# Generate Chopin-style music - A Recurrent Neural Network and Convolutional Generative Adversarial Network approach

**Article** · June 2022

CITATIONS
0

READS
254

**1 author:**

Giulio Federico
Università di Pisa
**3** PUBLICATIONS **0** CITATIONS

SEE PROFILE

# Generate Chopin-style music

A Recurrent Neural Network and Convolutional Generative Adversarial Network approach

Giulio Federico

June 2022

University of Pisa
MSc in Artificial Intelligence and Data Engineering
Computational Intelligence and Deep Learning

# Contents

# 1 Scope of work

Fryderyk Franciszek Szopen, also known by the Frenchized name of Frédéric François Chopin, was a Polish composer and pianist. Chopin wrote almost all compositions for piano, and in his 39 years of life he was considered the "poetic genius" of the Romantic period. The purpose of this work is **to understand what the style of the "piano poet" is** and **to generate some compositions that Chopin could also have written** if he hadn't died of tuberculosis so prematurely at 39 years of age.

After a deep study on the subject, I decided to **implement 3 architectures** capable of generating music:

1. The first is a very powerful architecture capable of generating polyphonic music by capturing Chopin's style very well, even if *it is only able to generate the next note or chord*, not being able to dynamically generate new durations or position the notes with respect to the others, which therefore will have a fixed duration (0.5 seconds) and a fixed offset (i.e. each note is positioned 0.25 seconds after the previous one).

2. **The second architecture is my proposal**, therefore a new architecture even if it is a mix of various solutions that i read on the papers, which tries with great success to *overcome the two limits of the previous architecture*. My **trident architecture** (i called my architecture in this way because of its shape) is able not only to predict the next note, but also the next duration and offset thus fully capturing Chopin's style. However, it is long to train and needs a lot of data to better generalize.

3. The third architecture takes a leap into what is the biggest search trend today, namely **generative adversarial networks (GAN)**. The created GAN, or rather the **C-GAN (Convolutional GAN)** is a basic from scratch implementation of the *MidiNet*.

# 2 Music theory

This section is intended to describe the basic music theory that you need to know for the implementation of the Chopin style AI music generator, without going into too much detail since I am not a musician, I cannot play an instrument and therefore I don't want to teach anyone. But for a novice reader like I was, such an introduction to basic music theory can help understand what follows.
A **staff** or **stave** is a set of 5 parallel lines. The spaces between two lines are also important. The lines, as well as the spaces, are numbered from bottom to top:

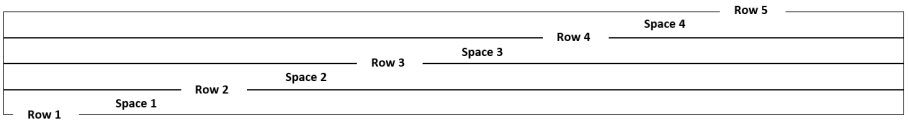


Figure 1: lines and spaces of a staff

The **musical notes** go into the lines and spaces. Depending on where these will be positioned we will have the Italian musical notes (the relative English ones in parentheses): DO (C), RE (D), MI (E), FA (F), SOL (G), LA (A), SI (B):

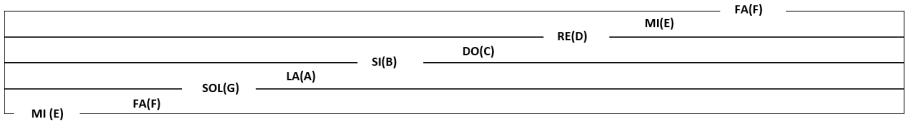


Figure 2: notes on a staff

In reality, this "scale" is decided by the so-called **musical key**. In fact, a single staff cannot contain all the sounds from the lowest to the highest. Depending on the key, the notes will then be positioned differently on the staff. The key of our interest (and the one most used) is the *treble clef*. This allows us to understand where the G is positioned, and from there following will derive the positions of all the other notes:

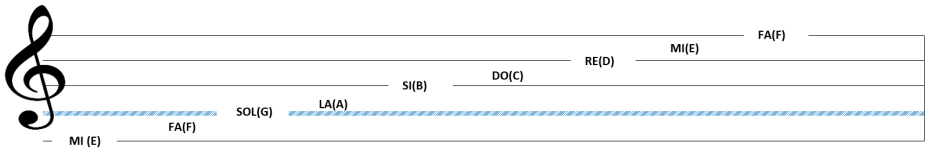


Figure 3: treble clef

Depending on where we place the notes we will have low or high sounds.

The higher the height (**pitch**) where we place the note, the more acute this note will appear, while if we place it lower we will get a lower sound.

There are several notes to place on a particular line or space and each has its own particular meaning. Let's imagine a man snapping his fingers every second for 4 seconds:

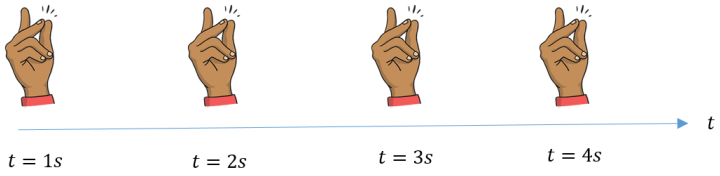


Figure 4: a man snapping his fingers

The first note is the *semibreve*:

Figure 5: a semibreve note

It has a duration of 4/4. This means that in that 4 second timeline, its duration must be 4 out of 4 seconds:

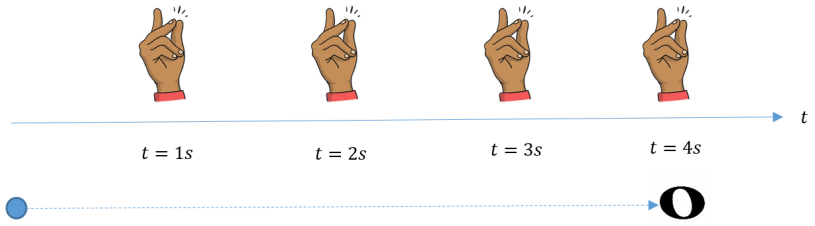


Figure 6: the duration of the semibreve note

Another note is the *minim*:

Figure 7: a minim note

whose duration is 2/4. This means that in that 4 second timeline, its duration must be 2 out of 4 seconds:

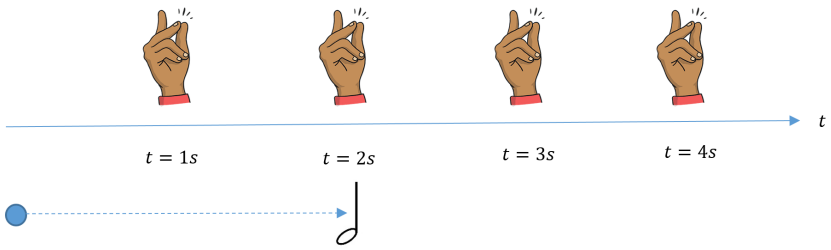


Figure 8: the duration of the minim note

This is the concept of duration associated with notes. The 4/4 **time signature** is the most common, but there are others as well. For the sake of completeness, I report the remaining notes with the relative durations:


Figure 9: the main other remaining musical notes

Each type of duration (eg 4/4, 2/4,..., 1/64) has associated not only with a note but also a pause called **rest**. There will therefore be rests that will last 4/4, rests that will last 2/4 and so on up to 1/64:

Figure 10: the main rests

Notes and rests will be inserted in the lines or spaces to indicate respectively to repeat that note and / or that rest for a certain amount of time indicated by the respective duration. At the temporal level, the notes and the rests are played (and therefore read) from left to right; at the end of the duration of each note or rest, the next is played.

A note is a single sound, while a **chord** is a group of notes played simultaneously. If with the notes we were able to express the duration and tone of a sound thus contributing to the *melody*, with the chords we are able to give

harmonic structure to the melody. In practice, the chords "accompany" the melody creating greater *harmony*. We speak of a chord when at least 3 sounds are played simultaneously. A 3-note chord is called a *triad*, a four-note chord is called *tetrad* and so on. The interval that separates one note from another must be a third interval.
What is a *third interval*? The white keys of the piano, with the adjacent black keys, create a distance called a *semitone*. The semitone is the smallest distance between two piano keys; a *tone*, on the other hand, is the distance between two white keys which is therefore equal to two semitones.

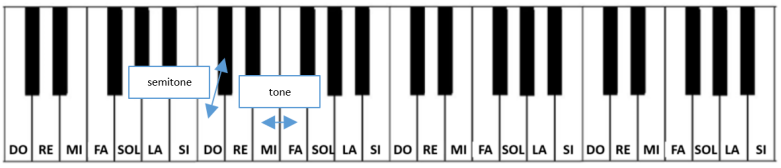


Figure 11: semitone and tone

An interval of a third is in fact an interval that consists of 4 semitones, therefore two tones. However, it can be greater or less. For example C-D-E has a *major third interval* as the distance between C and E is made up of 4 semitones. On the other hand, E-F-G is a *minor third interval* since the distance between E and F is a tone, but the distance between F and G is a semitone since the black key is missing. There are dozens and dozens of chords. Some of them express positive feelings, others of melancholy and sadness.

We have seen that if we associate a particular symbol with certain notes we are somehow indicating its duration. Below is the complete figure::

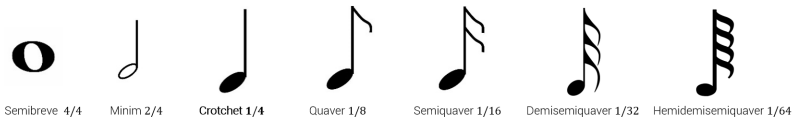


Figure 12: notes with their duration

The Crotchet (highlighted in bold) is the symbol taken as a reference. We will call its durations *1 beat*. Therefore, according to this new quantity, the whole becomes:


Figure 13: notes with their durations using 1 beat as the unit of measure

As mentioned, 4/4 is not the only type of time signature. To indicate the type of time signature we have a symbol (a sort of fraction) which is placed after the musical key:

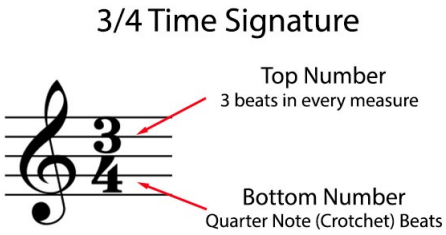


Figure 14: 3/4 time signature also called waltz time

The "numerator" (the top digit) indicates how many beats there are per *measure*, while the "denominator" (the bottom digit) indicates which note type is considered a beat. A **measure** is a section that contains a number n of notes such that the sum of their durations/beats equals the top number of the time signature. The separation between two measures is indicated with a vertical bar called **bar**.

To get the concept clearer, let's suppose again a 4/4 time signature like the one in figure 12:

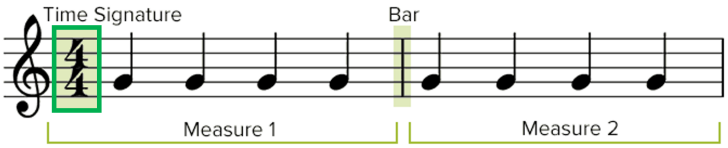


Figure 15: 4/4 time signature example

Being the "denominator" 4 means that the Crotchet note is treated as a beat, while the "numerator" 4 indicates that there will be a total of 4 beats in a measure. Obviously the notes can be different from the Crotchets; for example we could use two Minims instead of two Crotchets since their beats is equal to 2 (so we will have $2 + 2 = 4$) .

Time signatures can also be found in different parts of the staff, signaling a change of meter. In fact, imagine you want to use two different meters within your song, the first with four pulses per measure, the second (in the second half of the composition), with three pulses so that the second half is faster. This technique is called **metric modulation**.

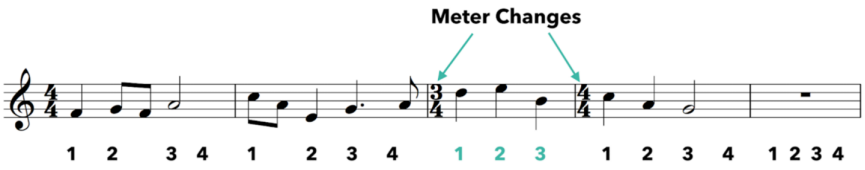


Figure 16: an example of metric modulation

Finally, a last very important concept is that of **Tempo**. This is expressed as a *word* (eg Allegro) or in terms of *Beats Per Minute (BPM)*. For example Allegro is equivalent to 144 BPM. We will see the BPM in an online simulator that we will use to play the .mid files generated by the network:

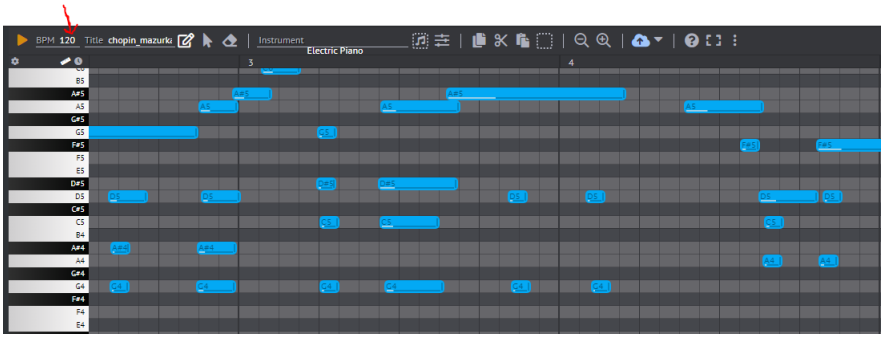


Figure 17: an example of Tempo ( 120 BPM )

# 3 Scientific pitch notation

How can we encode a certain note with its particular pitch? The American System for annotating pitches, called *Scientific Pitch Notation (SPN)* has established that the encoding must be a pair as follows:

$$(note, octave)$$

For example in the piano, referring to the note C (DO) we can find different Cs, each with its own particular pitch:

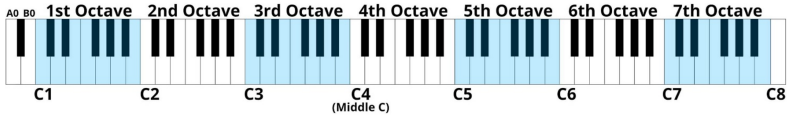


Figure 18: the octaves of C

If we consider the notes that go from the first C (C1) to the second C (C2) we get the first octave. Consequently the second octave will be the section between C2 and C3, and so on. The first C (C1) would be the lowest C, while the last C (C8) the most acute. To indicate this pitch gradient, the octave to which it belongs is linked to the note symbol (in this case C). For example C1 indicates the lowest C on the piano keyboard.

The same reasoning is done with the other notes. For example, the section from A1 to A2 will be the first octave of note A, and so on. We can then convert the notes on the staff according to the SPN encoding:

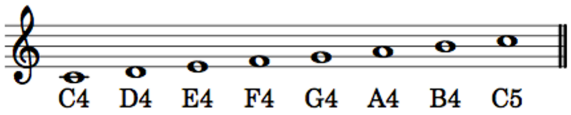


Figure 19: encoding of the notes on staff according to the SPN encoding

# 4 MIDI notation

MIDI stands for *Musical Instrument Digital Interface* and is an industry standard for passing musical performance information among electronic musical instruments and computers.
Many of the audio equipment we know of, such as CD players and radios, transmit and receive sound in the form of audio:

Figure 20: audio wave

MIDI instruments are not like CD players or radios, they receive information such as "Play a Middle C note softly using the clarinet sound for a duration of a quarte note at the current time" where "Middle C" is just another name for refer to C4, and so it is used to store message instructions which contain note pitches, their pressure/volume, velocity, when they begin and when they end and so on.
A representation known as *Piano Roll* allows us to show a kind of staff in which each note is a rectangle and its duration is the length of the rectangle itself:

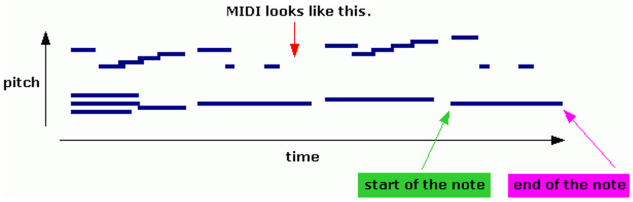


Figure 21: Piano Roll

The notes themselves in the SPN encoding are mapped to integers. The SPN notes are mapped into integer numbers ranging from 0 to 127. For example, Middle C, or C4, is mapped with the integer 60. An example is shown below:

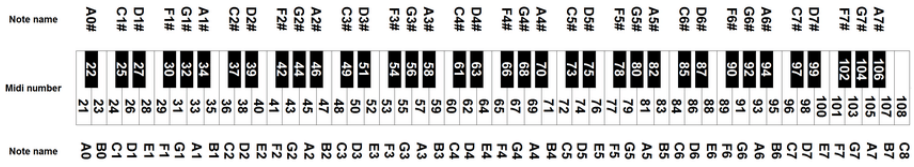


Figure 22: an example of the MIDI encoding

The example of the staff and the relative notes seen in figure 19, assumes the following encoding:

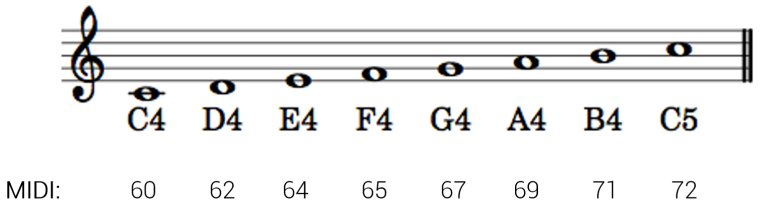


Figure 23: encoding of the notes using MIDI

# 5 Symbolic Music Representation and Network Implementation

Once we understand MIDI, we just have to figure out how to extract from MIDI the information we need to build the sequence of data that we will use to train the network. Depending on how this is built, the result will be a network more or less capable of generating music faithful to Chopin's style. I will present only 3 of the possible methods, *two of which have been implemented in this project*, analyzing the pros and cons of each. In addition to these, there are several others used then depending on the case.

## 5.1 The conditional probability

Ideally there is a $p_{data}$ distribution that describes what is the probability of playing a certain note.

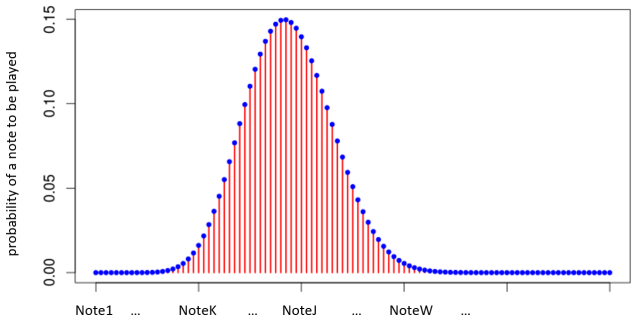


Figure 24: example of a simple distribution of notes

This distribution can be simple or very complex indeed. The one in figure 24 is just an example. Now, the problem is that we do not know $p_{data}$, but we can be able to approximate this distribution with a $\tilde{p}_{data}$ by constructing it through the observations available to us, that is the compositions themselves. This distribution is not so difficult to obtain and so far from the real one. The real problem is finding the distribution of the notes when some of them have already been played. This, in mathematical jargon, is called **conditional probability**:

$$p(note_2|note_1)$$

This distribution too could be found without much difficulty as long as we are only interested in two notes. In our case we need to be able to approximate a much more complex distribution that can model the probability of playing a note after N generic notes have been played previously:

$$p(note_{n+1}) = p(note_{n+1}|note_1, note_2, ..., note_n)$$

This is where the going gets tough, because those N notes previously played could be any combination of all possible ones. One assumption we could make is to think that there is no dependence between a note and the previous one played. This is called the *Naive Bayes assumption* and would lead us to this formula:

$$p(note_{n+1}) = \prod_{i=1}^{N} p(note_i)$$

This assumption is too simplifying. Thinking that the probability of playing a note now does not depend on the notes played previously is a simplification that we cannot afford for obvious reasons. Fortunately, we will see it in paragraph 6 a recurrent neural network equipped with memory cells such as LSTMs that be able to model this distribution. But here too we have another problem. The network we will build will be able to memorize only K of the possible previous notes to build the distribution of the next note. This means that if we train the network with a time window of 100 we could certainly get a really good approximation $\tilde{p}_{data}$ of the 101-th note after the previous 100 notes have been played, but this approximate distribution will start to become less and less precise if we wanted to predict notes taking into account more than 100 previous notes. For this reason, to predict the next one, we will always take the previous 100. Often the time window is chosen of such a length as to make it possible and within a reasonable time to train the network, and more so as to capture the **trend**. For example, figure 25 shows the time series relating to the number of passengers of the airlines.

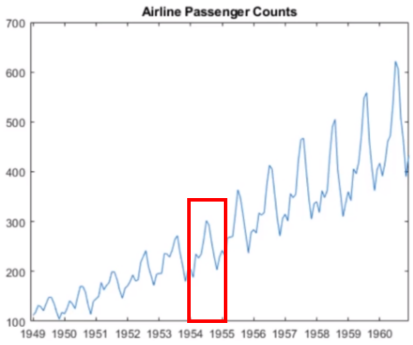


Figure 25: an example of trend

You can see how the trend repeats itself over time. This is the ideal case where you can choose the time window of that length. In the case of music, this

trend is almost non-existent, or if it exists for one composition, it will not exist for all the others. For this reason the choice of the length of the time window is heuristic and equal to 100 for the first architecture and 500 for the Trident architecture.

## 5.2 Overfitting is not your enemy

I wanted to dedicate a paragraph on the overfitting topic. We often see overfitting as a problem, something to be avoided. This is almost always the case. For this reason we use validation sets that can "monitor" the training and stop it in case the network is learning really well on the training set, but fails to generalize.
I want to clarify that what I am about to say is my consideration and my approach to the problem of generating music. Several states of the art have avoided overfitting, but I have done the opposite. In the two architectures implemented, at the beginning of the work, I used a validation set. Although this contained compositions belonging to categories (eg Mazurkas) also present in the training set, given the very different nature of each composition from the others, even belonging to the same category, I noticed that the network (in several attempts made), even with a very high patience, could not exceed 3% accuracy. This was one of the reasons that led me to intentionally choose to overfit the network.
But the real motivation lies in the fact that, in my humble opinion, when we talk about discovering the "style of a pianist" we are just talking about discovering the "conditional probability distribution" that will be better modeled the more the network goes into overfitting.
But then we might ask, *how can we generate new music if the network overfits*? Like the genetic operator of Russian roulette, what we will do, once we have the distribution, is a probabilistic sampling of predictions to vary the outcome. Therefore we will never always take the note with the highest probability from the distribution (otherwise we will get compositions identical to the originals), rather we will do a *probabilistic sampling*, and therefore each note has the probability of being chosen as the next generated note equal to the probability indicated by the distribution.

## 5.3 A method for monophonic music (not implemented )

Let's imagine we have a 4/4 time signature like the one in figure 26.

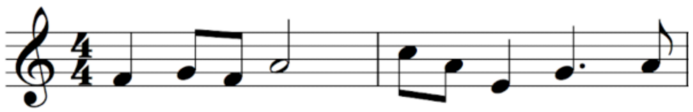

Figure 26: example of 4/4 time signature

We focus our attention on the first measure. The following reasoning will

apply to all other measures. Since the time signature is 4/4 we will have a total of 4 beats for each measure that we can currently represent in an array (Figure 27).

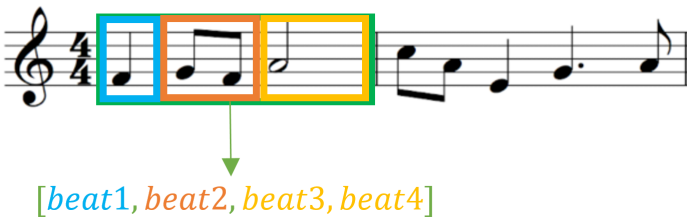


Figure 27: array of 4 beats in each measure

The array has the last two elements with the same color as the Minim note occupies two bits. Therefore in the first beat only the Crotchet is contained as it has a duration of 1 beat; in the second beat there are two Quavers as their duration is equal to 1/2 beat; finally the last two beats, as already said, are entirely occupied by the Minim as it lasts 1 beat. Since the Crotchet is taken as a reference, and its duration was 1/4, it is as if the 4 beats per measure became 16 temporal steps that each note, depending on its bit, can fill up to a certain number of temporal slots.

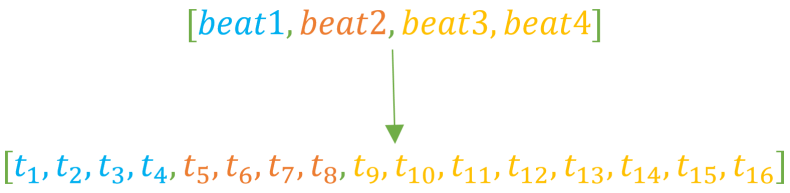


Figure 28: beats to time slot conversion

In each time slot $t_i$ we will insert 3 possible values:

1. The MIDI encoding of the note.
2. The symbol "_" if that instant in time is still occupied by the previous note or rest.
3. The "r" symbol if that instant in time is occupied by a pause.

In our case the first beat $t_1, t_2, t_3, t_4 \ldots$ is composed only by Crotchet on the F line (FA) whose MIDI encoding is 65 and has a duration of 1 beat or 4/4. This means that we will fill in the first beat like this as shown in the next figure.


Figure 29: MIDI conversion for the time slots associated with the Crotchet

The second beat is occupied by two Quavers, the first on the G (SOL) line whose MIDI encoding is 67, the second on the F (FA) line whose MIDI encoding is 65, both with 1/2 beat lengths, i.e. only two time steps:


Figure 30: MIDI conversion for the time slots associated with the two Quavers

Finally we have in the third and fourth beat the Minim on the A (LA) line whose MIDI encoding is 69 and has a duration of 2 beats, i.e. 8 time steps:


Figure 31: MIDI conversion for the time slots associated with the Minim

This is the encoding of the first measure. The subsequent measures will be concatenated to this array to finally obtain a single array representing the entire melody. Obviously, before using it in the network, the symbols introduced as the underscore "__" and the rests "r" need to be mapped into integers. Although this representation is sometimes used, it is possible to use it only for *monophonic melodies*, that is melodies with a single melodic line, without therefore the possibility of sounds that can occur simultaneously.
The latter is the real problem. Take for example a fragment of Chopin's composition "Nocturne in e minor, Op. 72 No. 1" as shown in the next figure.

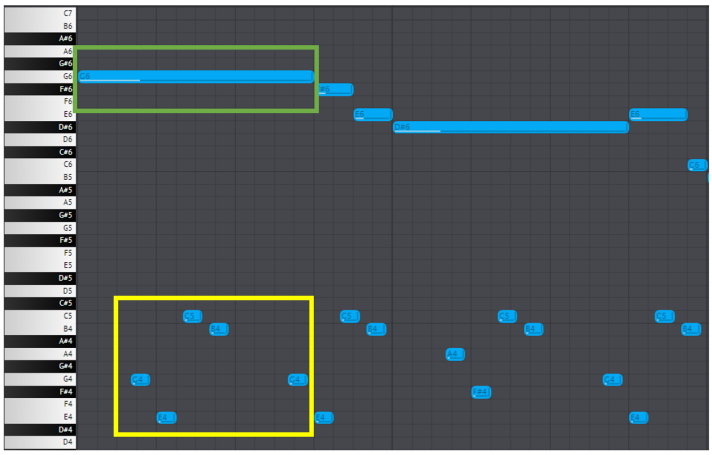

Figure 32: an extract from "Nocturne in e minor, Op. 72 No. 1"

Notice how the G6 is a note that lasts long enough for notes E4, G4, C5, B4, and G4 to still play at the same time. This "event" cannot be encoded in a time series because in a time series we can only represent a single piece of information at a time and therefore it is as if the other notes were translated at the end of G6 as shown in the next figure.

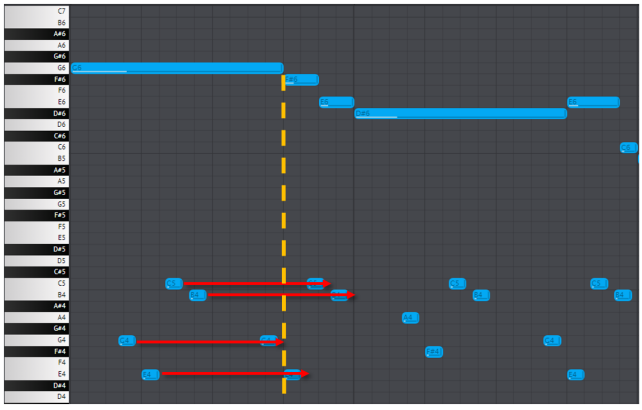

Figure 33: how an extract from "Nocturne in e minor, Op. 72 No. 1" appears after translation

By actually translating these notes, we translate the entire composition, effectively losing temporal information that is useful for understanding Chopin's style.

# 6 A Recurrent Neural Network (RNN) for polyphonic music

If with the first method we can succeed in predicting the next note of a monophony, that is a simple melody without any harmonic accompaniment, in this second method we will try to deal with the case of **polyphonies**, which in fact are the type of Chopin's compositions, therefore with the possibility that multiple notes are played at the same time. This, we have said before, is the concept of *chord*.

### 6.0.1 The data

The data are .mid files downloaded from the website *https://www.kunstderfuge.com/chopin.htm*. In total we have samples of the following Chopin compositions:

1. Ballades.
2. Etudes.
3. Mazurkas.
4. Nocturnes.
5. Polonaises.
6. Sonatas.

From the collection were not included compositions such as "Fantasie" and "Impromptus" because especially the last one could, in my opinion, confuse the network. In fact, romantics used to compose when a particular state of mind sculpted them (positively or negatively) by writing jet compositions (hence the name "Impromptus", precisely "sudden compositions") which could also move away from the "style" we are looking for to capture.

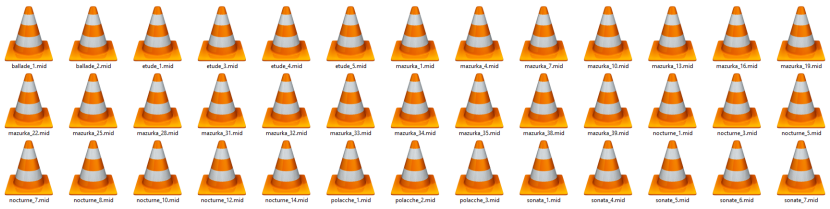

Figure 34: training .mid dataset

### 6.0.2 The vocabulary

The *load_midi_into_notes()* function will cycle inside the folder in the figure above (Figure 34) and for each .mid file it will extract two pieces of information:

- note.
- chord.

A chord is a set of notes played at the same time and with the same duration. We will encode this chord as a sequence of the notes that constitute it separated by the "dot" character ('.'). In this way we will get all the notes and chords that are present in the compositions. In figure 35 we have an example (in bold an example of chord coding).

$$
\begin{gathered}
[\prime C3\prime, \\
\prime 0\prime, \\
\prime C3\prime, \\
\prime 0\prime, \\
\prime 3\prime, \\
\prime 8\prime, \\
\textbf{'C1.C2.C3'}, \\
\prime 0\prime, \\
\prime 3\prime, \\
\prime B-2\prime, \\
... \\
\prime C4\prime, \\
\prime 7.0\prime]
\end{gathered}
$$

Figure 35: notes and chords extracted from .mid files

The next step is to create a *vocabulary* from these notes and chords extracted. Therefore a list will be created that will extract only the unique elements with which we will be able to associate a single integer number to each note or chord encoding.

$$
\begin{gathered}
\{0 :' C3', \\
1 :' 0', \\
2 :' 3', \\
3 :' 8', \\
4 :' C1.C2.C3', \\
5 :' B-2', \\
6 :' G\#3', \\
..., \\
594 :' C4', \\
595 :' 7.0'\}
\end{gathered}
$$

Figure 36: the vocabulary

#### 6.0.3 The sequences

Sequences are a series of notes and chords that logically follow one another. Each sequence will have associated a ground truth that will be equal to the note or chord that would come after in the sequence of notes or chords. The next sequence is created shifted on the left by one position the previous one; therefore the ground truth of the previous one will be part of the new sequence (throwing away the first note of the previous sequence), using as the new ground truth the note or chord that comes after.

$$
\begin{gathered}
input_1 : [0, 1, 2, 3, 4] \qquad output : [5] \\
input_2 : [1, 2, 3, 4, 5] \qquad output : [6] \\
... \qquad\qquad\qquad\qquad
\end{gathered}
$$

Figure 37: a toy example about the creation of sequences

For this architecture it was sufficient to take sequences of **100 notes as length**. 38896 sequences were extracted from the dataset used to train the model.
It is important to emphasize that **sequences must be created in such a way as to ensure that each sequence (input and output) is part of a single composition**. In fact, if no checks are made, a sequence can be found between two compositions. In this way the network is as if it were trying to learn the relationship between the end of one composition and the beginning of

the next. This makes no sense and must be avoided.
For greater efficiency in training we will **normalize the input values** with a simple but effective min max normalization.
Wanting then to predict the (conditional) "probability" of the next note, it will be necessary to convert each output into a **one hot encoding**. This will come in handy when using the categorical cross entropy and the softmax layer. Alternatively we could always leave the output as it is but remember to use the linear function as the activation function and use a sparse categorical cross entropy as loss.

$$input_1 : [0, 0.001680, ..., 0.006722] \quad output : [0, 0, 0, 0, 0, 1, 0, ..., 0]$$
$$input_2 : [0.001680, ..., 0.008403] \quad output : [0, 0, 0, 0, 0, 0, 1, 0, ..., 0]$$
$$...$$

Figure 38: normalization of the inputs and one hot encoding of the output of each sequence

### 6.0.4 Model and training

The model was implemented using Tensorflow APIs. To speed up the local training we used the GPU APIs on NVIDIA GPU card with a CUDA architecture on Windows 10 as operating system.
The problem is of **classification** of the next note or chord taking into account the past. The model (written with the sequential APIs) is the following:

```
model = Sequential()
model.add(LSTM(
    int(n_vocabs),
    input_shape=(network_input_training.shape[1],
                 network_input_training.shape[2]),
    return_sequences=True,
))

model.add(LSTM(int(n_vocabs/2)))

model.add(Dense(n_vocabs))
model.add(Activation('softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])
```

Figure 39: code of the model

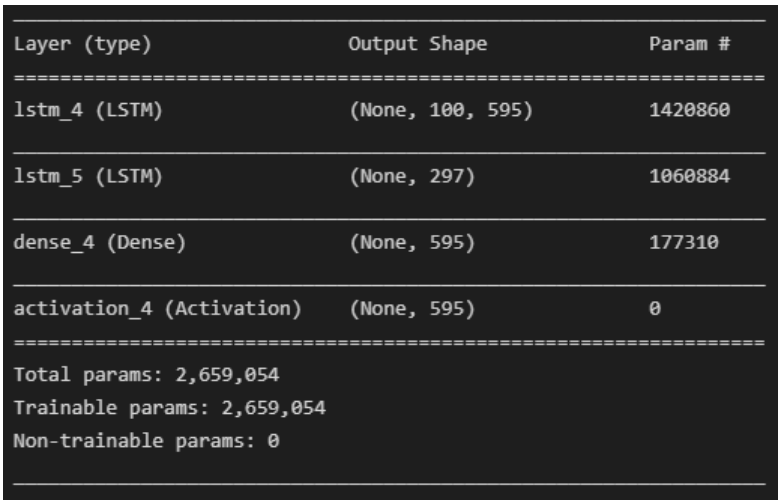


Figure 40: summary of the model

In the following figure, on the other hand, it is possible to see a figurative representation of it:

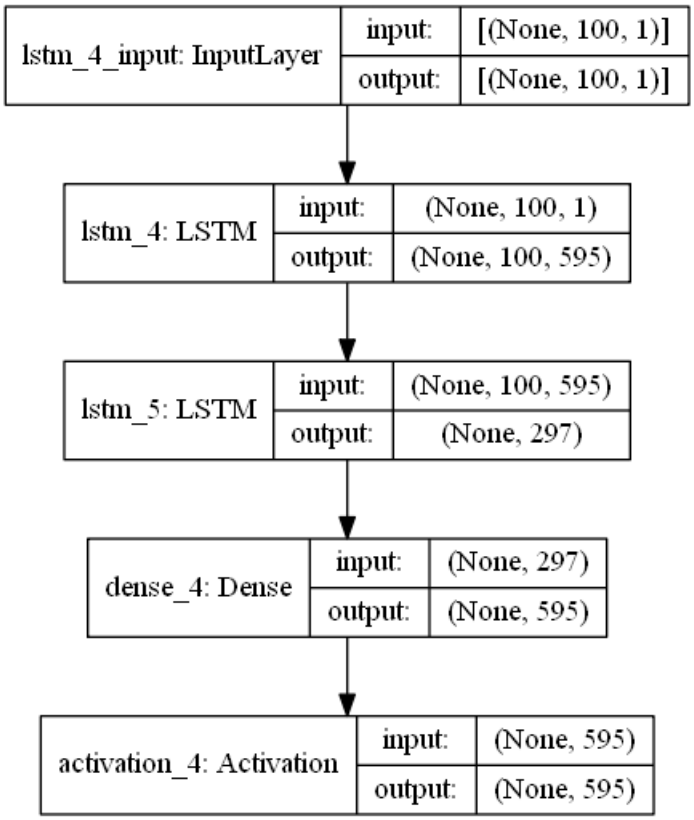


Figure 41: plot of the model

The heart of this *Recurrent Neural Network (RNN)* architecture are the two sequential levels of **LST memory cells**. These cells will allow us to capture the relationship between instances within each individual sequence. The hidden state of the second layer of LSTMs consists of 297 neurons whose output value is given to a softmax layer with the number of neurons equal to the number of words (595). This capacity will promote overfitting. The categorical cross entropy, whose formula I report below, will help us to measure the distance between the real distribution and that predicted by the network.

$$Loss = -\sum_{i=1}^{n_{vocabs}} y_{target} \cdot log(y_{predicted})$$

Figure 42: categorical cross entropy loss

The training was done using batch sizes of 64 out of a total number of epochs equal to 194 as you can see in the next figure.

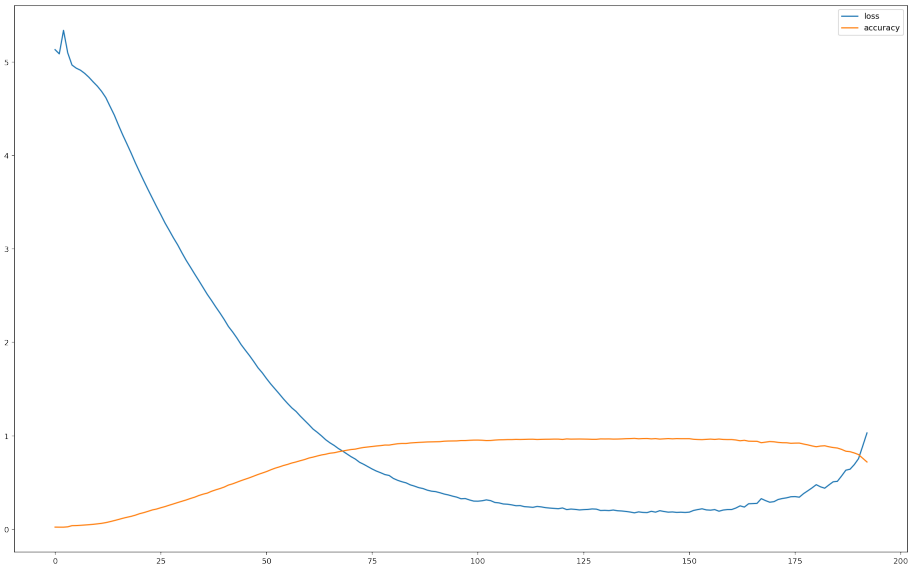


Figure 43: loss and accuracy during epochs of training

The accuracy achieved is 97.5%, therefore an almost perfect overfitting of the training data. In this way we modeled the conditional distribution of notes and chords.

### 6.0.5 The generation of music

For the generation of the music the skeleton of the model was recreated and its weights were initialized with those saved during the training. Having trained the network with a sequence of N notes or chords, the network will continue to make an unroll of the LSTM cell N times (in our case, N=100) before delivering its last hidden state to the neurons devoted to classifying the next note or chord. We will start with an initial **primer**, that is a sequence of N notes and/or chords. Since what later will be generated will depend on the quality of the primer initially given to the network, my advice is to give a meaningful primer, and therefore avoid inserting N notes at random. For this reason the primer will consist of N notes or chords taken directly from the training set. Once the next note or chord has been predicted, we will insert the result of this prediction at the end of the primer, discarding its first element of the sequence. We will use this new sequence to predict the second note or chord. Again, we will insert the second prediction at the last of the sequence, discarding its first element. At each instant, therefore, the sequence will always be of N elements.
To generate music never composed by Chopin, for each prediction made, having

an array of 595 ($n_{vocabs}$) probabilities, we will not always choose the highest one (otherwise we will always generate the same music), rather we will randomly sample the next note from this distribution according to the probability a each associated entry.

```
prediction_probabilities = model.predict(prediction_input, verbose=0)

index = numpy.random.choice(vocabulary,1, p = prediction_probabilities[0])
```

Figure 44: sampling the next note weighing each note with its probability

Below is an example of a probability distribution for the 300-hundredth note or chord to predict.

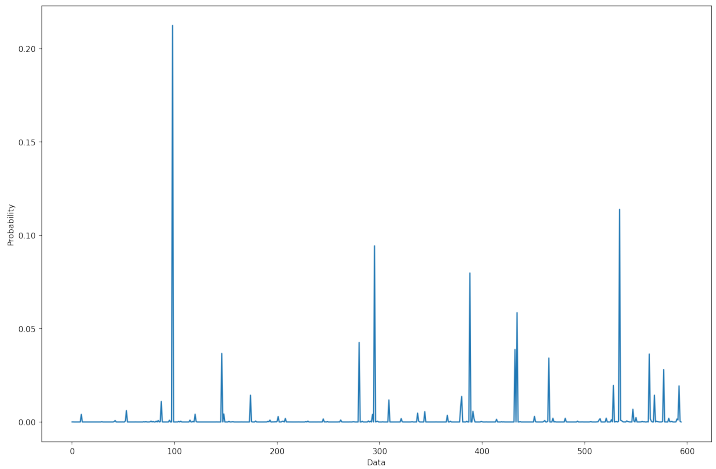


Figure 45: an example of probability distribution for the 300-hundredth note or chord

As you can see in the graph above, the previous notes influenced the probability of the next note or chord to be played. Choosing according to this probability, we will implement the concept of "generation of new music".

## 6.1 Trident architecture: an extension of the previous one

The previous architecture is very powerful in capturing the next note or chord, but not in understanding how long its duration should be nor in how far from the previous note or chord it should be reproduced. The extension of this architecture, which I have called *Trident*, provides for the insertion of 2 other

branches that can provide this information.

Below I will only deal with what is different from the first architecture.

### 6.1.1 The data

Since the previous architecture was trained on all the main Chopin compositions, I wanted to test this extension on only one subset: the **Nocturnes**. In this way I want to understand how much the concept of overfitting helps when the data available are even more scarce.
The Nocturnes I use are a total of 18 compositions.

### 6.1.2 The vocabulary

The Nocturnes I use are a total of 18 compositions. In this case the vocabulary consists of 1460 notes/chords.

### 6.1.3 The sequences

The sequences in this case need to be changed. In order for the architecture to predict the next note or chord with its relative duration and offset from the previous one, it is important that the exact same information is given at the input, obviously of the preceding notes or chords. As usual, we start by compacting the 3 information of each note or chord into a single array containing for each entry the note, the duration and the offset. We will then have an array of arrays like the following:

```
[['A#4', 1.7, 0.56  ],
 ['B4',  0.4, 1.7   ],
 ['C#2', 4.6, 0.22  ],
         ...,
 ['A#3.D4', 1.8, 0.1],
 ['D3', 1.8, 0.01   ],
 ['F3', 1.8,   0.01]]
```

Once we have created the vocabulary, here too we will proceed normalize everything with a *min max normalization*. For the notes and chords we will do as usual, while for the duration and the offset we should calculate the maximum and minimum. Eventually we will get something like this:

```
[[1.14383562e-01, 9.30232558e-02, 4.20105026e-02],
 [4.32876712e-01, 1.74418605e-02, 1.27531883e-01],
 [5.01369863e-01, 2.61627907e-01, 1.65041260e-02],
 ...,
 [9.38356164e-02, 9.88372093e-02, 7.50187547e-03],
 [8.45205479e-01, 9.88372093e-02, 7.50187547e-04],
 [9.53424658e-01, 9.88372093e-02, 7.50187547e-04]]
```

Since we will have 3 outputs, we will prepare 3 types of sequences. The input values will always be the same, but for each problem (e.g. probability of predicting the next duration) we will have its own output: a single output of the next note (or chord) for the problem of next note or chord prediction, a single output of the next duration for the problem of next duration prediction and a single output of the next offset for the problem of next offset prediction. For example, for the model that will be in charge of predicting the next duration, the sequences will be constructed like this:

```
input: [[1.14383562e-01, 9.30232558e-02, 4.20105026e-02],
        [4.32876712e-01, 1.74418605e-02, 1.27531883e-01],
        [5.01369863e-01, 2.61627907e-01, 1.65041260e-02],
                         ...,
        [5.21353564e-02, 1.74418605e-02, 7.50187547e-04]]

output:  [ 3.2 ]
```

For this architecture, having a smaller dataset (and therefore less data) I increase the length of each sequence bringing it to 500. This reduces the total amount of sequences to train the models with from 20800 (if each sequence was 100 long) to 13600.

### 6.1.4 Model and training

The model for the **prediction of the next note or chord** does not change. Obviously, by varying the vocabulary, the number of parameters that the network will have to handle will also vary.

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 100, 3)]          0

 lstm_3 (LSTM)               (None, 100, 438)          774384

 lstm_4 (LSTM)               (None, 100, 584)          2389728

 lstm_5 (LSTM)               (None, 730)               3839800

 dense_2 (Dense)             (None, 876)               640356

 dense_3 (Dense)             (None, 1460)              1280420

=================================================================
Total params: 8,924,688
Trainable params: 8,924,688
Non-trainable params: 0
```

Figure 46: summary of the pitch predictor model

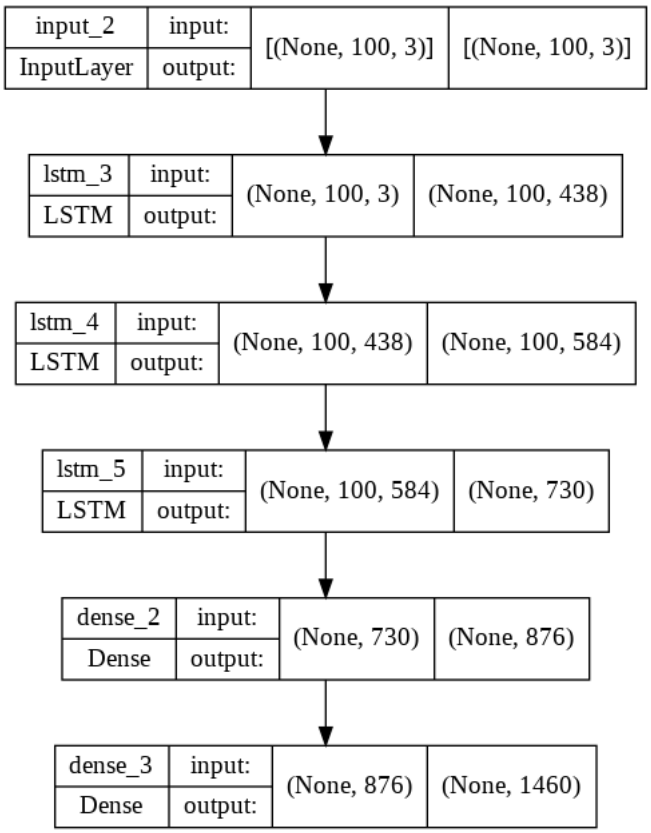


Figure 47: plot of the pitch predictor model

What has been added again are the two models for the prediction of the next duration and offset. They are identical, only the number of parameters change. Below is the code, summary and plotting of the model for the **prediction of the next duration**.

```
input_shape = (seq_length, 3)

inputs = Input(input_shape)
lstm_last_state = LSTM(int(n_vocabs*0.15), return_sequences=True)(inputs)
lstm_last_state = LSTM(int(n_vocabs*0.18), return_sequences=True)(lstm_last_state)
lstm_last_state = LSTM(int(n_vocabs*0.2))(lstm_last_state)

dense = Dense(int(n_vocabs*0.25))(lstm_last_state)

duration_output = Dense(1, activation="relu")(dense)

```

```
duration_predictor = Model(inputs, outputs=duration_output)
optimizer = Adam(learning_rate=0.00001)

duration_predictor.compile(loss="mean_squared_error",  optimizer=optimizer,
                           metrics=['mean_squared_error'])
```

Figure 48: code of the duration predictor model

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 100, 3)]          0

 lstm (LSTM)                 (None, 100, 219)          195348

 lstm_1 (LSTM)               (None, 100, 262)          505136

 lstm_2 (LSTM)               (None, 292)               648240

 dense (Dense)               (None, 365)               106945

 dense_1 (Dense)             (None, 1)                 366

=================================================================
Total params: 1,456,035
Trainable params: 1,456,035
Non-trainable params: 0
_________________________________________________________________
```

Figure 49: summary of the duration predictor model

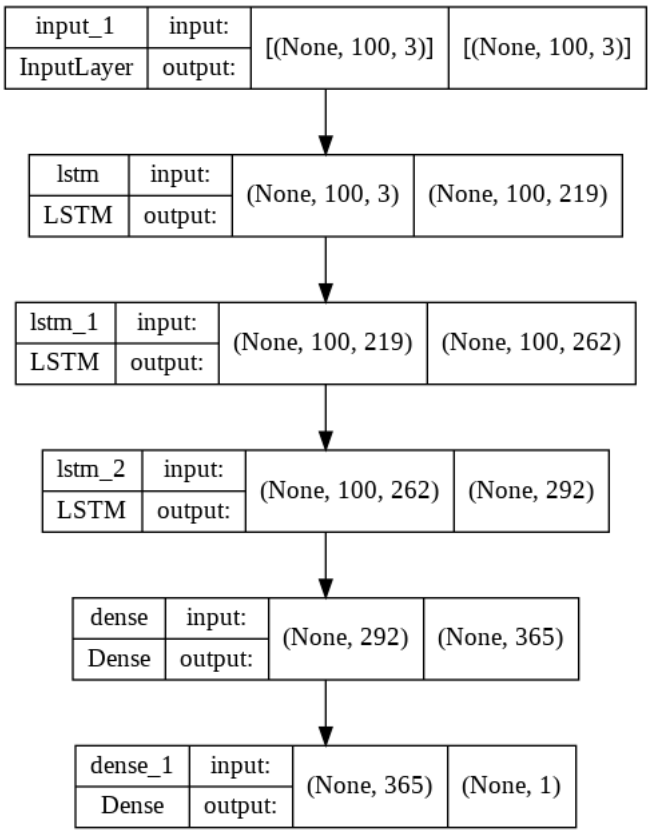


Figure 50: plot of the duration predictor model

Below is the code, summary and plotting of the model for the **prediction of the next offset**.

```
input_shape = (seq_length, 3)

inputs = Input(input_shape)
lstm_last_state = LSTM(int(n_vocabs*0.16), return_sequences=True)(inputs)
lstm_last_state = LSTM(int(n_vocabs*0.2), return_sequences=True)(lstm_last_state)
lstm_last_state = LSTM(int(n_vocabs*0.25))(lstm_last_state)

dense = Dense(int(n_vocabs*0.3))(lstm_last_state)

offset_output = Dense(1,activation="relu")(dense)

offset_predictor = Model(inputs, outputs=offset_output)
optimizer = Adam(learning_rate=0.0001)

```

```
offset_predictor.compile(loss="mean_squared_error",  optimizer=optimizer,
                         metrics=['mean_squared_error'])
```

Figure 51: code of the offset predictor model

| Layer (type) | Output Shape | Param # |
|---|---|---|
| input_2 (InputLayer) | [(None, 100, 3)] | 0 |
| lstm_3 (LSTM) | (None, 100, 233) | 220884 |
| lstm_4 (LSTM) | (None, 100, 292) | 614368 |
| lstm_5 (LSTM) | (None, 365) | 960680 |
| dense_2 (Dense) | (None, 438) | 160308 |
| dense_3 (Dense) | (None, 1) | 439 |

Total params: 1,956,679
Trainable params: 1,956,679
Non-trainable params: 0

Figure 52: summary of the offset predictor model

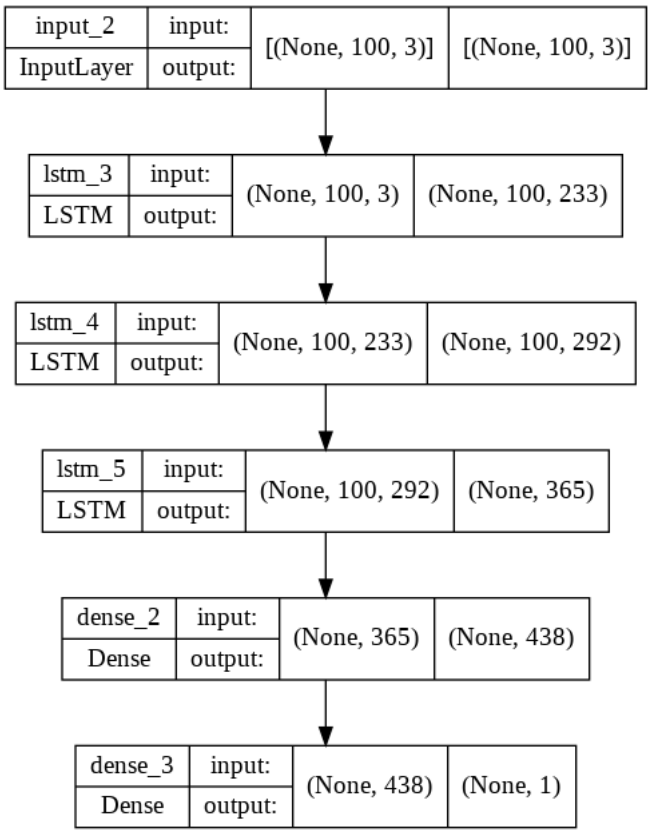


Figure 53: plot of the offset predictor model

Each model was trained individually until a good degree of overfitting was achieved. In the initial phase, **I used sequences of 100 instead of 500**. This allowed me to have a considerable speed up when I then wanted to extend the model with sequences of 500.
Below is the *training conducted for the predictor of the next note*:

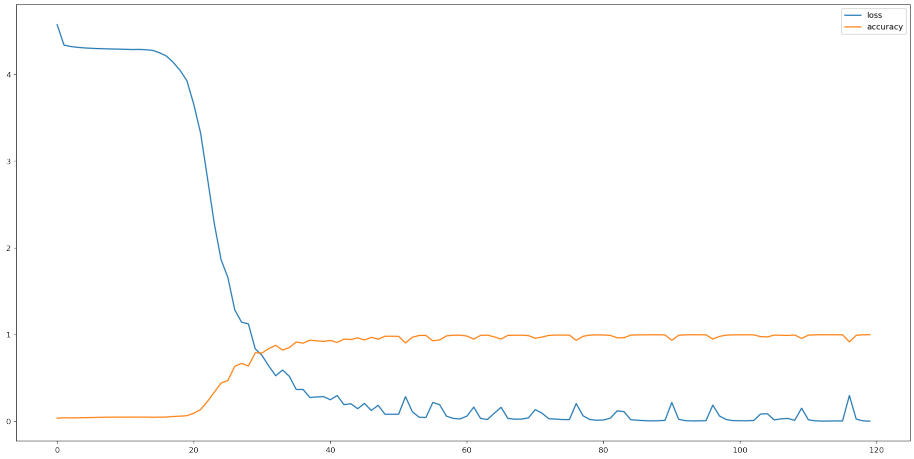


Figure 54: training of the pitch predictor model for sequences of length 100

The model achieves an accuracy of 0.9996 and a loss of 0.00324.
For the *predictor of the next duration*, the loss over time is shown in the following figure.

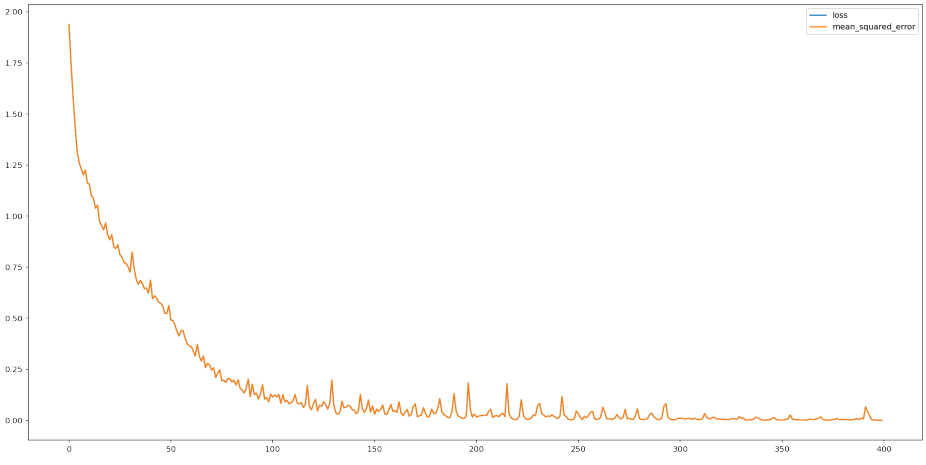


Figure 55: training of the duration predictor model for sequences of length 100

The model achieves a loss of 0.000633.
Finally, below, we have the training results for the offset predictor model.

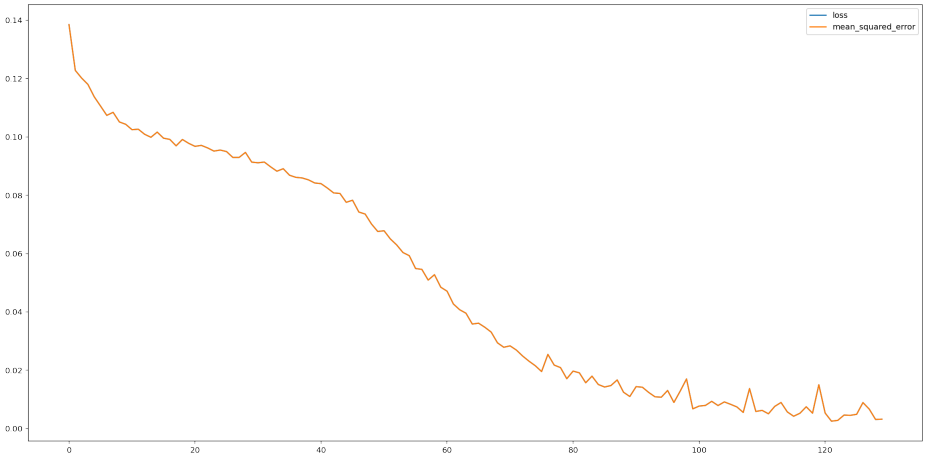


Figure 56: training of the offset predictor model for sequences of length 100

The model achieves a loss of 0.00255.

Training these 3 models for sequences of 100 was an initial step to find a good weight configuration as a starting point for training the nets with sequences of 500. In fact I noticed that training it starting with random weights (so without the step done now ) slows down training a lot (personally, after 150 epochs the accuracy was still 6%). This **fine tuning** of the network, starting therefore from weights that have "a meaning" rather than random, has greatly accelerated the training on sequences of 500 as can be seen from the following 3 figures.

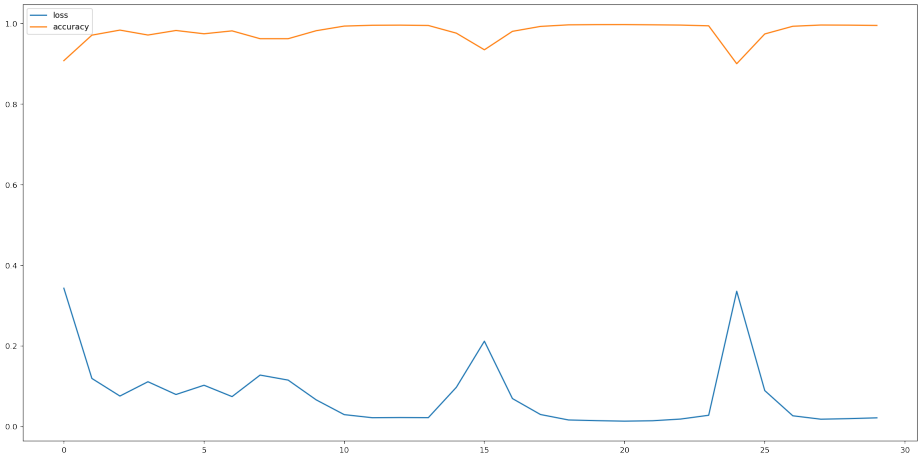


Figure 57: fine tuning of the pitch predictor model for sequences of length 500

The model achieves an accuracy of 0.9976 and a loss of 0.0134. Thanks to

the fine tuning, only 30 epochs were enough for the model to adapt itself to sequences of 500.

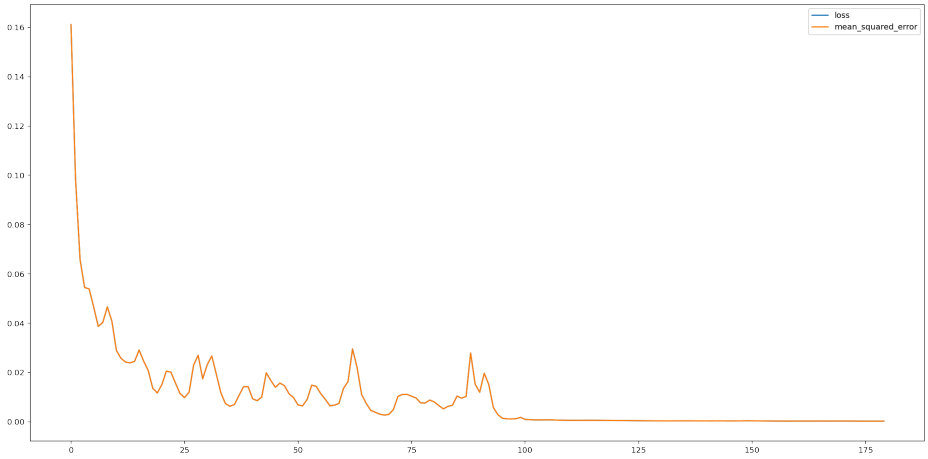


Figure 58: fine tuning of the duration predictor model for sequences of length 500

The model achieves a loss of 0.000255.

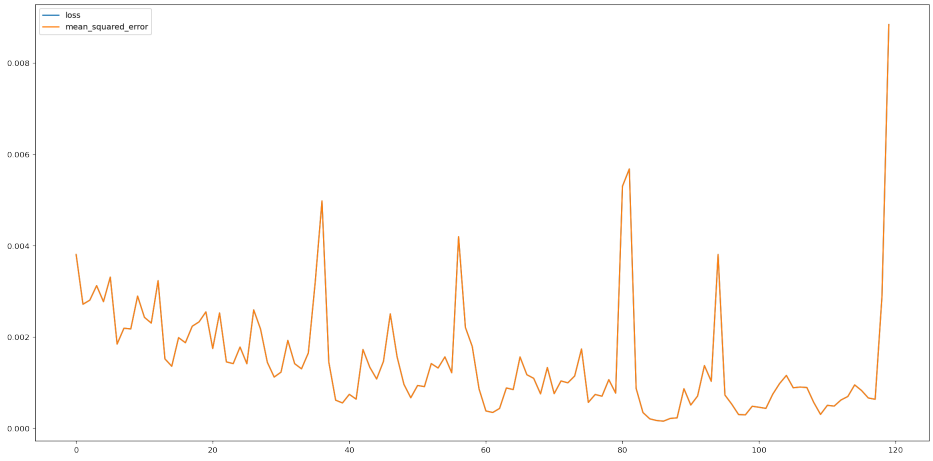


Figure 59: fine tuning of the offset predictor model for sequences of length 500

The model achieves a loss of 0.000160.

I would like to point out one fact. The Trident architecture has 3 branches.

It could all be compacted into a single branch, and therefore have a single network with 3 outputs. The network would then have solved one classification problem and two regression problems at the same time. I decided to split everything into three branches for two main reasons: trying to compact everything into a single network, the training took a long time and seemed to have difficulty in adjusting the weights to reduce errors in both the classification and the two regressions; the second reason is that the two regression problems do not need the high capacity that the classification branch needs.

### 6.1.5 The generation of music

Once we have trained the 3 models individually, we join them together to form the **Trident** I was talking about earlier.

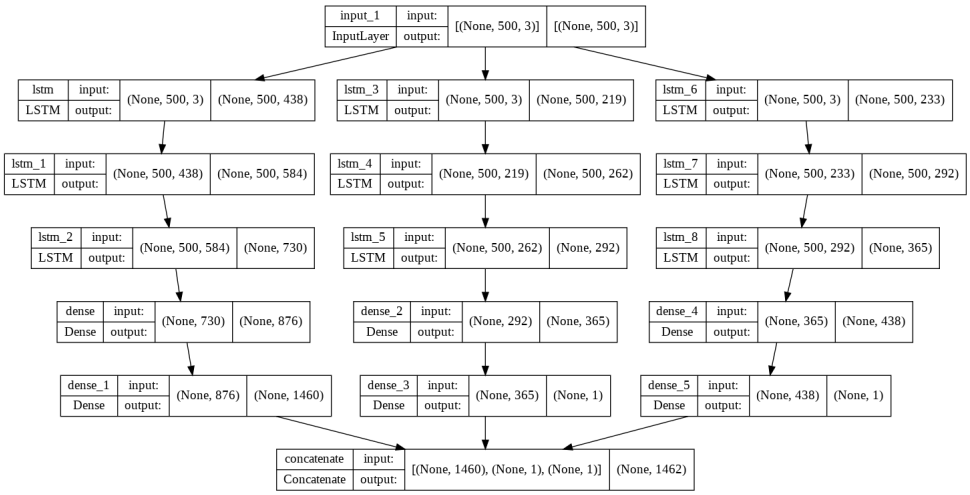


Figure 60: trident model

The model will input sequences of 500 elements. With the first branch on the left it will predict the next note or chord, with the branch in the middle it will predict the duration and finally with the branch on the right it will predict the offset to use to position the note. Finally I added a concatenation layer to concatenate all the results into one.
As usual we will start with a primer of 500. As mentioned before, I recommend a primer from the training set. Furthermore, this primer must start from a sequence such that the subsequent 499s do not end up in the next composition.

It is interesting to note how the probability distribution of the notes or chords changes the further away from the primer.

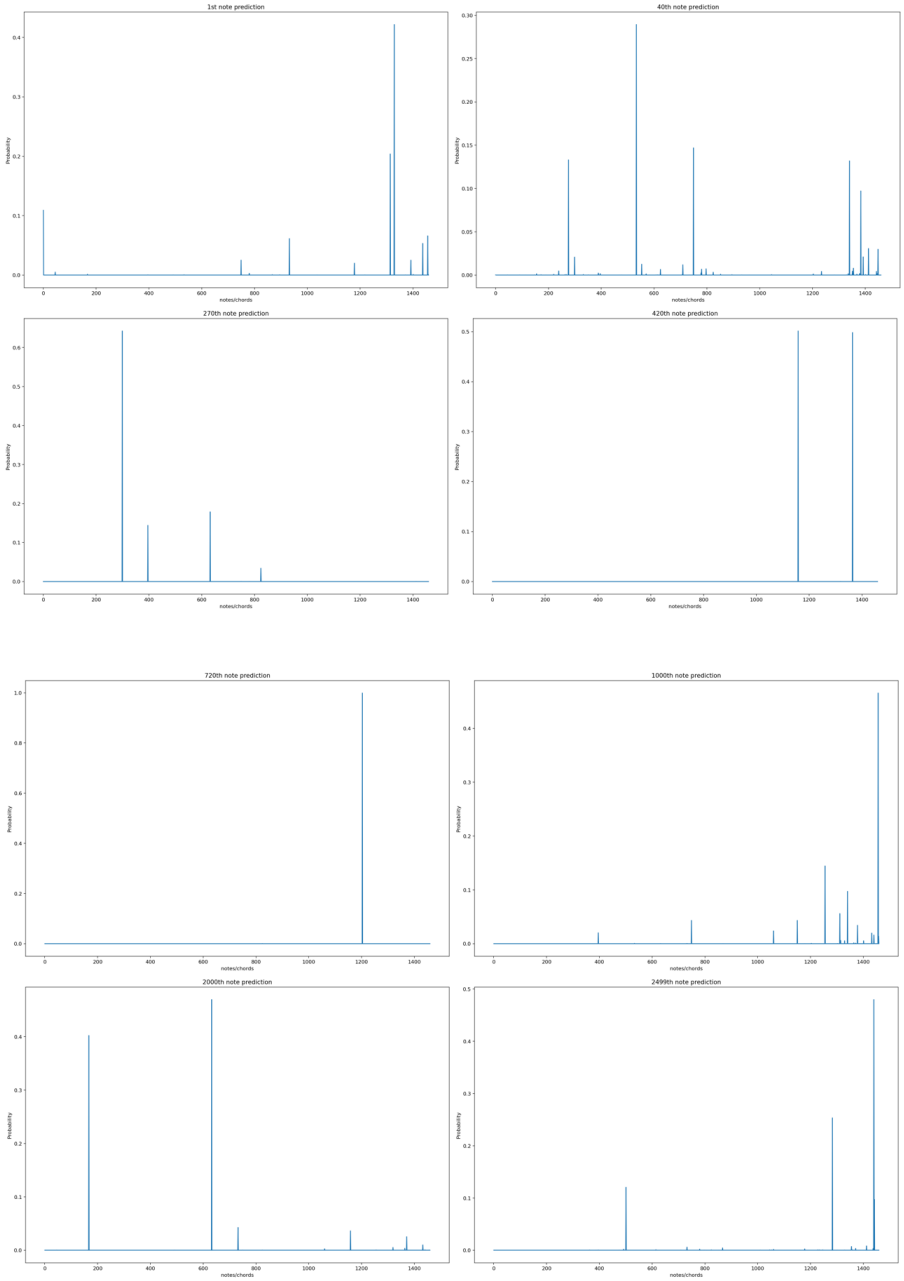


Figure 61: probability distribution of the ith note or chord

There are **two problems**.
In the first, as you can see, the distributions are initially richer. This is due to the fact that LSTM memory is great as long as we don't stray too far from

what the training set was. By predicting the next note or chord with a sampling according to the probability distribution, and using this to predict the next one, we give as an input a sequence that over time can really become different from the training ones and therefore the model will have to model a distribution conditioned by the previous presence of notes never seen before.
The second problem is related to the amount of data. As you can see, almost all distributions show a high probability only for 1 or 2 notes / chords. So even sampling according to probability will not allow us to generate new music, rather to reproduce the most probable one to which the primer belongs.

I advise you to listen to the *Chopin_AI_composition_No_1.midi* song generated by the network to "listen" to these two problems. The piece is wonderful, but having such a distribution, it can only (almost always) sample the same notes or chords.

But this is an excellent result in my opinion because it proves my theory that overfitting is not an enemy, it demonstrates how with a dataset that is important in terms of quantity of data we could really have richer distributions on which to sample and get the next significant note in auditory terms.

# 7 A Convolutional Generative Adversarial Network (C-GAN) for polyphonic music

To conclude my exploration, I implemented a **MidiNet-like architecture** from scratch. This is a very simple architecture, unlike **MuseGAN**. The results obtained are not wonderful, but certainly because although the GANs have greater "generative power" than the RNNs, they need a lot of data, which in my case is not possible, since I want to capture Chopin's style and therefore I only have these few data. But it seemed interesting to me to understand what is an approach that MuseGAN refines much more, but which is this at the base.

## 7.1 From midi to image

The key idea is to shift the dominance from that of music to that of images. There is a very simple but powerful mapping to transform a composition into one or more images. Let's suppose we have the composition in the figure below.

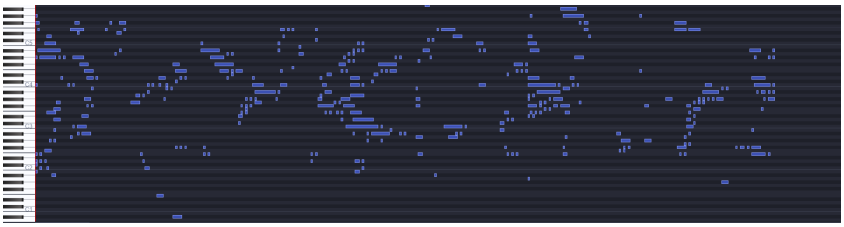

Figure 62: an example of notes and chords of a real composition

The image is first converted into its mirror form; after which it is divided into "pieces of 100", ie every 100 time slots is divided and each piece is converted into an image. The image is a 106x106 matrix of pixels with value only 1 and 0, so it represents a black and white image. This matrix "fills" those cells with 1 where in the real composition, in that slot, there was a note. Therefore if a note occupies for example 5 slots, in the row of the matrix we will have 5 cells of value 1. The height of the matrix is 106 because 106 are the possible MIDI notes. The width is 106 because it is easier for the network to work with square images. The whole process, described here, is shown in the image below.

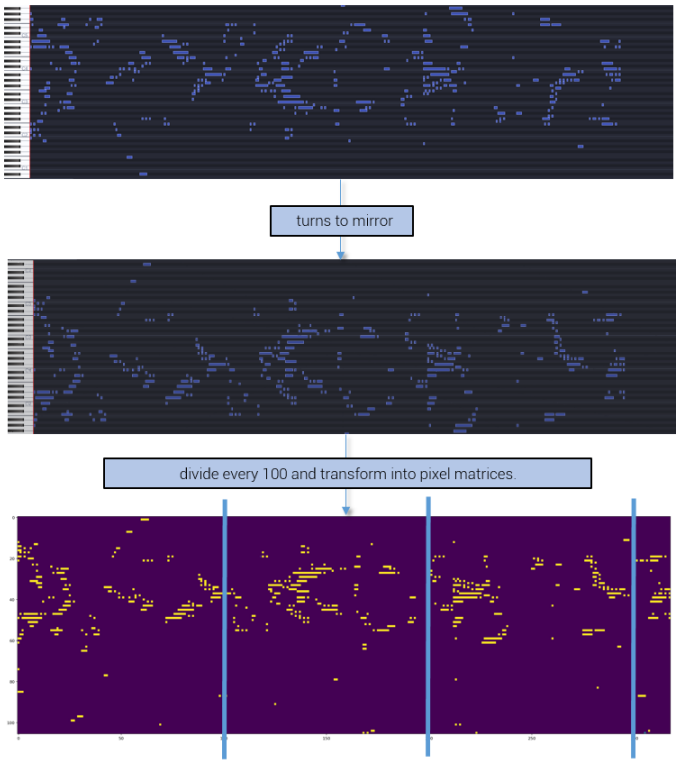


Figure 63: from midi to image flow diagram

In this implementation the division was every 100, but I want to emphasize that the shorter the width, the less correlation the various images will have with each other. In fact, the generative network will generate one image at a time. We will then be the ones to concatenate each of these images one beside the other. It is clear that each new generation is independent from the previous one. Stretching the width of the image, then giving to the generative network non-square images lengthens the training time and can make it much more complex. The purpose of this chapter is not to go into depth as we did previously with RNNs, but to present a possible and simple approach to using GANs to generate music.

## 7.2 From midi songs to images dataset

The dataset is the one already seen with the addition of other compositions by Chopin. In fact, for generative networks it is better to have as much data as possible.

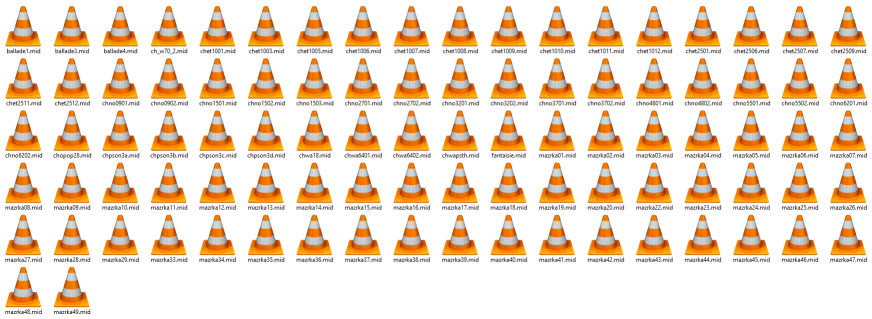

Figure 64: dataset of Chopin's midi compositions to train the C-GAN

In a loop cycle, for each composition we will transform this into a series of 106x106 images and save them in a folder. The whole process is shown in the figure below.

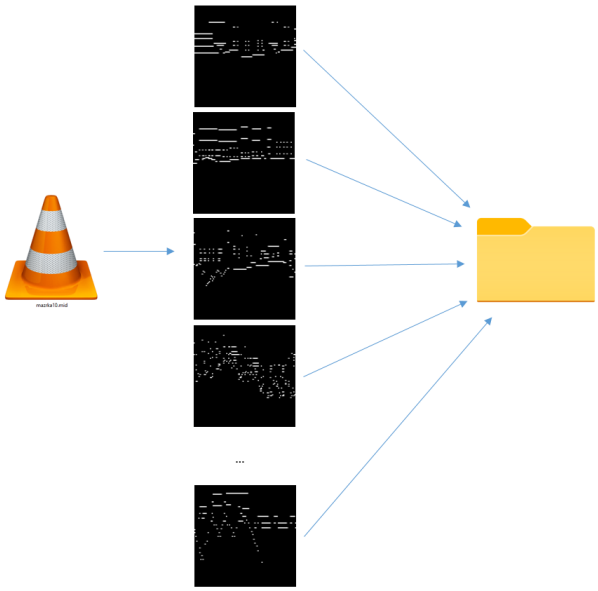

Figure 65: converting midi to images and save them to a folder (the dataset folder)

Eventually we will have a dataset of 1540 images.

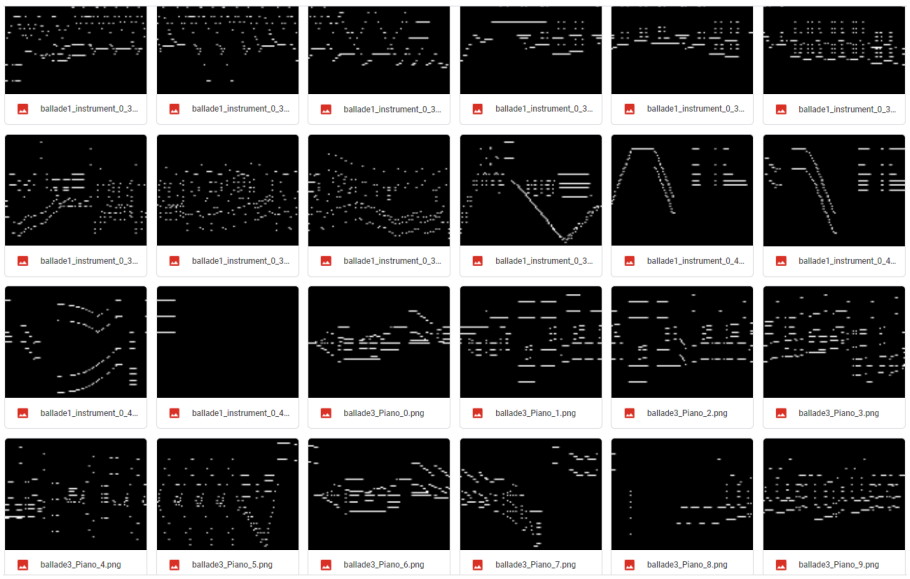


Figure 66: a dataset of images after converting midi compositions into images

## 7.3 The Architecture

The architecture presents as usual a generator and a discriminator. The **generator** takes in input a **latent vector**, that is a vector of values each time random with which it must be able to generate meaningful images. In this way, if the latent dimension is 100, the generator will generate $2^{100}$ possible images. Between this input and the output (matrix 106x106) we have various dense layers with different numbers of neurons, a LeakyReLU as activation function (with alpha equal to 0.2), a reshape layer and finally some transposed convolutions (i.e. they make a convolution opposite to the classic one). The use of *leaky rectified linear unit activation* (LeakyReLU for short) is to avoid the so-called "dead ReLU" (or "dying ReLU") problem, that is, the input values to the function are always below zero and therefore the output of the activation function will be zero (remember that the classic ReLU gave output $max(0, x)$). To avoid blocking the learning (because in the ReLU, for negative values the gradient is zero), a version of it called LeakyReLu is used whose activation formula is $max(alpha * x, x)$ where the larger the alpha the greater the importance will have the gradient when the x values are below zero, rather than always being zero in the case of ReLU.

```
input_shape = (latent_dimension)
inputs = Input(input_shape)

dense_1 = Dense(128*53*53, input_dim=latent_dimension)(inputs)
activation_1 = LeakyReLU(alpha=0.2)(dense_1)
```

```
reshape_layer = Reshape( (53,53,128))(activation_1)

dense_2 = Dense(1024)(reshape_layer)
conv2d_transposed_layer_1 = Conv2DTranspose(1024,(4,4), strides=(2,2),
                                            padding="same")(dense_2)

dense_3 = Dense(1024)(conv2d_transposed_layer_1)
activation_2 = LeakyReLU(alpha=0.2)(dense_3)
dense_4 = Dense(1024)(activation_2)
conv2d_transposed_layer_1 = Conv2DTranspose(1,(7,7), padding="same",
                                            activation='sigmoid')(dense_4)

generator_model = Model(inputs, outputs=conv2d_transposed_layer_1))
```

Figure 67: code of the generator

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 100)]             0

 dense_1 (Dense)             (None, 359552)            36314752

 leaky_re_lu_2 (LeakyReLU)   (None, 359552)            0

 reshape (Reshape)           (None, 53, 53, 128)       0

 dense_2 (Dense)             (None, 53, 53, 1024)      132096

 conv2d_transpose (Conv2DTra  (None, 106, 106, 1024)   16778240
 nspose)

 dense_3 (Dense)             (None, 106, 106, 1024)    1049600

 leaky_re_lu_3 (LeakyReLU)   (None, 106, 106, 1024)    0

 dense_4 (Dense)             (None, 106, 106, 1024)    1049600

 conv2d_transpose_1 (Conv2DT  (None, 106, 106, 1)      50177
 ranspose)

=================================================================
Total params: 55,374,465
Trainable params: 55,374,465
Non-trainable params: 0
_________________________________________________________________
```

Figure 68: summary of the model

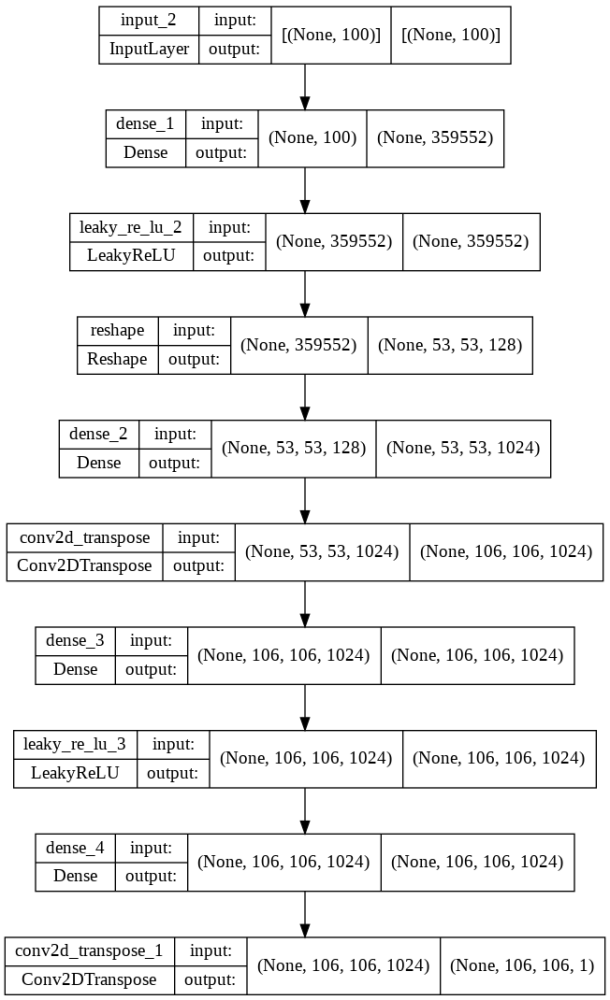


Figure 69: architecture of the generator

The **discriminator** has a very similar architecture with the addition of dropout and batchnormalization layers. Its input is a 106x106 matrix which during the training phase will be a real image (i.e. a real image resulting from the conversion of a midi file into an image) or a false image, i.e. generated by the generator.

```
#the input is an image (black and white) with 106x106 pixels
input_shape = (length_image, height_image, 1)
inputs = Input(input_shape)

#block1
```

```
convolutional_layer_1 = Conv2D(64, (3,3), strides=(2,2), padding='same',
                                        input_shape=input_shape) (inputs)
activation_1 = LeakyReLU(alpha=0.2) (convolutional_layer_1)
dropout_1 = Dropout(0.5) (activation_1)

#block2
convolutional_layer_2 = Conv2D(64, (3,3), strides=(2,2), padding='same',
                                input_shape=input_shape) (dropout_1)
activation_2 = LeakyReLU(alpha=0.2) (convolutional_layer_2)
dropout_2 = Dropout(0.5) (activation_2)

flattened_layer = Flatten()(dropout_2)
batch_normalization_layer = BatchNormalization()(flattened_layer)
output_discriminator = Dense(1, activation="sigmoid")(batch_normalization_layer)

discriminator_model = Model(inputs, outputs=output_discriminator)

discriminator_model.compile(loss='binary_crossentropy',
                            optimizer=Adam(lr=0.0002, beta_1=0.5),
                            metrics=['accuracy'])
```

Figure 70: code of the discriminator

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 106, 106, 1)]     0

 conv2d (Conv2D)             (None, 53, 53, 64)        640

 leaky_re_lu (LeakyReLU)     (None, 53, 53, 64)        0

 dropout (Dropout)           (None, 53, 53, 64)        0

 conv2d_1 (Conv2D)           (None, 27, 27, 64)        36928

 leaky_re_lu_1 (LeakyReLU)   (None, 27, 27, 64)        0

 dropout_1 (Dropout)         (None, 27, 27, 64)        0

 flatten (Flatten)           (None, 46656)             0

 batch_normalization (BatchN  (None, 46656)            186624
 ormalization)

 dense (Dense)               (None, 1)                 46657

=================================================================
Total params: 270,849
Trainable params: 177,537
Non-trainable params: 93,312
```

Figure 71: summary of the model

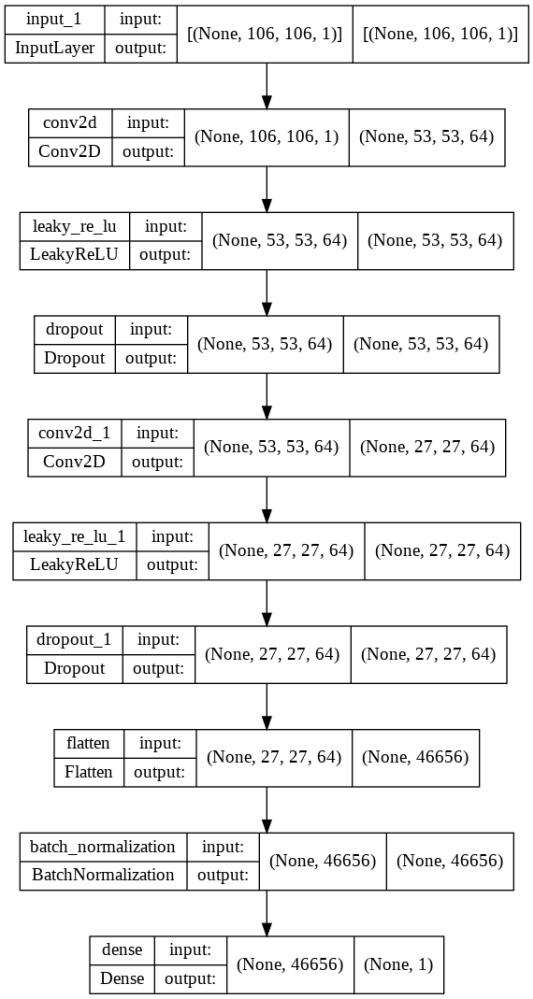


Figure 72: architecture of the discriminator

At the output of the discriminator we will have only one neuron with *sigmoid activation function* (the logistic one). In fact we will interpret the output as the probability that the input image is real. We will then use the *binary cross entropy (BCE)* as a loss function whose formula is reported below:

$$-1/N \sum_{i=1}^{N} y_i log(p(y_i)) + (1 - y_i) log(1 - p(y_i))$$

where $p(y_i)$ would be exactly the probability out of the sigmoid, i.e. that the image is real, while $y_i$ would be the ground truth at that instant, i.e. 1 if the image is real, 0 if it was generated artificially.

We will define this architecture as *Trainable = True*. We should then define it as Trainable = False when we train the generator, so that the discriminator cannot evolve and wait for the generator to reach its "level". Unfortunately **we cannot change the Trainable property of the discriminator to False dynamically after the model has been compiled**. For this reason we will create another architecture, called GAN, that has both the generator and the discriminator, where however before the compilation we will set the Trainable property of the discriminator to False so that by training the GAN we will train only the generator, while by training the discriminator model alone we will also train that of the GAN (which will therefore be the same).

```
discriminator_model.trainable = False
GAN = Sequential()
GAN.add(generator_model)
GAN.add(discriminator_model)

GAN.compile(loss='binary_crossentropy', optimizer= Adam(lr=0.0002, beta_1=0.5))
```

Figure 73: code of the GAN model (with discriminator blocked)

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 model_1 (Functional)        (None, 106, 106, 1)       55374465

 model (Functional)          (None, 1)                 270849

=================================================================
Total params: 55,645,314
Trainable params: 55,374,465
Non-trainable params: 270,849
```

Figure 74: summary of the model

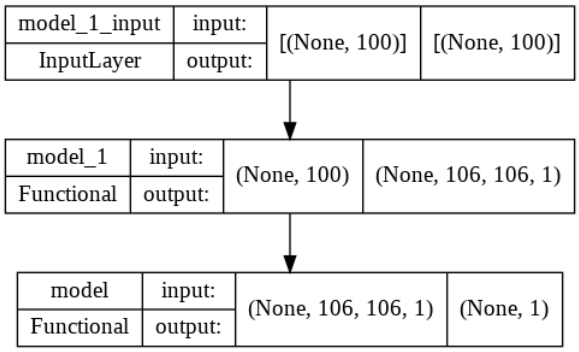


Figure 75: architecture of the GAN

### 7.3.1 The traning: a minimax game

The training is also called the *minimax game* as the generator and the discriminator compete with each other. In fact, the generator will create images with which it will have to be able to deceive the discriminator, i.e. this will have to classify that image as real even if it has been artificially created by the generator. It is therefore said that the discriminator wants to maximize the probability of correctly predicting real images from fake ones:

$$discriminator : maximize\ logD(x) + log(1–D(G(z)))$$

Where $D(x)$ is the probability predicted on the real sample x, while $D(G(z))$ is the one produced on the sample $G(z)$ generated by the generator. On the other hand, the latter wants to minimize this probability:

$$generator : minimize\ log(1–D(G(z)))$$

This is why it is called a minimax game, as one wants to maximize, the other to minimize.

I have implemented a GAN training as classic as possible but adding some new features in order to avoid some problems.
Once decided the number of epochs and the number of batches we want to process at each epoch, the first step is to create real music samples (i.e. images) by taking them from the previously created image dataset. Also, with the current state of the generator, an equal number of image samples are generated. We mark the first with target 1 and the second with target 0, and then we proceed to train the discriminator in distinguishing real images from false ones:

```
#take real music samples from pixels database (the ones obtained after conversion
from midi to image and after pure black and white conversion)
```

```
real_music_samples, ground_truth_real_music_samples =
            generate_real_music_samples(pixels, num_samples_for_discriminator)

#generate lantent samples
latent_samples =
        generate_latent_samples(latent_dimension, num_samples_for_discriminator)

#use the generator to predict an image giving lantent samples
images_generated_from_generator = generator_model.predict(latent_samples)
images_generated_per_epoch.append(images_generated_from_generator[0])

#create a zero ground truth (because are no real images)
ground_truth_images_generated_from_generator =
                                    np.zeros((num_samples_for_discriminator, 1))

#create the input samples to fed to discriminatore which are made up of both real
and fake music samples
discriminator_inputs =
            np.vstack( (real_music_samples, images_generated_from_generator ))
discriminator_ground_truth =
            np.vstack( (ground_truth_real_music_samples,
                        ground_truth_images_generated_from_generator ))


#train the discriminator (the one alone) on this current batch
discriminator_loss, _ =
            discriminator_model.train_on_batch(discriminator_inputs,
                                               discriminator_ground_truth)
```

Figure 76: training the discriminator

After the discriminator has been trained for a while (6 samples per batch), **we freeze it** (actually it is as if it was already locked if we now use the architecture previously called GAN), and train the generator. The idea is to input random latent vectors (12 per batch) and wait for the generator to create the related images. Then for each of these images we will have a discriminator able to tell if they are real or false. Since we are in training, the values of the respective ground truths will not be at 0 (since they are artificially created images) but at 1. In this way during the training we will change the weights of the generator so that the discriminator, with its current weights, can classify them as real images. In this way the generator will learn to make fun of the discriminator.

```
#generate latent samples (the double of before)
latent_samples = generate_latent_samples(latent_dimension,
                                         num_samples_for_generator)
```

```
#mark them as real (even if they are not)
ground_truth_latent_samples = np.ones( (num_samples_for_generator,1) )
#train the GAN (so, only the generator)
GAN_loss = GAN.train_on_batch(latent_samples, ground_truth_latent_samples)
```

Figure 77: training the generator

The following information was saved at each epoch:

- the accuracy of the discriminator in classifying real images as such.
- the accuracy of the discriminator in classifying artificially generated images as false.

A plotting of the values over time below makes it clear how these two accuracies vary.

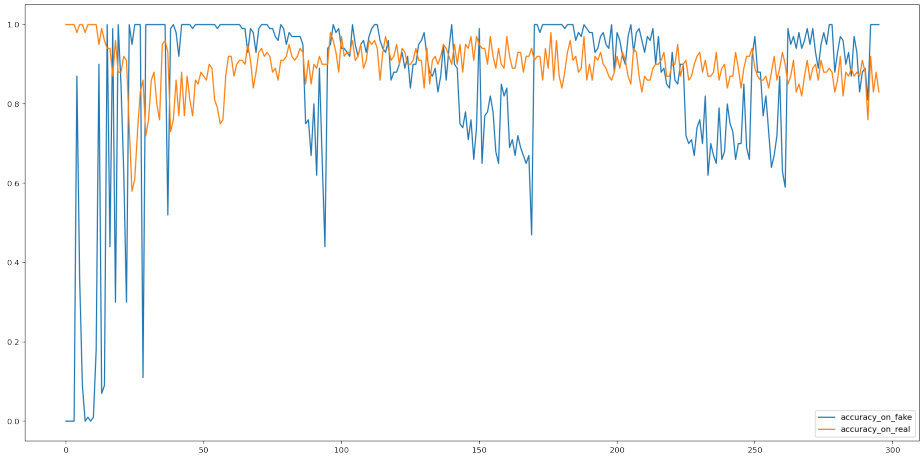


Figure 78: accuracy of the discriminator on real and fake images

Initially the discriminator is not able to classify real images as real, even if he is immediately able to classify false images as false. After a few epochs, the accuracy on real images increases, at the expense of fake ones. In fact, it almost seems that in order to understand when an image is real, he has to modify its weights enough to alter his understanding of fake images. However, 100 epochs are enough for the discriminator to be able to 100% distinguish real images from fake ones. In later times the generator learns to generate very realistic fake images, in fact the accuracy on fake images decreases (therefore the discriminator does not always classify a fake image as fake). The discriminator needs other epochs to increase its accuracy on fake, but at the expense of accuracy on real

images. This is a good sign, it means that the images produced by the generator are realistic. Around the 180th epoch it recovers almost 100% on both accuracies. From then on, the same similar situation mentioned above occurs. Then the generator increases its ability to generate realistic images, the discriminator loses percentages on both, to recover on the fake it decreases the percentages on the real ones, and then in the end it recovers both.

It is also interesting to **visualize** how training evolves over the epochs. The following image shows how the generator is getting better and better at generating realistic images.

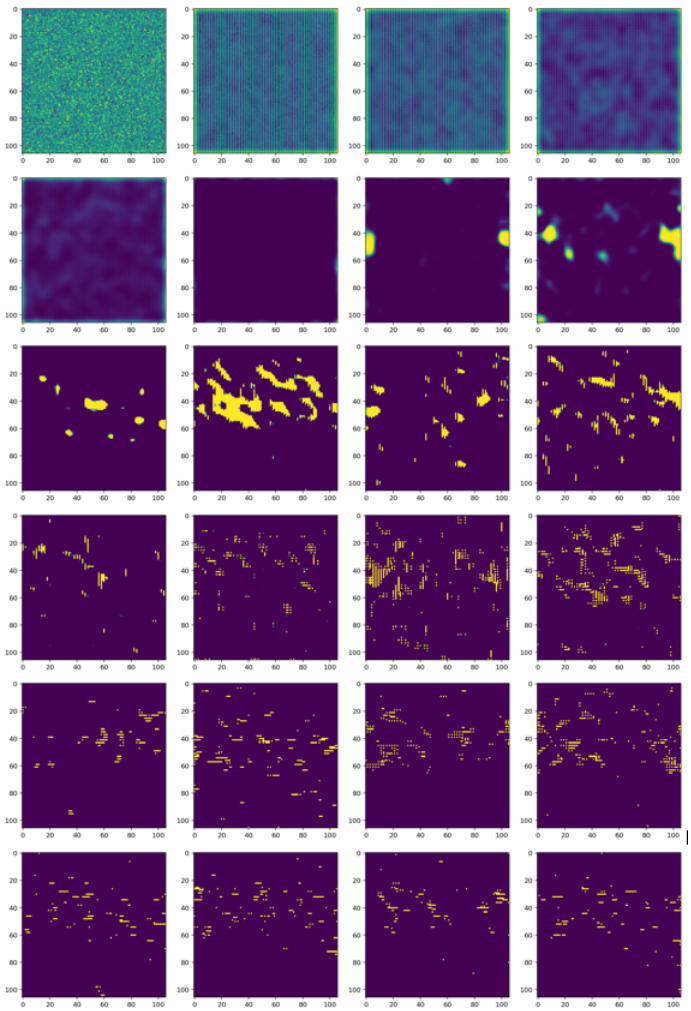

Figure 79: quality of images generated by generator during the epochs

### 7.3.2 Generating music

We have come to the end of this chapter. We just have to see how to generate music. What we will do is give the generator a latent vector as input and wait for it to generate an image. We will do this N times. Eventually we will concatenate each image next to each other to get only one. Finally we will convert this image to .mid file.

The following code generates 3 latent vectors and the related images generated by the generator, then it concatenates them into one.

```
number_of_images_to_generate = 3

images_new_song = None

for i in np.arange(0,number_of_images_to_generate):
  input_to_generator = generate_latent_samples(latent_dimension,1)
  image_generated = generator_model.predict(input_to_generator)
  if i==0:
    images_new_song = np.squeeze(image_generated)
  else:
    images_new_song = np.hstack( (np.squeeze(images_new_song),
                                  np.squeeze(image_generated)))
```

Figure 80: code to generate new music

The generated image is the one in the figure below.

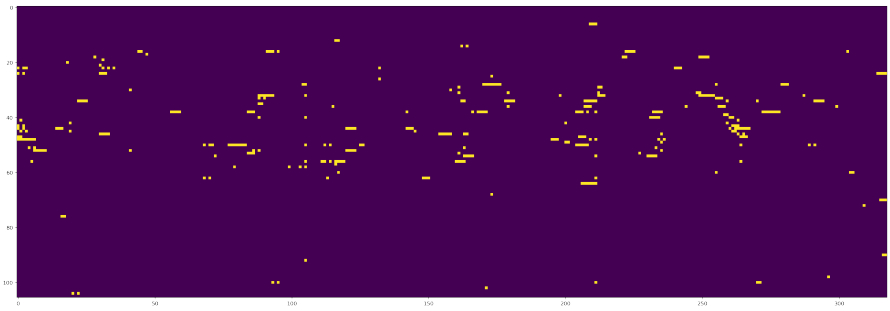

Figure 81: images (after concatenation side by side) generated by the generator

This image, once denormalized (i.e. its values are multiplied by 255) can be reverted to a midi file and reproduced as shown in the following image.

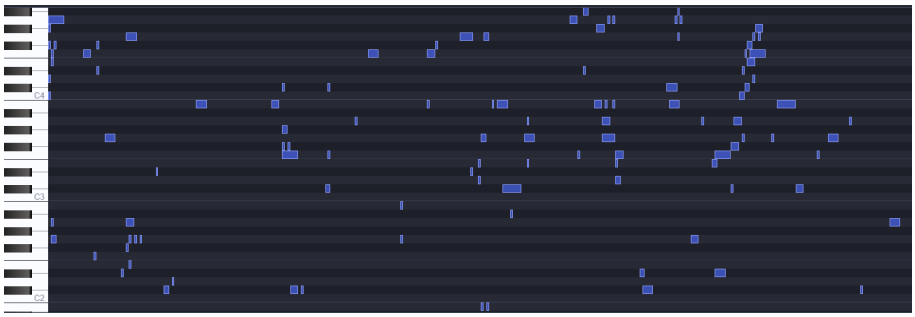


Figure 82: playing music artificially generated by GAN

# 8 Conclusions

It has been a beautiful journey, exploring the fundamentals of neural networks devoted to the generation of music. My treatise does not aim to invent anything (although I have proposed a simple Trident architecture, but nothing complex compared to the current state of the art), rather to give a clear idea of what is the modern approach to preprocess music and what are the network types, from RNNs to C-GANs, which are used to generate music. I have avoided using pre-trained models because I think that implementing from scratch is the only way to fully understand the theme. Furthermore, in the whole discussion I have ventured the hypothesis that overfitting is a friend, when it is a common rule that this is not. But I like not to give anything as a rule, obviously when I think there is a basis for "betraying" this rule. RNNs turned out to be better than C-GAN, although for the latter I brought a very basic implementation. In fact, GANs have been very successful in recent years (think MuseGAN) in this area. However, with the same (few) data, the RNNs proved to be better. GANs need a lot more data. Since the aim of the project was to capture Chopin's style, the data by their nature are few and therefore the GANs, in my opinion, cannot be used. As for the RNNs, thanks to the Trident architecture in which I tried to capture Chopin's style on the Nocturnes, I have also shown that even if the RNNs with few data are better than the GANs, in any case to have a greater generative power it is still necessary use a lot of data otherwise overfitting really becomes an enemy as it would always lead to sampling the same notes or chords and therefore reproducing the same training set.

# References


[1] Bron, *Rhythm basics: beat, measure, meter, time signature, tempo.*

[2] broughtonpianos.co.uk, *What are piano keys made of and why?*

[3] classical music.com, *What is the difference between a sharp and a flat note?*

[4] José Maria da Costa Simões, *Deep learning for dynamic music generation.*

[5] iZotope Contributor David Bawiec, *Time signatures explained, part 5: Creating interesting motion with meter changes.*

[6] David Foster, *Generative deep learning*, O'Reilly, 2019.

[7] Hasa, *What is the difference between monophony polyphony and homophony.*

[8] François-David Pachet Jean-Pierre Briot, Gaëtan Hadjeres, *Deep learning techniques for music generation – a survey.*

[9] Hollin Jones, *Midi velocity: What it is  how it works.*

[10] Yi-Hsuan Yang Li-Chia Yang, Szu-Yu Chou, *Midinet: a convolutional generative adversarial network for symbolic-domain music generation*, (2017).

[11] Bri Lundberg, *An easy guide to scientific pitch notation.*

[12] Anthony Brandt  Robert McClure, *2.16: Tonic, mode and key.*

[13] Musicnotes, *A complete guide to time signatures in music.*

[14] Salman AlSaigal Nabil Hewahi and Sulaiman AlJanahi, *Generation of music pieces using machine learning: long short-term memory neural networks approach.*

[15] NotationSoftware.com, *What is midi?*

[16] Rowan Pattison, *Understanding c major: First position, chords, and scale.*

[17] Tariq Rashid, *Make your first gan with pytorch*, Independently Published, 2020.

[18] Christian Salerno, *Gli accordi: cosa sono e come si formano.*

[19] ———, *Tono e semitono.*

[20] Catherine Schmidt-Jones, *Sharp, flat, and natural notes - lecture and notes.*

[21] silvernine209, *Edm generator using lstm.*