a sequence. In the end, the best song that we could extract from the LSTM was the included in the *Soundcloud* repository, which is deceiving compared to the outputs of the papers we found.

3.1.2 Variational Autoencoder

The idea behind a Variational Autoencoder is fairly simple: it has the same two parts that every autoencoder has, an encoder and a decoder, and it has as objective minimizing the error between the input and the output. Nevertheless, we introduce regularization by embedding all points as a probability distribution in the latent space. Doing so, the process to generate a new sample is quite straightforward: a random sample from a Gaussian distribution is computed and the decoder tries to reconstruct it.

It is also worth mentioning that the loss of a Variational Autoencoder is made of two terms: the reconstruction error (computed using Mean Square Error in our case) and a term related to regularization that checks if the distribution follows a Gaussian distribution, which is computed using the Kullback-Leibler divergence. The outputs that came out of this model were very poor. We observed that while MSE went down fairly nice, the KL values did not stabilize, which could be the reason to this problematic.

[3] https://soundcloud.com/pau-magarino-llavero/sets/music-generation-with-dl/s-L79mrISIDRO

[4] https://github.com/paumll/radio_gaga

3.1.3 GAN

The last model we made with the basic approach to MIDI preprocessing was a Generative Adversarial Network. This architecture is divided into two main parts: the generator and the discriminator. The generator creates possible new songs based on random noise. This noise is passed through a pair of linear layers in order to increment its size and then we use several transposed convolution layers until we end up with a new matrix with the same dimensions as the songs from our train set. Then comes the discriminator, which decides whether an input is either a real sample or a generated one using convolutional and linear layers.

This idea was very promising at first because we knew that MidiNet (the model we used as baseline during the next step of the project) also used a GAN architecture, but the results we got from this last model were really bad: there was no real correlation between notes and all fluctuations in the melody were abrupt. Moreover, there were a lot of hyperparameters to tweak in this model, such as the learning rates of both models, the amount of convolutional layers, the types of subsampling, etc. After a fair amount of tries, we ended up discarding this option as a model, because the results were not even close to the expected.

3.2 Main findings

Given the results we obtained with the three proposed models, we decided to take a change of direction for the rest of the project. Creating our own models seemed to lead to a deadlock, so we thought that taking a developed and tested model such as MidiNet would help us to achieve our goals.

Although this decision gave rise to several problems (more than expected), we are sure that we took the right decision, since it allowed us to develop higher-quality models with meaningful outputs.

4 MidiNet

In this section we describe a novel CNN-GAN-based model specifically designed for MIDI generation, which actually uses a very effective conditional mechanism: MidiNet. This network promises more realistic, pleasing and interesting results overall.

4.1 Overview

MidiNet arises from the need to improve symbolic music generation models with deep learning. The model can generate melodies either from scratch, by following a chord sequence or by conditioning the melody of previous notes.

The basics of its architecture are those of a GAN. The goal of the generator is to transform the inputted random noises into a 2D score-like representation, that "appears" to be from real MIDI. As opposed to a regular GAN, in order to take into account the temporal dependencies across different notes that articulate the melody, the novel conditional mechanism is used: training an additional model called *conditioner CNN*. This new part of the network allows to incorporate information from previous note beats to intermediate layers of the generator. That is to say, our model can "look back" previous beats without a recurrent unit as used in RNNs.

4.2 Dataset and Preprocessing

For this model, we had to work with a different dataset: The Lead Sheet Dataset[5]. It consists of 11.380 songs represented in *XML*, *json* and *npy* (Numpy arrays) files. This dataset collects information from TheoryTab[6], a website oriented to music students with songs represented in pianorolls. Unlike MAESTRO, (where the MIDI files only represented the beginning and ending time for each note), this dataset separates songs into melody and chords. This implementation helps us to easily feed the model (in MAESTRO it was practically impossible to extract both melody and chords separately).

As we have already stated in the previous section, MidiNet works with both melody and chords, and it has its own preprocessing code. Notwithstanding, the code provided only extracts melody, and we were not able to execute the model. As a result, we had to do chord extraction from the dataset (specifically, from the *json* files, which yielded more information than *XML* files).

The *json* files used to obtain them were the *symbol_nokey* type, which include all chords in the same key (C). Then, we looked at the variables *root* (which gives us the name of the chord) and *quality* (which tells us whether the chord is major or minor). To simplify, we only consider the first chord of each bar. The original paper of MidiNet implies that they are doing it too, but we do not know for sure how they did it (as there is no implementation).

Once we have the chord and its type, we encode them into a vector of size 13. The first 12 positions represent a one-hot vector (denoting the root), and the 13th is a flag which indicates whether the chord is major or minor.

In the end, the addition of the chords takes the preprocessing to a whole new level, because we are feeding the model with two types of information. The melody part has notes that are played one after the other and it typically has rapid changes, and the chord part has less changes and means more than one note per element.

4.3 Architecture details

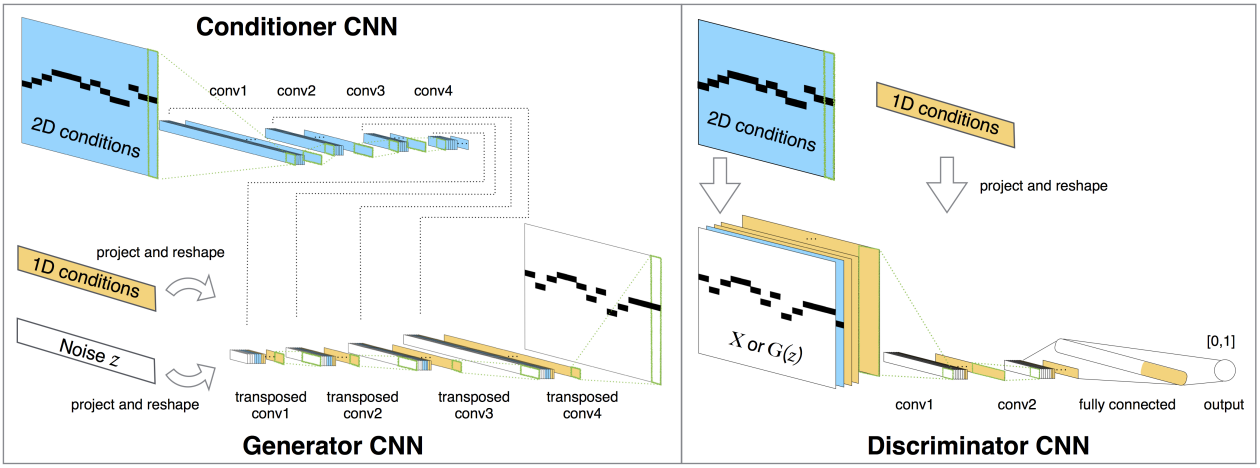

Figure 1: Architecture diagram extracted from the original paper [14].

[5] https://github.com/wayne391/lead-sheet-dataset
[6] https://www.hooktheory.com/theorytab

Going into the details, the input for the generator G is a vector of random noise $\mathbf{z} \in \mathbb{R}^l$, whereas the output of G is an h-by-w matrix $\widehat{\mathbf{X}} = G(\mathbf{z})$ that appears to be real to D, as typical in GANs. The discriminator is a typical CNN with few convolutional layers followed by fully-connected layers, optimized with a cross-entropy loss function, such that the output of D is close to 1 for real data (i.e. $\mathbf{X}$) and 0 for those generated (i.e. $G(\mathbf{z})$). Besides that, the generator aims to make the output of D close to 1 for the generated data. G transforms a vector $\mathbf{z}$ into a matrix $\widehat{\mathbf{X}}$ by using a few fully connected layers first, and then a few transposed convolution layers that upsample smaller matrices into larger ones.

As the generation result of the GAN is an h-by-w matrix of notes and time steps, the melody of a previous beat can be represented as another matrix of the same shape (2D) and can be used to condition the generation of the current beat. We can have multiple matrices to learn from several previous tempos. And, as the conditioner and the generator use exactly the same shapes in convolutional filters, they can be concatenated and trained simultaneously, by sharing the same gradients.

In addition, as a technique often used in GAN-based image generation, some prior knowledge can also be encoded with a 1D vector by reshaping and projecting it to the different layers of G and D, as an additional input. This information is crucial for chords, as they can be easily encoded as a vector of 13 dimensions.

4.4 Errors corrected

Regarding our work on MidiNet, we would like to highlight the enormous effort put into simply starting to make the model work. Firstly, we have to draw from the premise that the MidiNet implementation of the authors was originally made on TensorFlow [13], and the actual code on which we have based our assignment was a translation of it to PyTorch [6] (a framework in which we have a broader knowledge to work), but it was incomplete, improperly typed and somehow lacked several indispensable parts. So, taking into account all the time invested in choosing an approach to music generation, and exhaustively testing without success the three previous models introduced in this report, we can say that most of the weight of this project has fallen on improving MidiNet.

In the following, we describe some of the challenges we have faced and, more concretely, the bug fixes made to specifically address each of them. For instance, besides the lack of a part of code to extract chords from data (which has already been explained in the Dataset and Processing section), we found errors in the transposition of the melody to other keys, since the original key of the input was wrongly received and more samples were returned than those entered. This resulted into many duplicated samples, while others disappeared. Not to mention the errors in accessing the notes on the generator part, as they tried to access the wrong dimension of the data array and ended up generating matrix blocks. In a similar fashion, the demo script for the user to put the application arguments was missing type consistency on many variables and had to be redefined. In terms of more serious problems regarding the implementation and the execution flow, we emphasize the problem we had with *inplace* operations declared in the training part when executing in the PyTorch version 1.5.0. We do not know exactly why this happened, nor we are sure if it was related to the loss function or the model itself.

Next, the *batch normalization* step was declared in a non-optimal way throughout the code, defined in each of the *forward* functions several times (instead of just once) and, additionally, they got rid of it when executing on evaluation mode. The modification of this change, along with the

previous ones, really helped to understand and simplify the network structure.

Finally, we want to point out an important aspect of the MidiNet model regarding its astonishing results in the original paper. After investigating, we intuited that the output of the PyTorch implementation seemed to be post-processed, to our best knowledge, and it discredited the quality of the actual results of the network (and it is a thing to keep in mind). Furthermore, already thinking about the future evaluation of the obtained results, we noticed we had no data with which MidiNet had trained (nor of the chords, for granted), so that made more difficult to perform an accurate comparison with the original model. From the above, we decided to somehow evaluate the results of our improvements on MidiNet against our own baseline model, as we describe in the following sections.

5 Proposed models

In this section, we will explain all modifications we performed on MidiNet architecture. The resulting MIDI files are uploaded in the following link[7], and each model will be accompanied by the file name corresponding to the sample generated.

5.1 MidiNet Baseline

Once we corrected all errors, the first model we executed was MidiNet (as it was implemented originally; no changes were made), and we took it as baseline.

As we can hear, this result is far from the one yielded by MidiNet[8]. This is because we do not know how MidiNet model was trained, and the data it was provided with. For this reason, we cannot establish comparisons between our model and MidiNet, and we will compare the models generated with our baseline.

5.2 Embeddings

This implementation consisted in transforming each chord into an embedding of size 100. In order to do that, we changed the format of each of them from a vector into an integer (from 0 to 24, as 25 were the possible combinations for the chords: 12 major chords, 12 minor chords and the silence). Then, two different embedding layers were used in the discriminator and the generator.

Due to its high computational cost, we were not able to take advantages of all possibilities given by this model (it took 10 hours to run 20 epochs). As we find it quite useful, we believe that it should be more explored when the computational resources yield the expected performance.

From this moment onwards, all models will be executed with the new implementation of Batch Normalization. It consists on defining the normalization only once and use it in *eval* mode too. The Batch Normalization of MidiNet, as stated above, was neither properly implemented nor used (at least in the PyTorch implementation).

[7] https://soundcloud.com/pau-magarino-llavero/sets/music-generation-with-dl/s-L79mrISIDRO

[8] https://soundcloud.com/vgtsv6jf5fwq/model3?in=vgtsv6jf5fwq/sets/midinet-samples

5.3 Spectral normalization

This novel method proposed at [9] renormalizes the weights of the layers which it is applied onto, resulting in the mitigation of gradient explosion problems. Frequently, this type of normalization is applied only in the discriminator layers. In our case, we tried two different approaches: spectral normalization on the discriminator, and also on both generator and discriminator.

As it can be found in many papers, applying this normalization in the generator did not seem to improve the previous results. However, the other approach (only the discriminator layers were normalized) provided an improvement.

5.4 Previous data to the discriminator

As we explained in the architecture, MidiNet works with past data in order to refine the predictions. However, the discriminator only takes into account the melody and the chord (not the previous data). Aiming to have a deeper correlation between notes during time, this modification consisted of taking the previous data and feed the discriminator with it.

5.5 MLP discriminator

This model consisted in adding a MLP (Multi-Layer Perceptron) on the last layers of the discriminator. This decision was based on the fact that the generator seemed to fool the discriminator quite rapid. With this modification, we expected to increase the performance of the discriminator thus forcing the generator to create more realistic samples.

Regarding the results of this model, we expected a noticeable improvement on the samples generated. However, we did not foresee such an awful result: the bars generated were exactly the same as the first bar (which is actually a real bar from the test set), therefore the generator was only taking into account the previous data. Yet it may sound strange, we could explain this behavior based on the fact that the previous bar is actually real, thus fooling the discriminator constantly.

5.6 Embeddings with previous data plus spectral normalization

The last model we tried was a mix between the spectral normalization model and the embeddings one. Given that these two models provided the best results so far, we thought that combining them would provide an improvement. In addition to that, we still had in mind the idea that the generator was fooling the discriminator too easily.

The changes with respect to the rest of the models were the following:

- Reduce the number of times that the chords were concatenated with the input on the discriminator, so as to force it to pay attention to the melody, which is the part that the model is creating (the chords are provided by the data).
- As we already did in the embeddings model, use an embedding to transform the chords.
- In the generator, delete the concatenations between the melody and the chords. Instead, we concatenated the chords with the previous data and passed both through a series of convolutions separated from the melody (Conditioner Network).

While this changes were really promising, unfortunately we did not manage to find an equilib-

rium between the generator and the discriminator. Now, as opposed with the previous models, the discriminator seemed to be so much stronger than the generator.

It is also worth mentioning that we tried with different learning rates in order to find that desired balance. The same learning rate for both models did not work at all, even less a higher learning rate for discriminator and a lower one for the generator. In both cases the loss of the generator increased without stopping.

However, when the learning rate of the generator was higher than the one of the discriminator, the loss during the first epochs seemed acceptable, but then the discriminator started to learn and we were back to the bad results.

Although we could not find a good hyperparameter tuning for this model (or simply the balance between discriminator and generator was not good enough), we think that it still has room for improvement, and we would have been working on it if we had more available time.

6 Results comparison

In order to compare the results of each model, we have decided not to include the details about the loss of each of them. While in some other architectures different from GAN the loss could be a useful indicator of the performance of the model (specially in other domain problems), we think that in this problem it would not provide an insight.

Firstly, we have found that the loss always converges to the same values. As we know, the discriminator should not know how to distinguish between real and fake samples. This makes the probabilities with which the discriminator predicts a sample as real or fake converge to 0'5 in both cases. Consequently, the loss converges always to the same value.

Secondly, the quality of the samples generated does not seem to be highly correlated with the loss. For example, some models with slightly higher loss than other models generated better samples and vice-versa. As a matter of fact, there could be some cases where the generator easily fools the discriminator (so the loss would rapidly converge), whereas it could also happen that a stronger discriminator would distinguish better between real and fake samples, so maybe the loss could be higher. Despite that, the generated samples in the latter case could provide an improvement in comparison to the first one.

In the following sections, we will compare the outputs of each model. For better understanding, we encourage the reader to open the *Soundcloud* link provided before.

6.1 Baseline

We cannot conclude whether this result is satisfactory or not, as it is the baseline without any modification. However, we expect that our models will outperform this audio, because there are loads of discordant notes and we find it unpleasantly rapid.

6.2 Embedding

At first glance, there seems to be a slight improvement with this model. Regarding the speed, samples 1 and 3 reduce the distastefully fast feeling we talked before, and the notes make more sense. However, Sample 2 does not say much, with only having notes in the first and last bars. In overall, we can say that this model performs better than the baseline. It will not win a Grammy, but we can start to feel the essence of a song.

6.3 Spectral Normalization

Implementing the Spectral normalization in both generator and discriminator does not produce good results. One may think that it does not sound that bad, but this is due to the lack of melody notes. Consequently, most of the time we only hear chords, which sound appealing because they are real (not generated by our model).

Regarding the model with Spectral normalization only in the discriminator, we find more meaningful results. The most remarkable aspect of these samples is that we can appreciate a set of notes that repeat themselves along the song. This is a feature that most songs chorus have, and this model has been able to extract and represent it in some way in the results generated.

6.4 Previous data to the discriminator

Some interesting conclusions can be drawn more from these samples. They do not sound appealing at the first hearing, but they do have identity and uniqueness. We can find the repeated patterns we talked about, but there are more variations around them (it is not always the same notes played the same way). It looks like a more thoughtful melody.

6.5 MLP discriminator

This model does not yield good results. Up until now, we have found several pieces with pattern repetition, but not in a bad way: they made sense. However, these 3 samples are made of a few notes that are repeated along the bars, without any meaning or beauty.

6.6 Embeddings with previous data plus spectral normalization

As we can hear, these samples are horrible. They have nothing to do with the others. Apart from the fact that the melody is heavily dissonant, they appear low-pitched and high-pitched notes that make the song even worse. Having all the modifications explained before, we expected this model to outperform the rest and yield a quality song, but nothing could be further from the truth.

7 Conclusions

To begin with, we would like to make a reflection about our first approach to this project. We firstly believed that it could be suitable to create a music generation model from scratch. Taking into account the obtained results, the difficulty of implementing a new architecture has been proven. However, this type of problems will surely be helpful for future projects.

Another interesting point to expound on is the liberty we had while developing this project. During the course of this degree we have participated in several projects with strict guidelines to follow. This has been one of the first assignments where we have had absolute liberty in order to solve a problem, which is music generation in our case. While it has had its drawbacks (as we have already stated in the previous paragraph), we think that these are the type of projects that better prepare ourselves for our professional future.

Regarding the problem of music generation, we have also faced a new type of challenge, quite different with respect to the ones we are used to. In this case, there was almost no metric different from human evaluation that could help us to evaluate every model. While it may be true that the loss of the models helped us to discard some of them, when they converged, we had to compare them listening to the samples generated, leading to a subjective evaluation, which may not be the ideal metric on the field of Machine Learning.

As for the models we have created, we would like to emphasize the difficulties we have found in order to train GANs. The balance that both generator and discriminator have to share, the number of different hyperparameters (and its sensibility) or the specific problems that this type of architectures usually have (mode collapse, vanishing gradients, failure to convergence...) are some of the problems we had to deal with. These obstacles have surprised us since we did not think that training a GAN would be that hard. However, the magnificent results (or creations) that this architecture is able to provide makes it really worth it to use it.

Another important issue to include in these conclusions is the problems found with Midinet implementation. Since we have already stated which errors we have solved, we would only like to stress the fact that a repository linked from the authors of the original paper should be complete or, at least, it should indicate that it is an incomplete implementation. In addition to that, neither this implementation nor the original one provided the dataset in order to reproduce the results.

Last but not least, it is necessary to stop and think about the results obtained. Obviously, we have not provided a meaningful improvement with respect to Midinet. Even so, we consider that the results are quite positive considering the difficulty and subjectivity of music generation.

8 Further work

Despite being satisfied with the results of this assignment, we also thought about other paths to follow and improvements to be implemented in the future.

The first one is to increase the number of chords taken by bar up to 4 (as it is the number of beats per bar), which would bring us a step closer to real songs. In addition, we also considered taking real sequences of chords in the sample generator. As we know, the sample generator takes real chords to generate MIDI files. In order to select them, it takes random indexes from the dataset. As all chords and melody blocks have been shuffled, it is highly probable that chords from different keys are selected one next to the other when generating a sample.

To fix this problematic, we could store chords into song batches (instead of random bar blocks). Then, before augmenting the data and separating our dataset into train and test, we would store apart a set of bar blocks to be used in the generation process. This implementation ensures all chords will be in the same key and they will be highly correlated (as they come from the same set of songs). The models executed after this modification will generate songs which may make more sense.

Another improvement to be considered is to work with larger matrices. This implementation would imply more preprocessing and more modifications (when predicting more notes, it may lose quality, and we would have to spend more time developing a much more sophisticated model). In spite of that, we think it would produce songs which may make more sense. With a further due date, we would definitely try this idea, but currently it is out of the scope of this project.

Relating to this, an improvement in our technical equipment would make us focus more on the embeddings model. We believe that it is a very interesting approach, which can produce better results than the ones obtained in this assignment with the proper tuning and training time.

Lastly, we want to propose a change of perspective. During this project, we have worked with *XML* and *json* files provided by the dataset. It was a sort of ground rule. However, we wondered which results would yield a model trained with spectrograms or audio files (in turn implementing an end-to-end architecture). Nowadays, these strategies are considered state-of-the-art.

References

[1] Martin Arjovsky and Léon Bottou. *Towards Principled Methods for Training Generative Adversarial Networks*. 2017. arXiv: 1701.04862 [stat.ML].

[2] C.J. Bayron. *repo: PyTorch implementation of C-RNN-GAN for Music Generation*. Nov. 2019. URL: https://github.com/cjbayron/c-rnn-gan.pytorch.

[3] Jean-Pierre Briot. *From Artificial Neural Networks to Deep Learning for Music Generation – History, Concepts and Trends*. 2020. arXiv: 2004.03586 [eess.AS].

[4] Soumith Chintala. *repo: ganhacks*. Mar. 2020. URL: https://github.com/soumith/ganhacks.

[5] Martin Heusel et al. *GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium*. 2017. arXiv: 1706.08500 [cs.LG].

[6] Anna Hung. *repo: MidiNet-by-pytorch: Implement MidiNet by PyTorch 0.4.1*. Mar. 2019. URL: https://github.com/annahung31/MidiNet-by-pytorch.

[7] Abraham Khan. *Generating Pokemon-Inspired Music from Neural Networks*. 2018. URL: https://towardsdatascience.com/generating-pokemon-inspired-music-from-neural-networks-bc240014132.

[8] Sanidhya Mangal, Rahul Modak, and Poorva Joshi. *LSTM Based Music Generation System*. 2019. arXiv: 1908.01080 [cs.SD].

[9] Takeru Miyato et al. *Spectral Normalization for Generative Adversarial Networks*. 2018. arXiv: 1802.05957 [cs.LG].

[10] Marco Pasini. *10 Lessons I Learned Training GANs for one Year*. 2019. URL: https://towardsdatascience.com/10-lessons-i-learned-training-generative-adversarial-networks-gans-for-a-year-c9071159628.

[11] Alec Radford, Luke Metz, and Soumith Chintala. *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*. 2015. arXiv: 1511.06434 [cs.LG].

[12] Haryo Akbarianto Wibowo. *Generate Piano Instrumental Music by Using Deep Learning*. 2019. URL: https://towardsdatascience.com/generate-piano-instrumental-music-by-using-deep-learning-80ac35cdbd2e.

[13] Li-Chia Yang. *repo: MidiNet: A Convolutional Generative Adversarial Network for Symbolic-domain Music Generation*. Oct. 2018. URL: https://github.com/RichardYang40148/MidiNet/tree/master/v1.

[14] Li-Chia Yang, Szu-Yu Chou, and Yi-Hsuan Yang. *MidiNet: A Convolutional Generative Adversarial Network for Symbolic-domain Music Generation*. 2017. arXiv: 1703.10847 [cs.SD].