Escuela
Politécnica
Superior

Automatic Music Transcription Using Neural Networks

Computer Engineering Degree

Final Thesis

Author:
Manuel Mínguez Carretero

Advised by:
Antonio Pertusa Ibañez
Carlos Pérez Sancho

May 2018

Universitat d'Alacant
Universidad de Alicante

Abstract

The use of artificial intelligence to solve problems that were not previously viable is growing exponentially. One of these problems is obtaining the musical notes (the music score) given a song in audio format. This task has a high complexity due to the large number of notes that can be played at the same time by different instruments. This project makes use of the Musicnet dataset which provides the audio data of 330 songs with their corresponding note labels. To extract relevant information and derive the features, Constant-Q Transform has been applied to transform the audio data to the frequency domain in a logarithmic scale. In addition, one-hot encoding vectors have been used to represent the output data, i.e., the music notes. Then, a deep neural network is trained to recognise the score given the music audio information.

A research has been carried out to find the most appropriate methods to solve the problem. Besides, different topologies of neural networks have been developed to find which of them offers the best outcomes. The results obtained are positive since a high percentage of prediction accuracy has been achieved taking into account the great number of combinations that the problem presents.

Resumen

El uso de la inteligencia artificial para solucionar problemas que antes no eran viables está creciendo exponencialmente. Uno de estos problemas es la obtención de las notas musicales (la partitura) dada una canción en formato de audio. Esta tarea tiene una complejidad elevada debido al gran número de notas que pueden ser interpretadas al mismo tiempo por distintos instrumentos. Este trabajo de fin de grado hace uso de la base de datos Musicnet, la cual proporciona los datos de 330 canciones y sus notas correspondientes etiquetadas. Para extraer la información relevante y derivar las características se ha aplicado una Transformada *Constant-Q*, la cual transforma los datos al dominio frecuencial en una escala logarítmica. Además, se ha utilizado una codificación *one-hot* para representar la salida del sistema, es decir, las notas obtenidas. Finalmente se ha entrenado una red neuronal profunda para reconocer la partitura dada la información de audio de entrada.

Se ha realizado una investigación para encontrar los métodos más apropiados para resolver el problema planteado y se han desarrollado distintas topologías de redes neuronales para encontrar cuál de ellas ofrece mejores resultados. Los resultados obtenidos son positivos ya que se ha conseguido llegar a un porcentaje de acierto elevado dado el gran numero de combinaciones que plantea el problema.

Contents

Chapter 1

Introduction

1.1 Objectives

This project aims to develop and apply the knowledge, skills and competences acquired during the course of the degree. In my case, I have not studied anything related to artificial intelligence or audio signals. However, I have learned many elements that include fundamental aspects of this discipline such as programming in different languages and having a vision of how to solve problems that arise. Mainly, this project has two objectives:

- The first objective is to obtain good results for the proposed problem. The task to be solved is to get predictions of the musical notes played in an audio file containing a song. Therefore, the goal is to develop a software tool that, given a song, is able to analyse a song and output all the musical notes that have been played at every instant. In the past decades, solving this problem was not an easy task because there were no proper data to analyse, but nowadays, thanks to recent large databases of music, it is currently possible to create models that are able to predict the musical notes played in a song.

- In addition, it is rewarding and challenging at the same time to do a research and develop something I have never dealt with before. Learning from scratch on my own is an added effort to the development of the work. However, it is a challenge that I want to take on because I have always been interested in the potential of artificial intelligence and this project is an ideal opportunity to learn. Thanks to the development of this project, I will indirectly fulfil the above mentioned objective because I will have to learn the basics of artificial intelligence and how to apply it step by step.

1.2 Artificial intelligence

The purpose of this work is to get the musical notes played in an audio file containing a song. The first decision to make is which technology to use, if doing it with rules and programming we can get good results or, otherwise, is it better to apply machine learning methods to get more accurate outcomes. To analyse if deep learning is the proper way to solve the problem, it is necessary to know what artificial intelligence is, what are its advantages, and when to use it.

Artificial intelligence (AI) is the science of making computers solve problems as if they were humans. There are different kinds of AI but we are going to focus on machine learning.

Machine learning can be defined as a family of methods that are able to predict results using the experience learnt in an earlier state. During the training stage the program receives a set of features extracted from the inputs (e.g images or sound) and the data is analysed by the method which finally provides a result. Those results might be wrong but as the data has been labelled before the execution, the program is able to know whether the predictions are right or not [17].

If the output results are correct, the method knows that things are done properly and it has to continue in the same direction. On the contrary, if the results are not what they were expected to be, the model should be adapted.

The main interest of machine learning is that the computer can predict outputs from inputs that were not seen before. For example, in the case of MusicNet [11], the model will learn the musical notes from a set of given songs but after that, it is expected to predict the musical notes from songs that has never analysed before.

Differently from this procedure, deep learning methods aim to get predictions from the raw input data. Traditional machine learning approaches needed to get a series of features from the input data in order to use them to train a model. However, with deep learning, this process is not necessary, as these techniques are excellent at learning representations that are suitable for the target problem.

The use of deep learning has been proved to get high success rates in a variety of applications [17], including

- *Text and document classification*
- *Natural language processing*
- *Speech recognition, speech synthesis and speaker verification*
- *Optical character recognition*
- *Computer vision tasks*
- *Games*
- *Network intrusion*
- *Recommendation systems*

But all that glitters is not gold, and deep learning has also some disadvantages that are important to take into account. One of the main obstacles when designing a neural network is that it is indispensable to have a large amount of data and it has to be relevant. Having a proper dataset is one the the main challenges.

Another challenge when using deep learning is how to interpret the results to determine the effectiveness of the models used. In addition, several deep learning architectures are typically needed to be evaluated in order to find out which one is the most accurate and fits the better to solve the problem [30].

Furthermore, developing a deep learning model requires a high computational demand, but thanks to the fact that some large technological companies such as Google or Microsoft provide Jupyter Notebooks running on their servers for free, it is now possible to develop deep learning models fast and for free.

In this final year project, different neural network architectures are tested in order to determine which one is the most suitable one for this task. All things considered, in order to translate the music into musical notes we use deep learning and within this area, Convolutional Neural Networks (CNN).

For this, we use a large dataset (MusicNet [11]) provided by the Washington Research Foundation Fund for Innovation in Data-Intensive Discovery which contains all the necessary data to apply deep learning to address music transcription problems. Without using any artificial intelligence method, results are far from optimal in the prediction of musical notes from all kind of songs.

Chapter 2

Environment

Deciding which framework to use in order to create the neural network is a crucial part of the development of this project. There are many different programming languages used in existing artificial intelligence libraries, such as Java, Lisp, and Prolog. However, the two main languages used with deep learning techniques are C++ and Python. Each one has its own advantages and disadvantages.

- C++ is one of the most robust and fast programming languages currently available, therefore it is possible to apply artificial intelligence techniques because it works in a low level of abstraction. Having to deal with memory usage makes the development harder than using other languages such as Java or Python, but as the same time it is possible to get remarkable results. There are some libraries for machine learning written in this language which offer a great software reliability and also accelerate the development time of the program. There are also some libraries for the visualization of the results obtained [15].
- Python is the programming language that is most used in artificial intelligence, and in particular for deep learning, thanks to its simplicity and the amount of existing libraries. This is an interpreted language so it abstracts the user from all the low level complexities, and thus the user can focus on the problem and how to quickly translate ideas into code [22]. Thanks to the libraries for machine learning development, it is not necessary to have advanced knowledge to be able to start and develop code [29].

Due to the simplicity of this last programming language and the multitude of available libraries, this project will be implemented with Python 3.6 supported by the libraries explained below.

Frameworks

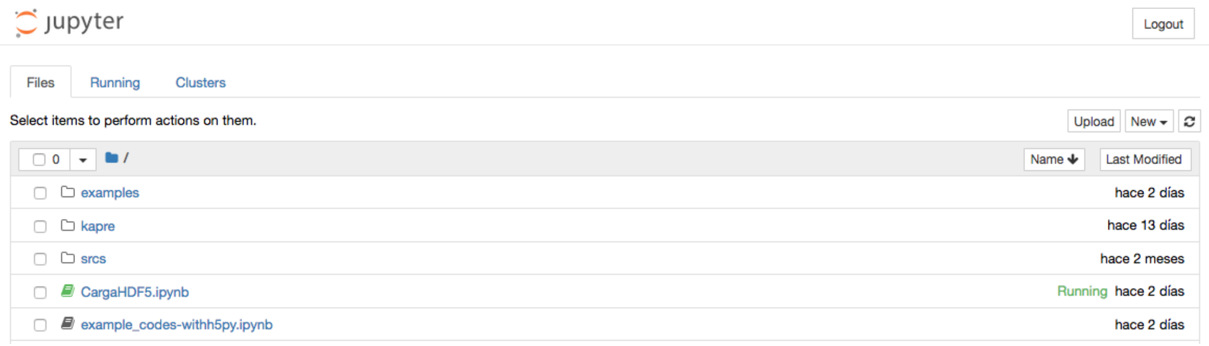
The first library and the most important one is *TensorFlow* since it is going to be the back-end of other libraries. Tensorflow is used by large companies such as Google, Twitter, Nvidia and Intel and it has been created by the Google Brain team. Tensorflow was introduced to develop deep learning using neural networks. One of the main perks is that can be executed on different operating systems as well as it offers the possibility to deploy the program by using the GPU or CPU acceleration to speed up the process as much as possible [4].

This library is based of executing Data Flow Graphs which are composed of nodes that represent mathematical operations and edges, which are multidimensional data arrays also called *tensors* [26].

One layer above Tensorflow, *Keras* is the front-end chosen to implement the neural network models in this project. Keras is a python API used for the design of neural networks at a higher level of abstraction. It is not necessary to program complex mathematical operations in order to make a neural network, although it is essential to understand how and why it works in order to create an optimised network. Thanks to the use of Keras, neural networks can be built as quickly as possible and therefore it is feasible to train and improve the accuracy in a shorter time than having to do everything from scratch. Furthermore, Keras can be run seamlessly on CPU and GPU.

The types of neural networks that can be created using Keras are Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN), giving also the chance to combine those layers to create mixed networks such as RCNN [20]. The details of CNN and RNN will be explained during the explanation of the neural network topology proposed for automatic music transcription.

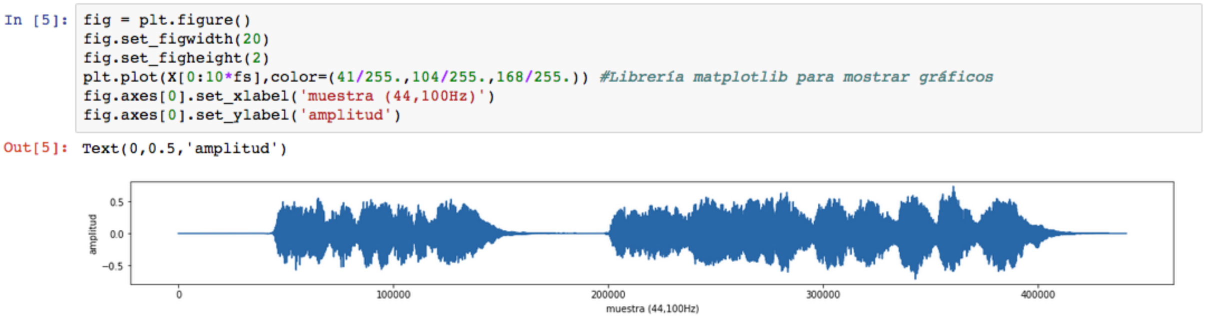
Jupyter Notebook App is an application that allows to execute a Python kernel either on a remote server or on the local machine, and offers a web interface to write and execute Python code. It has a control panel that allows to create, move, remove or duplicate documents as well as editing the existing documents [10]. An example of the control panel could be the following:



Select items to perform actions on them.

Name	Last Modified
☐ examples	hace 2 días
☐ kapre	hace 13 días
☐ srcs	hace 2 meses
☐ CargaHDF5.ipynb	Running hace 2 días
☐ example_codes-withh5py.ipynb	hace 2 días

The use of Jupyter is recommended because it is possible to run experiments easily. It is not necessary to run all the code every time because Jupyter can split the code into cells and each cell can be run independently of the others. Another advantage is that it makes very comfortable to write and test code, as well as visualizing the results which are displayed into charts and graphs or just by using prints below the executed code. For example:



```
In [5]: fig = plt.figure()
        fig.set_figwidth(20)
        fig.set_figheight(2)
        plt.plot(X[0:10*fs],color=(41/255.,104/255.,168/255.)) #Librería matplotlib para mostrar gráficos
        fig.axes[0].set_xlabel('muestra (44,100Hz)')
        fig.axes[0].set_ylabel('amplitud')
Out[5]: Text(0,0.5,'amplitud')
```

Figure 2.1: The graph shows the time evolution of an oscillatory signal.

We use the MusicNet database [11] to train and evaluate our model, downloaded in HDF5 format (see Section 3 for more details). In order to load data from an hdf5 file we use a library called *h5py*. Later on we will explain how the MusicNet dataset is structured and how to access all its data.

In a first instance, all the libraries were installed in a Python virtual environment. This environment is isolated from other python versions and the libraries are installed locally in the machine. The installation commands of the libraries are the following:

- *pip install –upgrade tensorflow*
- *pip install keras*
- *pip install jupyter*
- *pip install h5py*

Initially, due to the fact that the machine used for the experiments had not top specifications, and also because the graphic card was not NVIDIA, it was not possible to use GPU acceleration while training the neural network. Training a neural network on a GPU is highly recommended to speed-up the process, as it is not viable to run many tests on a CPU because it requires too much time.

Thus it was necessary to work in a different environroment in which it was possible to use the same libraries but with higher specifications. Google Colaboratory[1] is a good choice because it provides a free GPU Nvidia Tesla K80 with 13GB of memory and an Intel(R) Xeon(R) CPU @ 2.30GHz. This web application allows to run Jupyter notebook on the server side. It also has a good synergy with the unlimited storage Google Drive account given us by the University of Alicante so it is not a problem to store as many datasets as necessary. On the negative side, Google Colaboratory only allows twelve hours of continued availability and sometimes it does not assign a GPU in the backend.

---

[1] `https://colab.research.google.com/`

Chapter 3

MusicNet data structure

This section describes the data used for training and evaluating the proposed method.

The MusicNet database can be downloaded in three different formats: .h5, .npz, and a compressed .tgz file. The .tgz file contains four folders: *test_data, train_data, test_labels, and train_labels*. The first two contain the audio data, the *train_data* folder contains a larger number of songs to be used as the input data for training a neural network, while the other contains a smaller, non-overlapping set of data for evaluating the trained model. The stored audio files have the extension *.wav*, which is an uncompressed audio format [12]. On the other hand, the *train_labels* and *test_labels* folders contain all the necessary tags for checking the results obtained in the execution of the neural network. These labels are stored as *.csv* files.

It is not advisable to make use of the full dataset when comparing different models as it takes up a large amount of storage (21.73GB in total) and loading all this data can be quite slow.

Instead of using the *.tgz* data, the second possibility is to use the file with the *.npz* extension, which weights 11GB. This format is mainly used to store data that will be processed in machine learning. The data is stored using *NumPy* data structures. It has some limitations while dealing with metadata because *.npz* does not organise properly the data and it can be complicated to use it. Furthermore, data loading is not carried out quickly and optimally.

Finally, MusicNet offers the possibility of accessing your dataset through an. h5 file. HDF5 is a widely used format for large volumes of data as this format is intended to store large amounts of information and then be able to access them easily and quickly. The MusicNet dataset saved in this format occupies 7GB which is a remarkable difference from the 21'73GB of .tgz and the 11GB of .npz.

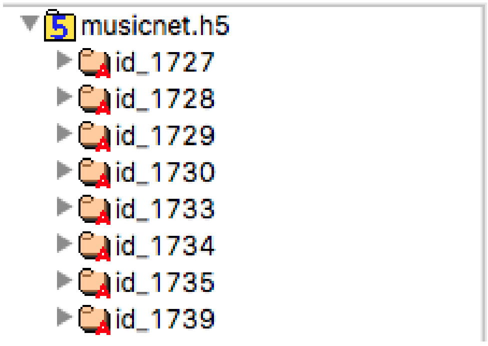
Due to the low storage size and optimized data loading, it was decided to use the MusicNet dataset in HDF5 format. HDF5 has a hierarchical data structure. The data is stored in a similar way to a directory service. There are two main elements which are the datasets and the groups. On the one hand datasets are multidimensional matrices in which all kind of data can be stored, e.g. images, videos, music, coordinates, etc. On the other hand, groups are kind of folders used to classify other groups or datasets.

MusicNet has 330 groups and each group represents a different song. The names of the groups are labelled by the pattern */id_ID* where *ID* is a number to identify the song.

The structure of the MusicNet database is the following[1]:

---

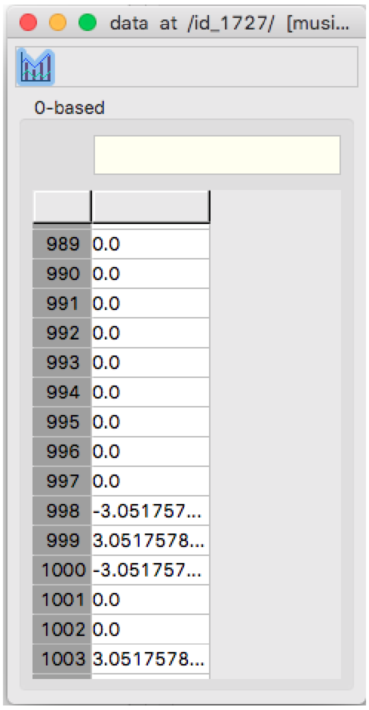
[1] The screenshots of the data structure have been taken using the program "HDF View"



musicnet.h5
id_1727
id_1728
id_1729
id_1730
id_1733
id_1734
id_1735
id_1739

Figure 3.1: 8 first groups of the total 330.

Within each group (which represents a musical song) there are two datasets. The first one represents the audio signal, and consists of an array containing a large number of values. An example of the content of these data can be seen in Figure 3.2.



data at /id_1727/ [musi...

0-based

989	0.0
990	0.0
991	0.0
992	0.0
993	0.0
994	0.0
995	0.0
996	0.0
997	0.0
998	-3.051757...
999	3.0517578...
1000	-3.051757...
1001	0.0
1002	0.0
1003	3.0517578...

Figure 3.2: Audio signal represented by a numerical vector showing the amplitude at each time.

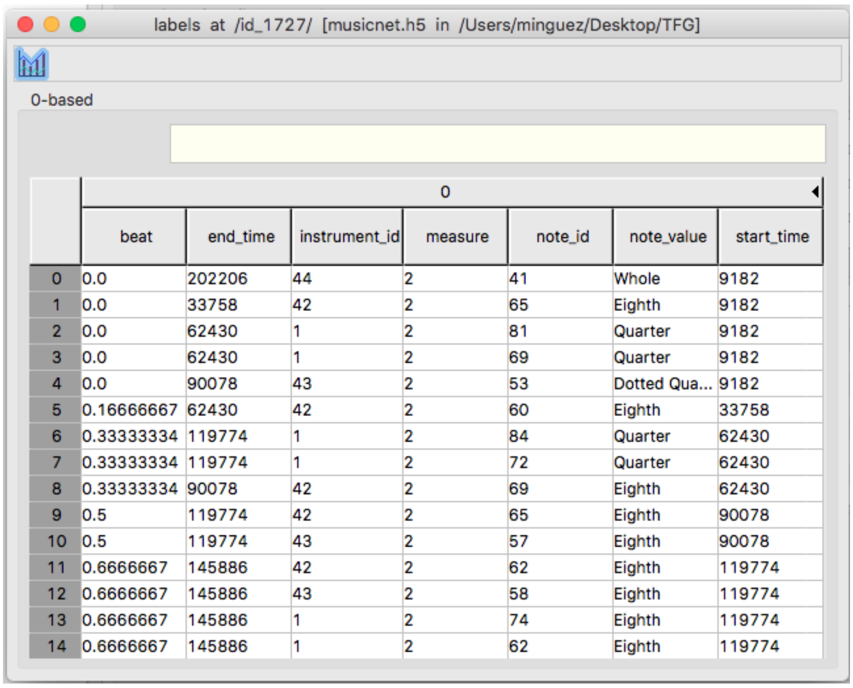
The second dataset contains all the labels of the song. Those labels have valuable information:

- *Beat*
- *End time*
- *Instrument id*
- *Measure*
- *Note id*
- *Note value*

- *Start time*

As we need to recognise the different musical notes, the columns of the labels that are important and are going to be considered are the *start_time*, the *end_time* and the *note_id*. The start and end times are expressed in number of samples. Therefore, in order to obtain the start/end time in seconds it is necessary to divide that number by the sampling rate (in MusicNet, this is 44100Hz).

Basically, if the network is going to learn how to predict musical notes, it is necessary to retrieve information of which musical notes have been played at each instant of time. An example of the label dataset in MusicNet is shown in Figure 3.3.


labels at /id_1727/ [musicnet.h5 in /Users/minguez/Desktop/TFG]
0-based
0

	beat	end_time	instrument_id	measure	note_id	note_value	start_time
0	0.0	202206	44	2	41	Whole	9182
1	0.0	33758	42	2	65	Eighth	9182
2	0.0	62430	1	2	81	Quarter	9182
3	0.0	62430	1	2	69	Quarter	9182
4	0.0	90078	43	2	53	Dotted Qua...	9182
5	0.16666667	62430	42	2	60	Eighth	33758
6	0.33333334	119774	1	2	84	Quarter	62430
7	0.33333334	119774	1	2	72	Quarter	62430
8	0.33333334	90078	42	2	69	Eighth	62430
9	0.5	119774	42	2	65	Eighth	90078
10	0.5	119774	43	2	57	Eighth	90078
11	0.6666667	145886	42	2	62	Eighth	119774
12	0.6666667	145886	43	2	58	Eighth	119774
13	0.6666667	145886	1	2	74	Eighth	119774
14	0.6666667	145886	1	2	62	Eighth	119774

Figure 3.3: Labels dataset.

Loading MusicNet in Python

In order to work with a *.h5* file we use the library *h5py* as previously mentioned in section 2. This library provides methods to access and modify data, as well as obtaining general information such as the total number of groups in the dataset. One of the perks that this library offers is that data is not loaded into memory all at once. Instead, it loads only the fragments of data specified by us. This library depends on *NumPy* because it uses its array types.

Loading the *.h5* dataset in python can be done as follows:

```python
import numpy as np
import h5py
# Loading the Musicnet dataset
hf = h5py.File('../musicnet.h5', 'r')
```

For example, to list all the groups of MusicNet, python provides a method called *list()* so if we use the loaded dataset as a parameter, the result will be an array containing all the groups:

```
all_groups = list(hf) # Content of all_groups: ['id_1727', ..., 'id_2688']
```

This functionality allows us to browse the entire dataset and access any required data.

Once the groups are obtained, it is important to retrieve the stored data of each group. For example, if we need to get all the data of the first song:

```
# To store all the audio data
X = hf['id_1727']['data'].value
# To store the labels associated to the audio
Y = hf['id_1727']['labels'].value
```

None of the datasets from MusicNet will be used directly. Both of them will need a transformation of the data to use it as the input of the neural network.

Chapter 4

Methods and techniques

This section details the data transformations required to train the different deep learning topologies evaluated.

4.1 Input Representation

First, we have to understand how to deal with the data and how can it be converted in order to use it as an input of the neural network.

Working with sound in neural networks is different from dealing with images. Sound is produced by the vibration of a source that is typically propagated through a transmission medium such as the air, water or metals. Therefore, it is produced by compression and decompression of the medium, causing the affected molecules to start colliding with each other and it cannot propagate in the vacuum because there are no molecules that can collide with each other [21].

A sound source can be represented in different ways. The most common is the audio representation in the time domain. An example of it can be the following.

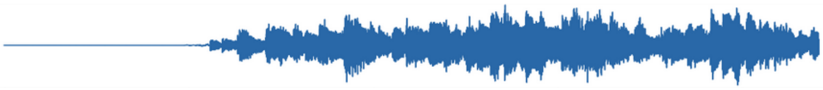


Figure 4.1: Signal representation of the song 1752 of MusicNet

The figure 4.1 sheds light on the evolution in time of the song, and there it is possible to see the oscillation of the signal. The Y axis represents the amplitude while the X is the time. In this kind of function it is not possible to distinguish the music that is actually playing, what music notes are present or other relevant information.

The mathematician Joseph Fourier proved that every periodic signal $s(t)$ can be decomposed as the sum of a number of elementary cosine and sine signals of different frequencies, amplitudes and phases [21]. Fourier was able to convert the sound signal from time to the frequency domain. The representation of the signal as a function of frequency is known as *spectrum*. The spectrum reveals much more relevant information that can be crucial to analyse the audio.

Moreover, music audio signals have to be represented in a way that the computer is able to recognise. This is done by using a discrete vector of numbers. When signals are converted into numbers, time ceases to run continuously so its value depends on the sampling frequency used.

$$t = n \cdot T_s \tag{4.1}$$

where T_s is the sampling period and the t is the time.

In this project, different types of transformations are applied in order to test which one provides better prediction results. There is a study [6] that proves that the choice of the spectrogram highly determines the performance of the neural network for music transcription tasks. Table 4.1 shows the three most influential hyper-parameters in different models and the percentage of variability in performance they are responsible for. In this work, two kind of transforms are going to be computed: the Fast Fourier Transform (FFT) which consists of a fast algorithm to calculate the Fourier Transform, and the Constant-Q Transform which applies a transform in the logarithmic scale. Both transformations are described in the next section.

Model Class	**Pct**	**Parameters**
Logistic Regression	48.6%	Spectrogram Type
	16.9%	Spectrogram Type x Normed Area Filters
	10.4%	Spectrogram Type x Sample Rate
Shallow Net	64.4%	Spectrogram Type
	20.8%	Spectrogram Type x Sample Rate
	5.7%	Sample Rate

Table 4.1: The three most important hyper-parameters determining input representation for two different models. The column Pct is the percentage of variability in performance they are responsible for.

Fast Fourier Transform

The first type of transformation in the frequency domain that is going to be studied is the Fast Fourier Transform (FFT). Before understanding what is the FFT and how it works it is necessary to know first the Fourier Transform. The equation of the Fourier Transform (FT) can be represented as:

$$S(f) = \int_{-\infty}^{+\infty} s(t)\sin(-2\pi f t)dt \tag{4.2}$$

As we can see, in this equation the signal goes from $-\infty$ to $+\infty$, something that is not possible in real scenarios because a sound signal does not last forever. To redress the theoretical issue it is necessary to apply *windows* $w(t)$ where only parts of the signal are computed. The resulting equation will be the following:

$$S(f,\tau) = \int_{-\infty}^{+\infty} s(t)w(t-\tau)e^{-j2\pi f t}dt \tag{4.3}$$

The window is moving forward from the beginning to the end of the signal and the FT equation is going to be applied only to the data of the song that appears in that window. Therefore, the window location and size will determine which data has to be converted to the frequency-domain at that moment.

Then, the *Short Term Fourier transform* (STFT) shows all the concatenated windows which the FT has been computed. In this work, the Fourier Transform at each window will be calculated using the Fast Fourier Transform (FFT) which is a modification of the FT with the advantage that it is calculated faster. The complexity of the standard FT is $O(N^2)$ while the complexity of the FFT is only $O(N \log_2(N))$. This is a significant difference of complexity when large vectors are processed. The only drawback of FFT is that the windows needs to be a multiple of 2 such as 1024 or 2048 in order to work properly.

The FFT algorithm is faster because it is able to decompose recursively the FT into simpler operations until it reaches two element transforms where k can take the values 0 and 1. If the number of samples (N) is a power of 2, the recursion can be repeated until it reaches a situation where only pairs of samples remain. That is why it is necessary to compute the FFT with window sizes that are power of 2 such as N=512, 1024 or 2048.

The succession in time of the computed spectra using the FFT or FT creates a temporal evolution of the signal, called STFT or spectrogram (see Figure 4.2).

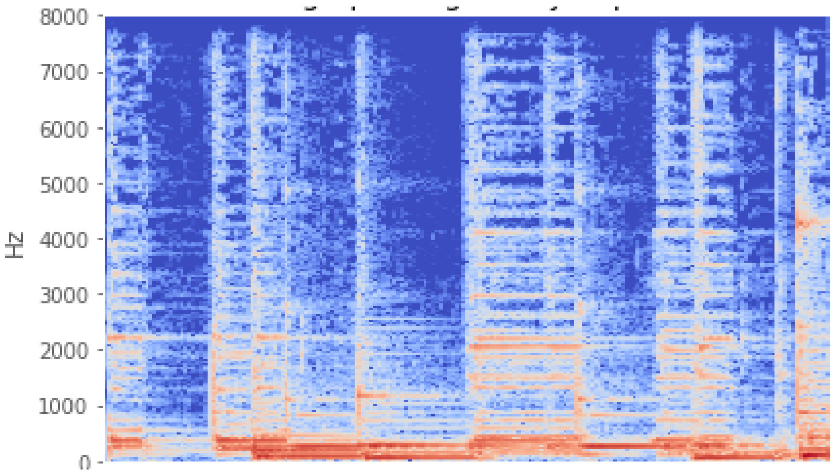


Figure 4.2: Spectogram calculated using the Kapre library for Python.

A spectrogram is a representation of the frequencies of an acoustic signal as it varies in time. It is basically a 3D matrix showing time, frequency and the magnitudes, which are typically represented by different colours. The vertical axis represents the frequency (Hz), which is related to the pitches of the notes. The lower frequencies are at the bottom and the higher at the top [19].

It is possible to distinguish different colours that represent the magnitudes, also called as levels of energy. In Figure 4.2, blue means that there is no energy (low magnitudes) and orange represents the high magnitudes [8].

The main reason to use a frequency representation instead of the original audio signal is because it is easier to distinguish the musical notes using spectrograms, so the neural network can faster process and learn from this data.

Regarding to the windows applied in the transformations, there are different shapes and sizes to take into account while calculating an FFT. Depending on the size of the Fourier analysis window, different levels of frequency/time resolution are achieved. As it was mentioned before, the windows in the FFT must be multiple of 2 to achieve the expected efficiency ($O(N \log_2(N))$). Furthermore, the size also affects to the resolution of the spectrogram, the bigger the window is, the better frequency resolution is possible to obtain, although lowering the temporal resolution It is recommended to find a suitable balance between the frequency and time resolutions in order to get adequate results.

Another aspect to take into account while deciding how the FFT is calculated is by choosing the window hop. At the beginning, the window will start from the time 0 and is going to move through the signal by jumping a number of samples at each shift. This movement of the window can be done using overlapping (considering samples from the previous window) or not.

Not using overlapping is computationally more efficient due to the fact that less data is calculated.

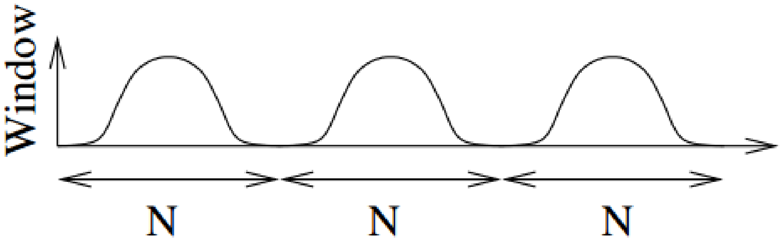


Figure 4.3: Segmented data stream with window and without overlap.

Overlapping consist of jumping less samples that the window size. For example, if the window has 2048 values and the window shift (hop) is 512 means that every jump will take 512 new samples, so the overlap will be 75%.

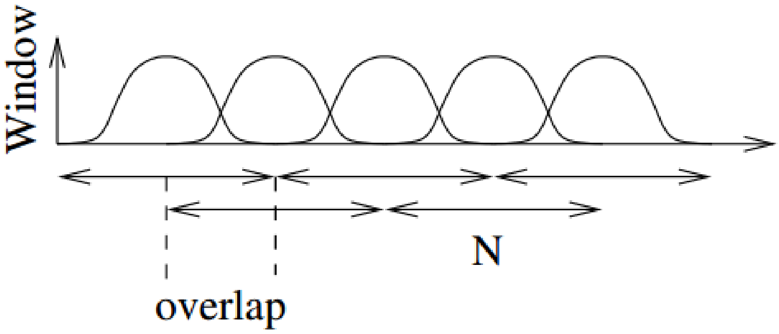


Figure 4.4: Segmented data stream with window and overlap.

Constant-Q transform

Another possibility is to calculate the spectrogram using a logarithmic scale. The Constant-Q Transform (CQT) usually gives a better representation of the audio signal than the FFT. It also transforms a time-domain signal into the frequency domain and, like the FFT, the CQT is a bank of filters but it has geometrically spaced central frequencies.

$$f_k = f_0 \cdot 2^{k/n_b} (k = 0, 1, ...) \tag{4.4}$$

Where f_0 is the minimal centre frequency and n_b the number of filters per octave. The interesting point about CQT is that by using an appropriate number of f_0 and n_b, the centre frequencies can match those from the musical pitches. As the number of bins per octave could have the values $n_b \in \{12, 24, 36, 48\}$, different values will be tested in order to find the most suitable one.

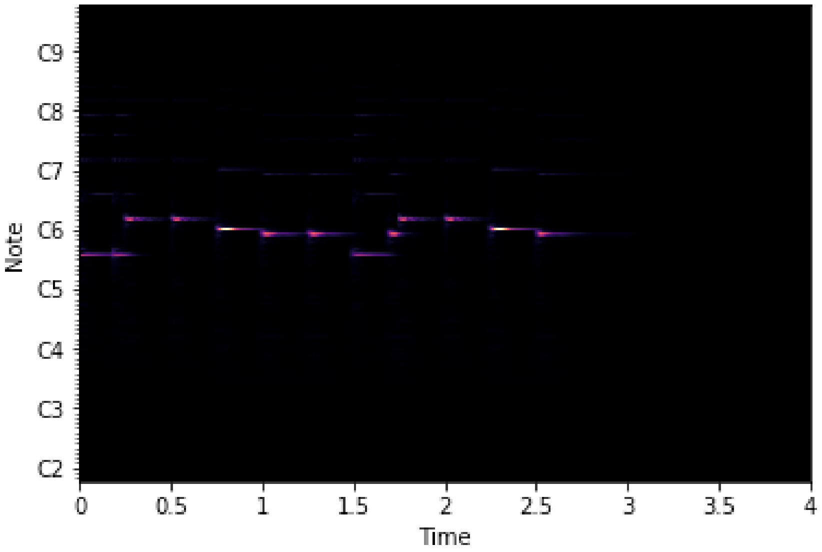


Figure 4.5: Example of a CQT representation. It can be seen that non-black values represent the spectral contents of the musical notes.

To compute the Constant-Q transform, the library *Librosa* will be used. The particular method is called *librosa.cqt* and is based on the recursive sub-sampling method described in [2]. The parameters that can be tuned are:

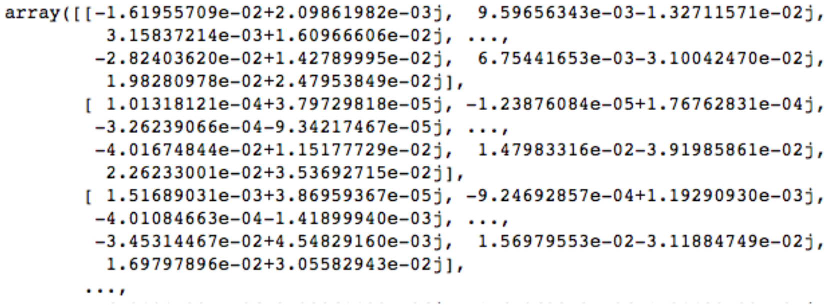
- `y`: Audio signal.
- `sr`: Sampling rate.
- `fmin`: Minimum frequency.
- `n_bins`: Number of frequency bins.
- `bins_per_octave`: Number of bins per octave.
- `norm`: Type of norm to use for basis function normalization.

The result of executing the librosa function is a matrix with the size of the number of windows that can fit in the data provided (y), and the number of bins specified. For example, Figure 4.5 shows the CQT of 5 seconds of a song with a resolution of 384 bins. This figure shows the CQT of a piano song where some musical notes are played sequentially. Thanks to the CQT representation it is possible to distinguish the different notes and its frequencies. In the case of the Figure 4.5, all notes are played in the octave five and six hence, they are close to each other and they have similar frequencies (around 698 Hz and 1174 Hz).

The values of the matrix of the CQT are complex, so it has a real part and an imaginary one. The matrix of the previous song is shown in Figure 4.6.

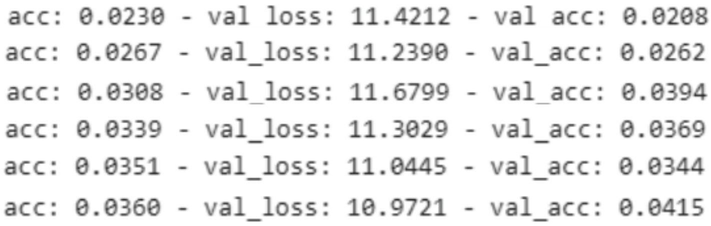
Dealing with complex numbers is more complicated than dealing with real numbers. If we want to store the result in a HDF5 dataset in order to compute the CQT only once, and then use the values from the dataset, it is necessary to create a *Compound Datatype* dataset which has two types of values in each column, the real and the imaginary part. The real part is represented as a r and the imaginary as an i.

An important problem while dealing with complex numbers is that Keras (the framework used for the neural network) does not support them so it will discard the imaginary part while training or testing the model.



```
array([[-1.61955709e-02+2.09861982e-03j,  9.59656343e-03-1.32711571e-02j,
          3.15837214e-03+1.60966606e-02j, ...,
         -2.82403620e-02+1.42789995e-02j,  6.75441653e-03-3.10042470e-02j,
          1.98280978e-02+2.47953849e-02j],
        [ 1.01318121e-04+3.79729818e-05j, -1.23876084e-05+1.76762831e-04j,
         -3.26239066e-04-9.34217467e-05j, ...,
         -4.01674844e-02+1.15177729e-02j,  1.47983316e-02-3.91985861e-02j,
          2.26233001e-02+3.53692715e-02j],
        [ 1.51689031e-03+3.86959367e-05j, -9.24692857e-04+1.19290930e-03j,
         -4.01084663e-04-1.41899940e-03j, ...,
         -3.45314467e-02+4.54829160e-03j,  1.56979553e-02-3.11884749e-02j,
          1.69797896e-02+3.05582943e-02j],
        ...,
```

Figure 4.6: Matrix resulting from the calculation of the CQT.



```
acc: 0.0230 - val loss: 11.4212 - val acc: 0.0208
acc: 0.0267 - val_loss: 11.2390 - val_acc: 0.0262
acc: 0.0308 - val_loss: 11.6799 - val_acc: 0.0394
acc: 0.0339 - val_loss: 11.3029 - val_acc: 0.0369
acc: 0.0351 - val_loss: 11.0445 - val_acc: 0.0344
acc: 0.0360 - val_loss: 10.9721 - val_acc: 0.0415
```

Figure 4.7: Model trained with the FFT input data.

In order to get the magnitude given the complex values, the *Magphase* is computed. This is a Librosa method which transforms a complex-value spectrogram into its magnitude (S) and phase (P) components. The magnitude part will contain only real and positive numbers and will become the input of the neural network.

Comparison between CQT and FFT

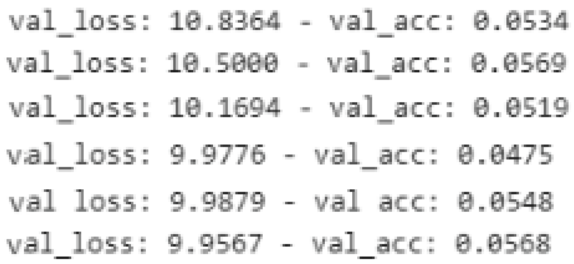
In this section we test whether using CQT provides better accuracies than using the FFT. In both scenarios, the same amount of songs will be used, which does not mean the same quantity of data, as CQT and FFT do not have the same window length. In case of the FFT, the dataset containing the data of the 20 first seconds of each song and its labels weight 3.5 GB while the same songs and labels computed using the CQT weight 300 MB. The tests are not going to obtain good results in accuracy because the neural network is going to be trained with a few number of samples, the importance is the different results between the two transforms.

The fact that the Constant-Q Transform needs less memory space helps the neural network to train using the same quantity of data in less time. Besides, the fact that more data can fit in the main memory speed-ups the training process. The results of applying the FFT are shown in the Figure 4.7.

In this experiment, the time to compute each epoch is about 60 to 70 seconds. The accuracy grows steady but really slow, around a 0.40% of improvement of the accuracy each epoch, which leads us to a 4% of accuracy in the last epoch.

On the other side, the CQT has similar results but taking into account that it requires much less memory. The prediction results show a higher accuracy (around a 6%). And it takes only 20 seconds to compute each epoch (see Figure 4.8).

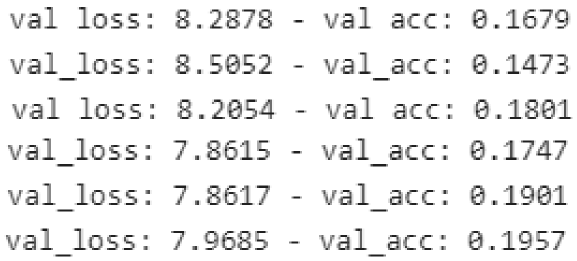
Thanks to the fact that the CQT needs less number of values to represent a spectrogram which contains a similar valuable information about a song, it is possible to load in the main



```
val_loss: 10.8364 - val_acc: 0.0534
val_loss: 10.5000 - val_acc: 0.0569
val_loss: 10.1694 - val_acc: 0.0519
val_loss: 9.9776 - val_acc: 0.0475
val loss: 9.9879 - val acc: 0.0548
val_loss: 9.9567 - val_acc: 0.0568
```

Figure 4.8: Model trained with the CQT input data.

memory more samples than using the FFT. For example, the experiment showed in the Figure 4.9 tests what accuracy is possible to obtain after training a neural network using the same allocated memory that has been required to do the experiment shown in the Figure 4.7.



```
val loss: 8.2878 - val acc: 0.1679
val_loss: 8.5052 - val_acc: 0.1473
val loss: 8.2054 - val acc: 0.1801
val_loss: 7.8615 - val_acc: 0.1747
val_loss: 7.8617 - val_acc: 0.1901
val_loss: 7.9685 - val_acc: 0.1957
```

Figure 4.9: Model trained with the same quantity of data as the FFT.

The prediction accuracy reaches almost the 20%, five times better compared to the FFT result.

4.2 Labels

A supervised machine learning model needs the data to be labelled. As it was explained above, the input song is going to be represented in the frequency domain, but the labels provided by MusicNet are no longer viable because they show all the musical notes that are played in a song, but they do not state which musical notes belong to which particular window.

The neural network needs the training songs and their associated note labels. The input data is represented as *NumPy* arrays containing the FFT or the CQT (depending on the chosen transform). Additionally, the output (labels) will be arrays showing the played notes represented with a *one-hot encoding*.

One-hot encoding is a widely used technique in neural networks which consists of creating an array of boolean values. In our case, every column symbolises a possible musical note that can be played in a certain moment, so there must be as much columns as different musical notes. Therefore, the objective is to mark which notes are played in the windows using the numbers 0 or 1. When a cell of the array has the number 1 in it, means that a specific musical note has been played in that window. In this work, there are going to be 88 possible musical notes that can be played at the same time (all the existing notes) hence, the one-hot encoding vector will have a length of 88 values. For example, lets suppose the song showed in the Figure 4.10 has only musical notes that belong to the octave presented in the Table 4.2. The resulting one-hot (See Table 4.3) will be a vector in which each column corresponds to each musical note that can be played. The row # corresponds to a moment in the song and the elements 0 and 1 mean whether that note is being played at that moment or not.

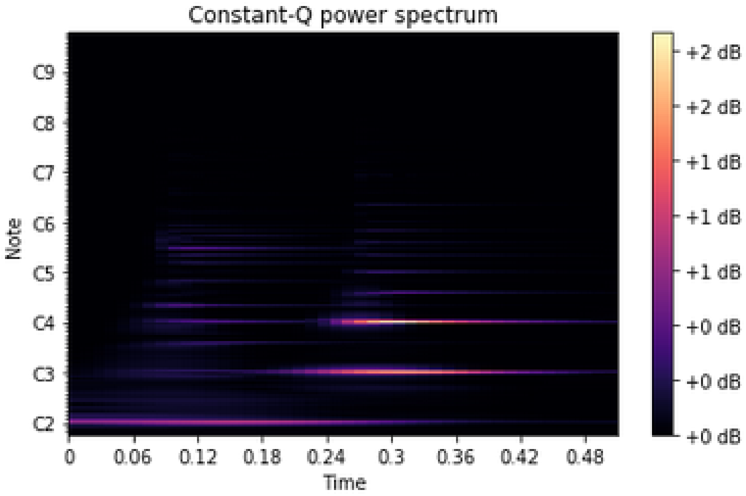


Figure 4.10: Constant-Q Transform of a piano song where the C3 and C4 notes are played.

#	Note ID
0	C3
1	D3
2	E3
3	F3
4	G3
5	A3
6	B3
7	C4

Table 4.2: A list of some musical notes that can be played.

Window	C3	D3	E3	F3	G3	A3	B3	C4
#1	1	0	0	0	0	0	0	1

Table 4.3: Resulting one-hot encoding vector.

In this work, to construct the one-hot encoding it is necessary to analyse each label of Musicnet and transform the data that has been provided into one-hot encoding vectors.

The first step is to get the vector containing the input labels of the neural network. As mentioned before, the array has to contain as many elements as different musical notes that can appear in a song. The outputs are the possible activations of the 88 notes considered. A one-hot array encodes only one window, so for each song it is necessary to create the same number of one-hot arrays than the number of windows.

It is necessary is to initialise a matrix containing all the one-hot vectors that are going to be required to label each song.

```
number_musical_notes = 88
one_hot = np.empty([int(seconds*wps), number_musical_notes])
```

The initial state of the one hot encoding array will be an array with zeros which means that there are not musical notes played.

```
for i in range(one_hot.shape[0]):
        one_hot[i] = 0 #initializing the array to zeros
```

Before going though all the labels it is important to understand what role each label plays and its relationship to the different windows. Firstly, a song has many labels associated, and each one has an start time and end time. That period of time is represented in a set of windows of the song. In other words, a label (a note in this case) can be played in a set of windows, so to create the one-hot encoding it is necessary to make sure that those windows have the value 1 in the position that represents that musical note.

In order to calculate the one-hot encoding for all the songs, a function is created that goes through all the data of the labels dataset of MusicNet. What it does is to go label by label and analyse the start and end times of each musical note. We calculate the time as mentioned in the *Dataset MusicNet* chapter, dividing the number of the start/end time by the sampling rate, in this case 44100 (Hz).

Once the times (in seconds) are obtained, it is possible to compute in what windows the notes are played. To calculate that range of windows where a specific note has been played we use the following formula:

$$window = (f_s/stride) \cdot t \tag{4.5}$$

Where f_s is the sampling rate, *stride* is the number of samples between windows, and t is an number which represents the time in seconds of the current label.

This formula has to be applied both for the start and end times. The result will be two variables that determine the range of windows where the music note appears. Thus, the next step is to change from 0 to 1 the note in those windows where it appears. For example, if a musical note has an start time of 1 second and an end time of 3 seconds, then it goes from the window 86 until the window 258. Hence, the column corresponding to that note is changed to 1 in this range of windows.

Finally, in order to increase the efficiency, all the one-hot encodings will be stored in a HDF5 file. This way, in order to train the neural network it will not be necessary to calculate the labels every time, just loading the results from the .h5 file. Furthermore, the transforms of each song will be stored in another dataset because of the same reason, to speed up the process. The HDF5 dataset will have an structure similar to that from Musicnet. There is going to be a set of 330 groups that represent every song of MusicNet. Each group will contain two different datasets. The first one is will be the transformed data and the second one the one-hot encoding matrix (see Figure 4.11 for an example of the structure of the resulting dataset).

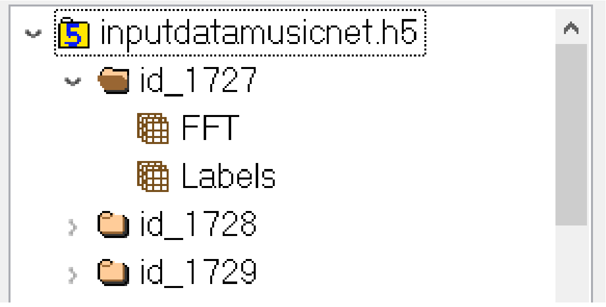
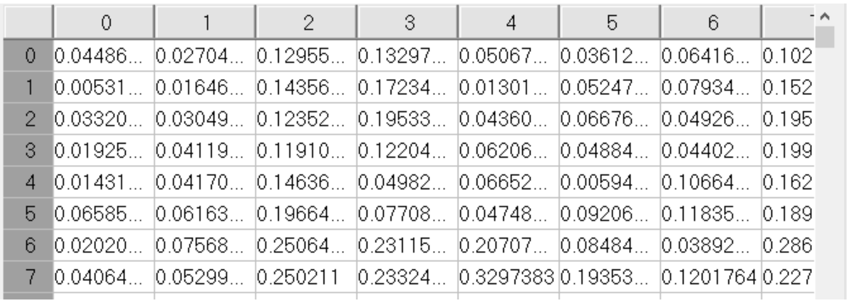


Figure 4.11: There is a group per each song. Inside a group there are two data, the first one contains the spectrogram of the song and the second one contains the one-hot encoding arrays of each window of the song.

As there two different time-frequency transformations that are going to be evaluated, there will be two types of data for a song: The Fast Fourier Transform (labelled as as 'FFT') and the Constant-Q Transform ('CQT').

The FFT data has 2048 columns which represent each frequency value of that window. The number of rows is determined by how many windows are computed in the song. For example, if it is needed to store the first 10 seconds of a song, it will be necessary to create the FFT dataset with 861 rows. To calculate how many rows have to be stored depending on the size of the song, we can use the formula 4.5.

An example of a resulting *FFT* data for a song:



	0	1	2	3	4	5	6	
0	0.04486...	0.02704...	0.12955...	0.13297...	0.05067...	0.03612...	0.06416...	0.102
1	0.00531...	0.01646...	0.14356...	0.17234...	0.01301...	0.05247...	0.07934...	0.152
2	0.03320...	0.03049...	0.12352...	0.19533...	0.04360...	0.06676...	0.04926...	0.195
3	0.01925...	0.04119...	0.11910...	0.12204...	0.06206...	0.04884...	0.04402...	0.199
4	0.01431...	0.04170...	0.14636...	0.04982...	0.06652...	0.00594...	0.10664...	0.162
5	0.06585...	0.06163...	0.19664...	0.07708...	0.04748...	0.09206...	0.11835...	0.189
6	0.02020...	0.07568...	0.25064...	0.23115...	0.20707...	0.08484...	0.03892...	0.286
7	0.04064...	0.05299...	0.250211	0.23324...	0.3297383	0.19353...	0.1201764	0.227

Figure 4.12: Excerpt of the first 10 seconds of a song, containing 861 rows (windows) and 2048 columns (FFT frequency bins).

The CQT data is similar to the FFT, the only difference is the size of the columns. Instead of having 2048 values, it will have the specified number of frequency bins.

Regarding to the labels data stored in each group, each row represents a window and in this case, every column stands for a musical note. If this matrix contains a 0 that means that the musical note has not been played, while 1 stands for the opposite, as can be seen in Figure 4.13.

To obtain the data described below, it is necessary to follow these steps:

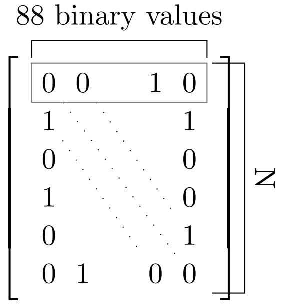


Figure 4.13: A matrix containing the one hot vectors of each window. Each row represents the corresponding one-hot vector of an associated window. The number N the total number of windows.

1. The first step is to import the libraries which are going to be used to create the .h5 file and to compute the FFT or CQT, and initialise some variables that are going to be used as well as the HDF5 file which will contain 330 groups, one for each song of MusicNet. The variables required are:

    - The size of the window size and the stride.
    - The seconds computed of each song.
    - The sampling rate.
    - A NumPy array which the FFT values will be stored.
    - A NumPy array which contains the one-hot encoding labels.

    And the libraries:

    - import numpy as np
    - import h5py
    - from scipy import fft
    - import librosa
    - from os import listdir

2. The next step is to compute the CQT or FFT following the indications described in Section 4.1. Once the values are transformed, the results have to be saved into the HDF5 dataset.

3. Finally, the last step is to check what musical notes are played in each window, saving the results from the one-hot encoding vectors in the *Labels* dataset.

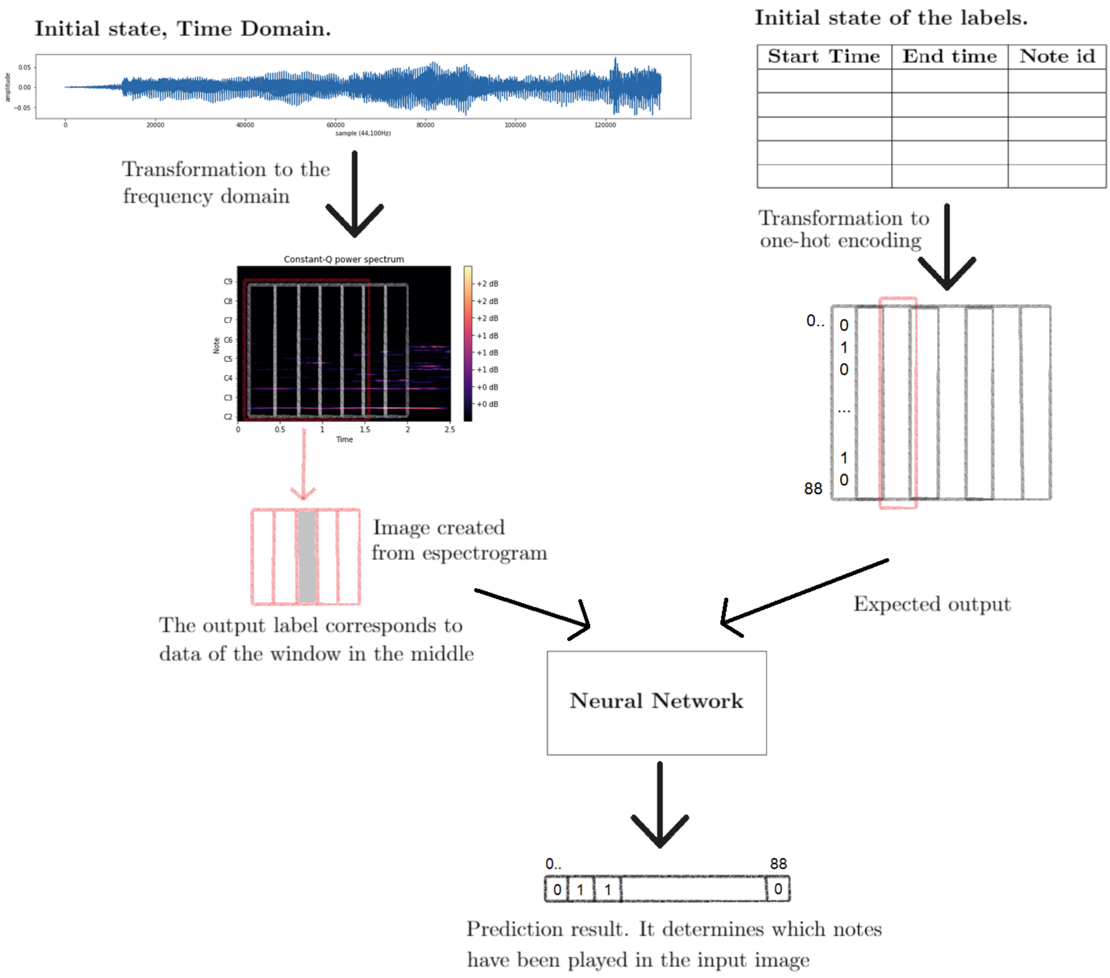

Initial state, Time Domain.
amplitude
0.05
0.00
−0.05
0
20000
40000
60000
80000
100000
120000
sample (44,100Hz)
Transformation to the frequency domain
Constant-Q power spectrum
Note
C9
C8
C7
C6
C5
C4
C3
C2
0
0.5
1
1.5
2
2.5
Time
+2 dB
+1 dB
+0 dB
Image created from espectrogram
The output label corresponds to data of the window in the middle
Initial state of the labels.
Start Time
End time
Note id
Transformation to one-hot encoding
0..
88
0
1
0
...
1
0
Expected output
Neural Network
0..
88
0
1
1
0
Prediction result. It determines which notes have been played in the input image

Figure 4.14: Overall diagram for data preprocessing in the proposed method.

The overall diagram of the data preprocessing stage is shown in Figure 4.14.

4.3 Convolutional neural networks

The first challenge of running a neural network is to adapt the input data.

In this work, spectrograms are going to be used as if they were images. Those images can not be rotated or flipped because sound is not symmetrical [3]. In a first instance, a spectrogram has only one row and 2048 columns length when using the FFT data. To create a sort of image, spectrograms are going to be gathered 5 by 5 and then those new images will be used as the input data of the neural network model (see Figure 4.15).

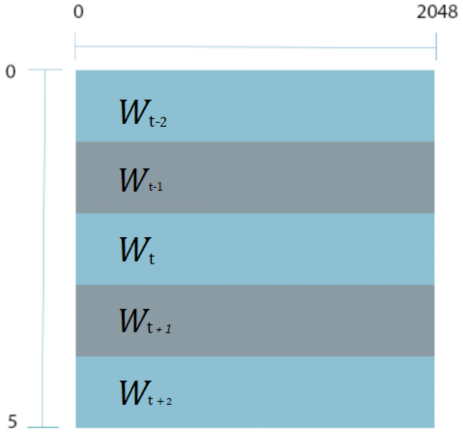


Figure 4.15: Input data of the first convolutional layer, an image of size 5x2048.

The expected output (the labels) of the network will be the one-hot encoding of the centre window W_t, as shown in Figure 4.15. The advantage about putting 5 windows together is that the neural network will have some context information that may be relevant, but it will also learn by itself that the main features are represented in the middle window and therefore it is expected that it will focus on the details described in that window.

In order to avoid memory problems, instead of loading each window every time to create the images, pointers are going to be used. As there are four windows that are repeated from the previous one, that data will reference the window that is actually allocated. Thanks to the use of pointers it is possible to load five times more data than loading all the windows every time.

Convolutional Neural Networks (CNN or ConvNets) are going to be evaluated in order to determine if they yield a successful accuracy. CNN have been very effective in areas such as image recognition and classification [27]. The data that represents the audio is stored into a 2-dimensional array as an image, so it is plausible to obtain reliable results using this kind of networks because CNN are intended for data arrays which are typically 2 or 3 dimensional.

To understand why should we use CNN, it is necessary to know how they work and why do they provide good results comparing with other architectures.

CNN match parts of the image rather than the entire picture. For example, a classical song will have similar music records as other classical songs, while it will be different from Rock music. But if the computer needs to compare two classical songs as a whole, both songs will be different. The point of the CNN networks is that they focus on the small details which determine if an input is classified as one class or another. In this work, the objective is the same. Musical notes have different patterns and the same musical note of two instruments will have similar frequencies. In case that the neural network would also analyse different instruments, it should be able to distinguish between different harmonic spectra.

The following explanation shows how a CNN works step by step until the model weights are learned. Consider for example that our network is intended to analyse an image and predict what is the object in the picture. The CNN goes through the image step by step in jumps of the filter's size. For example, if the filter is 3x3 the filter will start at the top-left side of the input image and will compare those 3x3 values with the expected output by multiplying their values. The resulting number is divided by the total number of values in the filter (in this case, 9) and placed into a new matrix which will contain in every pixel how much that feature matches with the expected image. The elements of the matrix that has strong matches will have values close to 1 and the ones that does not match of any sort will be close to 0. The values close to -1 show strong matches for the photographic negative of the feature.

The neural network repeats this process with all the features that the input image has, obtaining a new matrix for every feature that has been analysed.

The next step is pooling, which is a sample-based discretization process to down-sample an input representation (the hidden layer output in this case) by reducing the size of the matrix and obtaining significant values from it. Pooling can be either Max, Average, etc. What MaxPooling does is to apply a *Max* filter, which goes through the matrix preserving only its maximum value. The size of the filter can differ and should be not too big to loose other relevant information and not too small because it will store information which is not significant. The resulting matrix has only the maximum values that the filter has analysed [28].

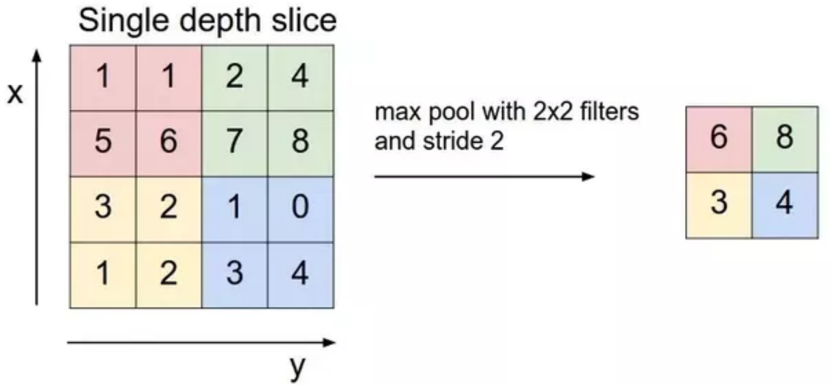


Figure 4.16: MaxPooling process in a 4x4 matrix.

Applying a pooling to the output of a Convolutional layer is recommended in order to reduce the number of parameters to learn and also to prevent over-fitting.

After the Max Pooling is applied to the matrices, the next step is to proceed with the activation functions. There are different types of activation functions but the most common ones are *Sigmoid, Tanh and ReLU*, as explained next.

- The mathematical form of a *Sigmoid* function is:

  $$f(x) = \frac{1}{e^{-\beta x}} \tag{4.6}$$

  This activation function has two problems which makes it to be not frequently used. The first one is that when a neuron activation is close to zero or one, the gradient at these regions is close to zero. This problem will make the gradient slowly vanish and no signal will flow through the neuron. The second problem is that the output is not zero centred, that means the number in the middle is not zero but 0.5. It starts from zero and ends at one making that the gradient of the weights become either all positive or negative. Therefore the gradient updates go too far in different directions, which reduces the optimization.

- The second type of activation function is *Tanh.* This activation function squashes the output between -1 and 1 (which prevents from the non zero-centred value). Its mathematical function is the following:

$$f(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{4.7}$$

Nevertheless, the Tanh function has the same vanishing problem of Sigmoid.

- The last and most common activation function nowadays in deep neural networks is the Rectified Linear Unit (ReLU). $ReLU$ is base on obtaining the maximum value between zero and the value of the matrix. When the values of the matrix are negative, the function will result in a zero and if the value is greater than zero, the function will result as the value of the matrix. The mathematical equation is as follows:

  $$f(x) = \begin{cases} 0 \text{ for } x < 0 \\ \text{x for } x \geq 0 \end{cases} \tag{4.8}$$

  The good point about ReLU is that it does not have the problem of vanishing gradients and it does not require expensive operations so it learns faster [14].

The last part of a Convolutional Neural Network is usually a Fully Connected layer. This kind of layer treats every input as an individual and every neuron is considered identically as the rest. It is a cheap way of learning non-linear combinations of the features. The output of the convolutional layers represents high-level features in the data and the Fully Connected layer receives that data and turn it into predictions. In some cases, adding additional Fully Connected layers lets the network learn more sophisticated combinations of features that will help to make better decisions.

4.3.1 Optimizers

Optimizers are the tool to minimise loss between prediction and real value. Keras provides the possibility of using many different optimizers. Every neural network is different from the others, so sometimes it is adequate to use one type of optimizer and sometimes other. Keras allows to use 8 types of optimizers but the most important ones are: Stochastic Gradient Descent (SGD), RMSprop, AdaGrad, AdaDelta, and Adam.

There are different possibilities provided by Keras. The size of the dataset is directly connected to which optimizer is recommended to use. For small datasets (e.g. 10.000 samples) or datasets without strong redundancy, it is recommended to use a full batch methods. Otherwise, for large datasets with data redundancy, the use of mini batches usually works properly such as gradient descent with momentum or RMSprop. Nevertheless, all neural nets differ from the others and there is not a single and simple recipe for each one. That is why it is important to evaluate the neural network with different optimizers and compare their results.

In this work, different optimisers have been tested and the two optimisers that obtain better results were Adam and RMSprop, being RMSprop the best one when the learning rate was set to 10^{-4}.

4.3.2 Overfitting

Overfitting is a common problem while dealing with large amounts of parameters. This issue refers to exceeding some optimal size of the hidden layer. It was shown in [25] that predictive ability of a neural network decreases substantially if the number of neurons in the hidden layer increases. What happens is that the neural network tends to memorize the input instead of learning from it. A symptom of overfitting if when the train accuracy is much higher than the test accuracy. This means that the neural network is learning specific data and when it receives new data that has not seen before, it is not able to predict it successfully. [9].

Pooling layers are one of the main components to keep the overfitting low. Furthermore, another essential technique to prevent this problem is to use Dropout. This technique consists of dropping out temporally some units in a neural network. This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updated are not applied to the neuron on the backward pass [1]. The choice of which units to drop is random. Nevertheless it is possible to specify the probability of a neuron being dropped, it has been proved [5] that dropping out 20% of the input units and 50% of the hidden units was often found to be optimal.

It is important to remark that Dropouts are only used during the training of a model and they are not taken into account in the test stage.

To prove that the techniques previously mentioned actually work, some images are shown with a simple experiment. The first image (Figure 4.17 shows a CNN which has Dropout and pooling layers in order to avoid having too much input data into the hidden layers. The second image (4.18) shows the same neural network but without using those layers. On the one hand, the expected result should be that the first image has a validation accuracy similar to the validation accuracy. On the other hand, the second network should not learn from the data and tend to memorize it so the accuracy in the training part should be as the first image but then the validation accuracy has to be really low because it will try to predict data that has not seen before and it is not able to do it properly.

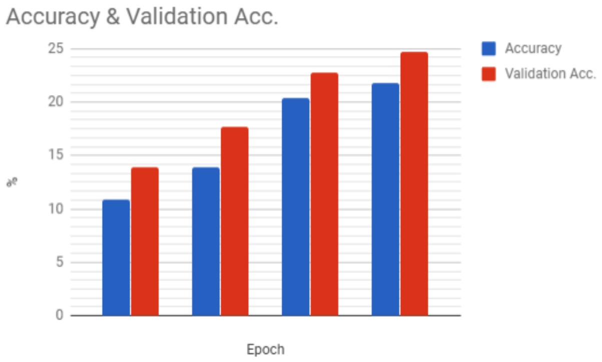


Figure 4.17: Keras model with dropout and pooling layers.

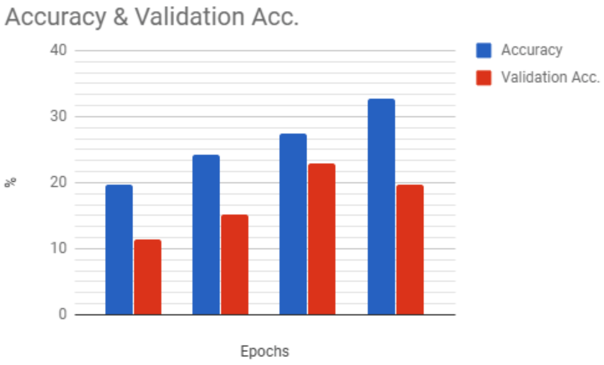


Figure 4.18: Keras model without dropout and pooling layers.

4.3.3 Underfitting

Underfitting a model leads to the same problem of overfitting which is the low accuracy prediction on the unseen test set [7]. It indicates the impossibility of identifying or obtaining correct results due to lack of sufficient training samples or very poor training. The problem of underfitting is that the model is trained with a set of features that are not expressed in the test data set. Thus, when the neural network wants to predict what features are in the testing set, it cannot distinguish them because it is the first time it sees those new characteristics, as it was not able to generalise the data from the training set [18].

In case of Musicnet, it is necessary to find a training and testing sets that have the similar musical notes in common or at least, having a training set that has diverse types of samples, which will prevent from overfitting and underfitting.

4.4 Fit Generator

The Musicnet dataset has 7GB of data. This seems not to be a very large value but, after applying the FFT or CQT and after calculating the hot encoding on the labels, the newly created dataset weighs much more.

One of the problems while dealing with large amounts of data is the memory space. A computer is not capable to store many gigabytes of data at the same time because there are limited resources. To avoid that problem, Keras provides the Fit Generator which brings an opportunity to deal with data more efficiently. The aim of the fit generator is to generate a dataset on multiple cores in real time and feed it right away to a deep learning model [23]. What python and Keras do is to not load all the data at the same time but to do it step by step freeing up memory that is no longer being used. To understand how Keras use the fit generator, it is important to know how generators work in python.

Generator functions allow us to declare a function that behaves like an iterator that it is possible to iterate over once [24]. The concept is simple, it is a function that instead of returning a list of values, it yields a value which is the next one that is going to be returned. Hence the computer will not store all the results at once and it will store only the value that is going to be used at the moment. For example the following function creates an iterable object which each step will return a new number from 0 to N.

```
 def firstn(n):
    num = 0
    while num < n:
        num += 1
        yield num

list_n = firstn(20)
next(list_n)
```

Each time we execute *next*($list_n$), python will go to the first n object and will execute the code in it, returning the next value that is yielded. In this example, at the beginning the first value will be 1 and if we execute the next ($list_n$) again the result will be 2 and so on. The state of the iterable object remains between the executions of the method *next*. Once the iterable object has gone through all possible values, it throws an *Stop Iteration Error* if we want to get a new value.

The computed values in the generators can be displayed by printing them using a loop if we want to get more than one result or typing *next()* if we want to get just the next value of the generator, which returns the next yielded value. The main benefit of generators is encapsulation, which provides new and useful ways to package and isolate internal code dependencies. Furthermore, using iterators tremendously reduce memory footprint, improve scalability, and make the code more responsive to the end user [16].

Keras use the same concept to load the data in order to train and test a neural network. If the project did not use the generators it would be impossible to load as much information into memory without having problems. That is why it is necessary to load the data stepwise in case we want to train the model with all the data that MusicNet provides.

Our generator has only one argument, the batch size. This parameter represents how many songs are going to be stored at the same time in the input data array.

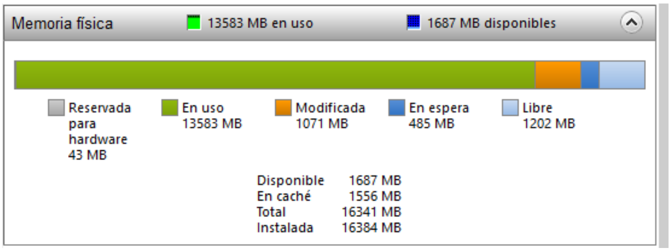
There are two generator functions, one for the training data and the other one for the validation set. Both have the same structure, the only difference is the data they collect. The generator functions consist of a infinite while loop which only finishes when the function has gone through all the data of the dataset. When that happens, the function will throw a *Stop Iteration Error* because it is not possible to iterate anymore. Thus, the values determined in the `fit_generator` call are very important to avoid throwing the error by mistake. It has to be taken into account how many values will be computed in every batch and how many steps are going occur as it is going to be explained below.

Keras fit generator also allows to shuffle the order in which the samples are fed to the classifier, which is helpful so that batches between epochs do not look alike. This way the neural network is able to learn data in a more arbitrary way making the model more robust.

The difference in memory location is significant when using the generator as opposed to not using it. Figures 4.19 and 4.20 reveal the difference of the same neural network trained with the same amount of data:

The green bar represents the memory in use and the light blue one, the free memory. It is also possible to see that in the first image that the computer has to swap between the main and the secondary memory in order to load all the required data which slows down the process.

The problem when using generators is that it is not possible to obtain the maximum value of all the songs, so it is not possible to normalise the data. This happens because songs are loaded progressively and the ones which are already used are discarded. Hence, to know the maximum value of the songs it is required to have the data at the same time, something that does not

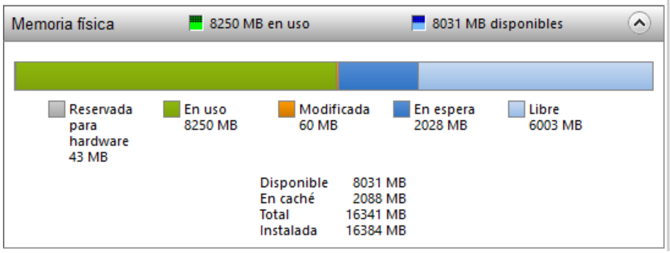


Memoria física 13583 MB en uso 1687 MB disponibles

Reservada para hardware 43 MB
En uso 13583 MB
Modificada 1071 MB
En espera 485 MB
Libre 1202 MB

Disponible 1687 MB
En caché 1556 MB
Total 16341 MB
Instalada 16384 MB

Figure 4.19: Memory use without using the `fit_generator`.



Memoria física 8250 MB en uso 8031 MB disponibles

Reservada para hardware 43 MB
En uso 8250 MB
Modificada 60 MB
En espera 2028 MB
Libre 6003 MB

Disponible 8031 MB
En caché 2088 MB
Total 16341 MB
Instalada 16384 MB

Figure 4.20: Memory use using the `fit_generator`.

happen with generators. In order to fix that problem it is required to compute in advance the maximum value of all the songs. Once that value is stored, it has to be saved and used in the generator function.

Therefore, a python script has been created to load all the songs (training and testing) and return their maximum amplitude value. Once this process is done, the returned value can be saved and used at a later stage and it will no be required to load all the songs anymore.

Chapter 5

First CNN Topology

This first neural network has been created with the aim of obtaining some initial results. In addition, it has served to learn how a model developed with Keras behaves and what characteristics are important to be taken into account.

5.1 Input Data

In this first approach, the FFT is applied to transform the input signal to the frequency domain. The resulting data will be used as the input of the neural network. The transform has a 75% of overlap and it has windows of 2048 values. Therefore, the input shape is a 2-D matrix of 5 by 2048 pixels.

5.2 Proposed Architecture

In the first instance, an architecture similar to the Keras MNIST is evaluated. Since this neural network has been created to predict handwriting digits, the input and output data differ completely from the network that has to be used, but this is a standard architecture for image recognition with small patches. The objective of this neural network is to deal with the problem when creating a simple network and learn to solve it. Besides, the objective is not to create an optimal network that is able to obtain a good prediction results but to perform a first approach that deals with the input data.

The architecture is shown in Table 5.1. It has a basic structure, consisting of eight layers. The first one is a convolutional layer. In this case, the input shape is (5, 2048,1), where the first two dimensions represent the input matrix and the third (1) is the number of channels. This last dimension is specified because Keras is using Tensorflow as the backend, which uses *channels last*. Otherwise, if Keras would use Theano as the backend, the channels should have to be at the beginning of the call by default.

The next layer is also convolutional and it has the same hyper-parameters: 64 filters, Kernel size of 3x11 and a *ReLu* activation function.

Then a pooling layer is added with size 1x2. That explains why the output shape has half the values: Instead being (None, 1, 2028, 64), it is (None, 1, 1014, 64).

Finally, a flatten layer is applied in order to use dense layers. The first dense layer uses the *ReLu* activation function and 128 neurons while the last dense layer has 88 neurons because it is the total number of musical notes that can be played. Between those layers, drouputs have

Layer (type)	**Output Shape**	**Param #**
Conv2D	(None, 3, 2038,64)	2176
Conv2D	(None, 1, 2028, 64)	135232
MaxPool	(None, 1, 1014, 64)	0
Dropout (0.25)	(None, 1, 1014, 64)	0
Flatten	(None, 64896)	0
Dense	(None, 128)	8306816
Dropout	(None, 128)	0
Dense	(None, 88)	11352
		$\sum$ 8,455,576

Table 5.1: Architecture of the first neural network proposed.

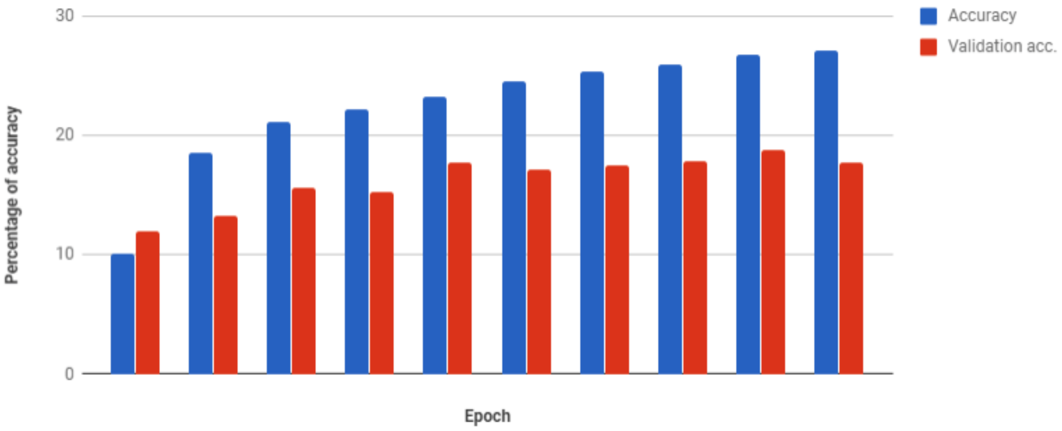


Figure 5.1: Evolution of the accuracies from the first epoch to the tenth.

been used to avoid overfitting. The first dropout layer has a chance of 25% of deletion and the last one a 50% of chance.

This architecture does not provide a state-of-art results but they are not too bad either. After testing with the first minute of 50 songs to train and 20 to test, the architecture reaches a 17% of validation accuracy while it loads almost 12GB in the main memory. With additional epochs it reaches the accuracy mentioned but then the improvement does not make great advances. While the training accuracy keeps improving, the validation accuracy remains (see Figure 5.1).

Chapter 6

Second CNN topology

The previous CNN based on the MNIST image recogniser from Keras and using FFT does a relatively good work but not as good as it could be. A second neural network is proposed by changing the representation of the input data and the architecture of the CNN. Further studies will prove whether this second CNN is better or not.

6.1 Input Data

On this occasion, instead of applying a FFT, the Constant-Q transform (CQT) is going to be used in order to have a lower dimensionality at the input. Furthermore, one of the main advantages of using the CQT is that its Q-Factors are all equal and also the centre of the frequency bins are geometrically spaced. So that, it is better suited for the analysis of music than the FFT.

The CQT will have the following parameters in this second approach:

```
fmin = librosa.note_to_hz('A1')
C = librosa.cqt(y, sr=44100, n_bins=384, bins_per_octave=48, fmin=fmin, norm=1)
```

The first parameters are crucial for determining the shape of the input data. A piano has 8 octaves, so if we specify that we want to have 48 bins per octave, the total number of bins will be 384. Hence, each region of the piano (octave) will be represented with 48 numbers.

In this experiment, the number of bins per octave will be 48 but there are also other alternatives such as using 12, 24 and 36. As it was mentioned in the section *Neural Network*, the input representation is one of the most important parts to be taken into account when designing a neural network. The success of the prediction is highly related to how the input data is represented.

Moreover, we are going to specify that the minimum frequency will be 55Hz (which represents the *A1* pitch on a piano). Below this frequency, the neural network is not going to be able to recognise the notes because the spectrogram will not show their fundamental frequency.

6.2 Proposed Architecture

In this second approach, the architecture of the neural network that it is going to be evaluated is shown in Table 6.1. The architecture is based on the *SmallConvNet* topology described in [13] for music transcription. The architecture defined in this paper uses a Constant-Q Transformation but instead of using 384 bins, it uses 229. Thus, the input and output shapes of the architectures

Layer (type)	**Output Shape**	**Param #**
Conv2D	(None, 5, 384, 8)	80
BatchNorm	(None, 5, 384, 8)	32
ReLu	(None, 5, 384, 8)	0
Conv2D	(None, 3, 382, 8)	584
BatchNorm	(None, 3, 382, 8)	32
ReLu	(None, 3, 382, 8)	0
MaxPool	None, 3, 191, 8)	0
Dropout (0.25)	(None, 3, 191, 8)	0
Conv2D	(None, 1, 189, 8)	584
BatchNorm	(None, 1, 189, 8)	32
ReLu	(None, 1, 189, 8)	0
MaxPool	(None, 1, 94, 8)	0
Dropout (0.25)	(None, 1, 94, 8)	0
Conv2D	(None, 1, 92, 8)	200
BatchNorm	(None, 1, 92, 8)	32
ReLu	(None, 1, 92, 8)	0
MaxPool	(None, 1, 46, 8)	0
Dropout (0.25)	(None, 1, 46, 8)	0
Flattern	(None, 368)	0
Dense	(None, 16)	5904
BatchNorm	(None, 16)	64
ReLu	(None, 16)	0
Dropout (0.5)	(None, 16)	0
Dense (Sigmoid)	(None, 88)	1496
		$\sum$ 9040

Table 6.1: Architecture of the second CNN.

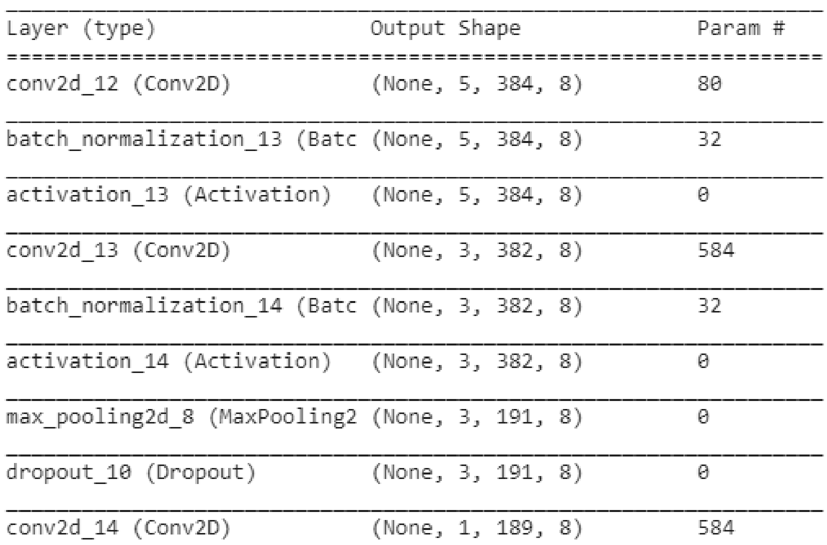
are different. In addition, the SmallConvNet is intended to predict 23 musical notes and in our case, the neural network has to be able to predict up to 88 different notes.

This second model uses a typical CNN architecture with three different types of layers: Convolutional, Dense and Pooling layers.

The convolutional layers shown in Table 6.1 have 8 filters and kernels of 3x3 while the MaxPooling layers have a pooling size of 1x2. This size cannot have squared magnitudes (e.g. 2x2, 3x3) because the input dimension has more elements in the columns than in the rows. If we tried to put a 3x3 as the pool size, the following error would be raised:

```
ValueError: Negative dimension size caused by subtracting 3 from 1 for 'conv2d_11/convolution'
```

This can also happen while adjusting the parameters of the convolutional layers or simply by adding new layers. The reason why this error is thrown it is because the input shape is to small for all the pooling applied. After each pooling layer, the size of the next layer is reduced, so it can happen that the shape is too small to fit in the next layers of the model. Consequently, it is very important to pay attention to the input and output of each layer and see how does it



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_12 (Conv2D)           (None, 5, 384, 8)         80
_________________________________________________________________
batch_normalization_13 (Batc (None, 5, 384, 8)         32
_________________________________________________________________
activation_13 (Activation)   (None, 5, 384, 8)         0
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 3, 382, 8)         584
_________________________________________________________________
batch_normalization_14 (Batc (None, 3, 382, 8)         32
_________________________________________________________________
activation_14 (Activation)   (None, 3, 382, 8)         0
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 3, 191, 8)         0
_________________________________________________________________
dropout_10 (Dropout)         (None, 3, 191, 8)         0
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 1, 189, 8)         584
```

Figure 6.1: First part of the summary of the architecture described in the Table 6.1

behave with the following layers. To see that data, `model.summary()` shows each type of layer, the output shape and the number of parameters like in Figure 6.1.

The final objective is to maximise the validation accuracy and also to prove that the neural network can learn from the data to create accurate predictions. The tests made in the Graph 6.2 shows that the CNN actually learns from the data because the more data is trained, the greater accuracy it gets. Henceforth, the next steps of this work are to progressively modify the architecture shown in the Table 6.1 with the objective of maximising the prediction accuracy using the same quantity of data.

Figure 6.2 shows an experiment using the same model but with a different number of input songs. This model has been trained in 20 epochs and at each epoch, the training and validation accuracy increase from the previous epoch (the initial learning rate was set to 10^{-4} in order to not having large accuracy variations between epochs). The results are as expected. It is possible to see that the higher number of samples trained, the better accuracy is obtained.

6.3 Modifications in the architecture

Some changes are applied in the architecture described in Table 6.1 with the purpose of increasing the validation accuracy. All the tests are going to be performed with the same data, the only thing that differs is the architecture of the model. The adjustments made and their resulting accuracy will be detailed next.

With the model shown in Table 6.1 after loading 50 songs in the training set and 15 in the validation set which leads a relation of 70-30%, the accuracy after 25 epochs reaches a **8,37%** (See Figure 6.3) which is not a very good result. All the following tests made with different architectures will have the same input data in order to test entirely the behaviour of the new topology. Moreover, the learning rate has been set to 10^{-4} in order to avoid the large accuracy jumps. The learning rate is a hyper-parameter that controls the adjustment of the weights. The smaller its value, the slower the network is going to learn because it is going to travel slower along the downward slope to converge to the local minima. In other words, the learning rate

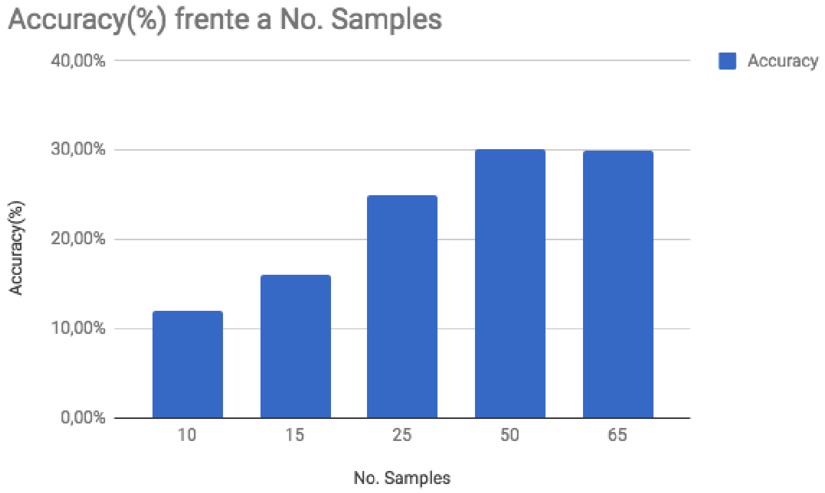


Figure 6.2: Graph showing the improvement of accuracy.

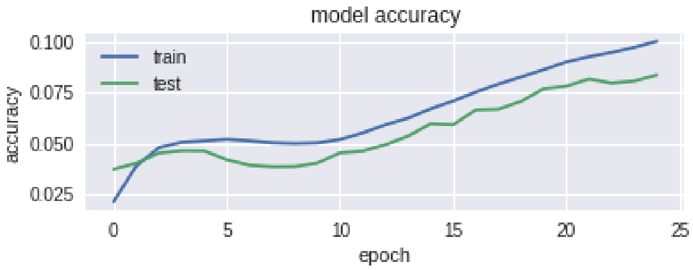


Figure 6.3: Evolution graph of the accuracy and validation accuracy in the initial proposed topology 2.

hyper-parameter determines how fast the neural network will reach the best accuracy. If the learning rate is too big, huge jumps will be made and it may fail to converge or even diverge. Otherwise, if it is too small, the neural network will need too much time to reach the local minima [31].

Furthermore it is important to highlight that MusicNet sorts songs by similarity between them. We can say that songs with *id_17* are similar to each other and different from songs starting with *id_22* for example. This detail is highly important to know which songs to train as it would not make sense, for example, to train with piano songs and validate with songs that are only played with a violin. The expected results will be poor.

Version 1

In this version of the architecture there are five convolutional layers. These layers will have a different number of filters, 16, 16, 16, 32 respectively in the convolutional layers and 64 in the last dense layer. Furthermore, the first Conv2D layer will have a 5x5 kernel instead of a 3x3.

After 25 epochs and several tests made, the accuracy increased to **24,42%**, as shown in 6.4.

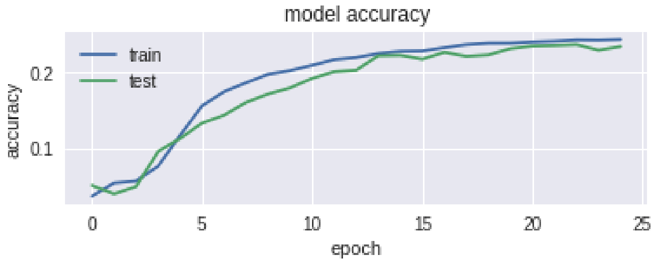


Figure 6.4: Evolution graph of the accuracy and validation accuracy in the version 1.

The training accuracy grows continuously every epoch as well as the validation accuracy that keeps improving. The difference in accuracy between the training and testing is minimum, being the training accuracy a little bit higher.

Version 2

In this second experiment, different activation functions are going to be compared. The previous research made in chapter 4.3 shows that there exist three main activation functions, sigmoid, tanh and ReLu. The performance should improve in the same order, being sigmoid the less recommended and ReLu the most suitable one.

The results applying the same data but changing the activation functions in the proposed topology *version 1* are as follows:

- Sigmoid: 15,23% accuracy. The first seven epochs the learning rate is really small and after that epoch starts to learn faster.
- Tanh: 24,02% accuracy. It learns but it has high peaks of accuracy, the growth is not steady.
- ReLu: 24,42% accuracy. It learns steady and achieves the highest accuracy.

To sum up, ReLu is the activation function from those evaluated that works better in the model. As ReLu was used in the proposed architecture, no changes will be made regarding to the activation functions.

Version 3

In this version, some new layers are going to be applied such as new dropouts, new dense layers as well as a Maxpool layer. Besides, the current Conv2D layers will have more filters and the model will have one less convolutional layer. The proposed architecture is shown in Table 6.2.

The results are the best compared to the rest topologies. It reaches **26,32%** after 25 epochs (see Figure 6.5).

Layer (type)	**Output Shape**	**Param #**
Conv2D	(None, 5, 384, 16)	160
MaxPool	(None, 5, 192, 16)	0
Dropout	(None, 5, 192, 16)	0
BatchNorm	(None, 5, 192, 16)	64
ReLu	(None, 5, 192, 16)	0
Conv2D	(None, 3, 190, 16)	2320
BatchNorm	(None, 3, 190, 16)	64
ReLu	(None, 3, 190, 16)	0
MaxPool	None, 3, 95, 16)	0
Dropout (0.25)	(None, 3, 95, 16)	0
Conv2D	(None, 3, 93, 32)	1568
BatchNorm	(None, 3, 93, 32)	128
ReLu	(None, 3, 93, 32)	0
MaxPool	(None, 3, 46, 32)	0
Dropout (0.25)	(None, 3, 46, 32)	0
Flattern	(None, 4416)	0
Dense	(None, 64)	282688
Dropout(0.25)	(None, 128)	0
BatchNorm	(None, 64)	256
ReLu	(None, 16)	0
Dense	(None, 128)	8320
Dropout(0.25)	(None, 128)	0
BatchNorm	(None, 128)	521
ReLu	(None, 128)	0
Dropout (0.5)	(None, 128)	0
Dense (Sigmoid)	(None, 88)	11352
		$\sum$ 307,432

Table 6.2: Model architecture of the version 3

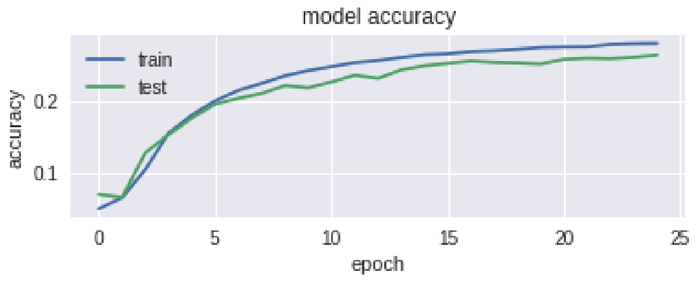


Figure 6.5: Evolution graph of the accuracy and validation accuracy in the version 3.

The accuracy improvement grows steady without having large jumps of accuracy.

Another aspect to highlight it that the neural network has a substantially higher number of parameters, from 9,040 to 307,432. This affects the computational cost.

Although the efficiency are a not as positive than the ones obtained in version 1, results are good and the growth is constant. That is why the modification 1 and 3 will be evaluated using more data to see how the models react to the a larger dataset.

Training with additional data

Once it has been made more changes in the hyper-parameters and adding and removing some other layers, the best architecture achieved is the 3. However, the versions have been tested with only the first minute of 50 songs in the training data and 20 in the testing set. After proving that the version 3 achieves the highest accuracy, some tests are going to be made with more data to obtain better results and hence, a better trained model.

This time, there is a memory limit of 13Gb in the graphic card K80 so it will be loaded as much songs as it is possible at once, but then, the model is going to be trained many times loading different data. Keras maintains the weights values of the inner layers so it is possible to fit the model each time with new data, the concept is similar to the proposed *generator* but it provides a faster loading in this case.

It is also important to take into account that the data for training and validation must be of similar nature. That is why MusicNet sorts the song by similarity so to train the model with all types of music, new sets of songs are loaded at each time. The training and validation accuracy may differ from the different sets of data loaded because the network may learn new frequencies with the different data. The important accuracy is the one that once all the network is trained will all the data, when the model is tested with a set of songs that MusicNet recommends. That set of songs (eleven pieces in total) contains a wide variety of styles and musical notes and it is going to be tested only once the model has been saved. Those songs are unknown to the neural network, in other words, the model has never faced such data before.

It is necessary to find the right balance between the length of the songs because Google Colab has a memory limit, hence, sometimes it is better to train the model with data from different songs instead of the same quantity of data from only one record. What the neural network will do if each iteration web train it with only one type of music, is that it will tend to learn from the last piece and will forget some features learnt before.

In this final test, the model has been trained and validated with 318 songs. The time of execution is about 5 hours and a half when it is trained with 200 seconds of each song. The model acquires a **37.25%** of global accuracy but, there are songs which have features that the neural network is likely to predict them better, thus reaching up to 55% accuracy in the predictions of musical notes for some songs.

The total number of windows used for the training set are 1.808.800, 651.168 for the validation and finally 113.695 windows to perform the final test.

All things considered, after developing dozens different architectures and modifying existing ones, the topology that responds better with as much data as possible is the architecture proposed in the version 3. The learning is robust and the results are satisfactory. It is not possible to show a graph of the learning evolution of the model because as it has been trained in an iterative mode. The important values are the resulting from testing songs after the model has been saved.

Chapter 7

Conclusions

In this last section, an overview of the work done will be presented, as well as an analysis of the results obtained and the future implementations that could be carried out on the basis of this project.

7.1 Overview

The recognition of musical notes played in a song is a complex problem to deal with because several notes can be played at the same time. The neural network has to be capable to distinguish the different notes from different instruments. To solve this problem, a Constant-Q Transform has been applied in order to obtain more valuable information from the song. Besides, the MusicNet dataset provides raw labels of each song but those labels do not fit in the expected output of the neural network. Herewith, all the musical notes have been analysed and it has been calculated where they are expected to be in the windows, creating a one-hot encoding per window which determines how many and what notes are played in each moment. After the data is preprocessed, the performance of different neural network architectures was evaluated, as well as studied the fundamentals of convolutional networks to provide understand how it works.

7.2 Results

The results are better than it was expected to be at the beginning. It has been accomplished a 37.25% of accuracy when detecting different overlapping notes played by different instruments. Nevertheless, there is a group of songs that the proposed topology was able to predict with a 55% of accuracy. Due to the number of possibilities and the multiplicity to solve the problem, the results obtained are really positive, taking into account that the state-of-the-art methods (from Google Magenta) are around 70%.

7.3 Future work

Musical notes recognition is just an opportunity that MusicNet gives. But thanks to the fact that more than 300 songs are labelled with different information such as *instrument id*, this allows us to create a neural network that not only predicts notes but also which instrument has played it. It is a more complex problem because there is a wider range of possibilities. From the research already done in this project, it is possible to modify the code to focus not only on the notes but also their timbre, which determines which instrument is being played.

By the prediction of the musical notes and the instruments that have played it, it would be possible the automatic generation of scores for each instrument playing in the song.

In addition, this work has made a study of the behaviour of audio in computers, the different types of existing representations and the possibilities they offer. The music's world on computers and its application to artificial intelligence can be further explored to create new software tools that are both innovative and useful.

This project is just the beginning of new challenges to come.

List of Figures

List of Tables

Bibliography

[1] J. Brownlee. Overfitting and underfitting with machine learning algorithms. *Machine Learning Mastery*, March 21 2016.

[2] A. K. Christian Schorkhuber. Constant-q transform toolbox for music processing. *7th Sound and Music Computing Conference, Barcelona, Spain.*, 2010.

[3] J. Despois. Finding the genre of a song with deep learning. `https://hackernoon.com/finding-the-genre-of-a-song-with-deep-learning-da8f59a61194`.

[4] M. A. et al. Tensorflow: A system for large-scale machine learning. *USENIX*, Noviembre 2016.

[5] N. S. et al. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research15 - University of Toronto*, 2014.

[6] R. K. et al. On the potential of simple framewise approaches to piano transcription. 15 Dec 2016.

[7] A. A. Freitas. Understanding the crucial differences between classification and discovery of association rules: A position paper. *SIGKDD Explor. Newsl.*, 2(1):65–69, June 2000.

[8] R. Hagiwara. How to read a spectrogram. `https://home.cc.umanitoba.ca/~robh/howto.html`.

[9] D. J. L. Igor V. Tetko and A. I. Luik. Neural network studies. 1. comparison of everfitting and overtraining. *University of Portsmouth*, January 27, 1995.

[10] A. Ingargiola. What is the jupyter notebook? `http://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/what_is_jupyter.html`.

[11] S. K. John Thickstun, Zaid Harchaoui. Learning features of music from scratch. 2017.

[12] P. P. Kabal. Wave- audio file format specifications. `http://www-mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/WAVE.html`.

[13] R. Kelz and G. Widmer. An experimental analysis of the entanglement problem in neural-network-based music transcription systems. *arXiv preprint arXiv:1702.00025*, 2017.

[14] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.

[15] R. K. G. J. R. Long. *MLC++: a machine learning library in C++.* Stanford University, Agosto 2002.

[16] A. Maxwell. 2 great benefits of python generators. `https://www.oreilly.com/ideas/2-great-benefits-of-python-generators-and-how-they-changed-me-forever`.

[17] M. Mohri. *Foundations of machine learning.* The MIT Press, 2012.

[18] Na8. Qué es overfitting y underfitting y cómo solucionarlo, May 2018.

[19] P. N. S. Network. What is a spectrogram? `https://pnsn.org/spectrograms/what-is-a-spectrogram`.

[20] K. official website. Keras documentation. `https://keras.io/`.

[21] J. M. I. Quereda. Sonido y musica por computador. 5 Septiembre de 2014.

[22] S. Raschka. *Python Machine Learning.* University of Pennsylvania-Packt Publishing Ltd., 2016.

[23] Shervine. A detailed example of how to use data generators with keras. `https://stanford.edu/~shervine/blog/keras-how-to-generate-data-on-the-fly.html`.

[24] A. Singh. Generators. `https://wiki.python.org/moin/Generators#CA-bdc68d1c164def45d62ea764ccc6bd755bf24e0b_1`.

[25] S. S. So and W. G. Richards. Application of neural networks: quantitative structure-activity relationships of the derivatives of 2,4-diamino-5-(substituted-benzyl)pyrimidines as dhfr inhibitors. *Journal of Medicinal Chemistry*, 35(17):3201–3207, 1992.

[26] TensorFlow. `https://www.tensorflow.org/`.

[27] Ujjwalkarn. An intuitive explanation of convolutional networks. `https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/`.

[28] S. University. Convolutional neural networks (cnns / convnets). `http://cs231n.github.io/convolutional-networks/`.

[29] D. Wolf. The 5 best programming languages for ai development. `https://www.infoworld.com/article/3186599/artificial-intelligence/the-5-best-programming-languages-for-ai-development.html`.

[30] R. W. World. Advantages of machine learning, disadvantages of machine learning. `http://www.rfwireless-world.com/Terminology/Advantages-and-Disadvantages-of-Machine-Learning.html`.

[31] H. Zulkifli. Understanding learning rates and how it improves performance in deep learning, Jan 2018.

Appendices

Appendix A

Github

All the code developed and used in the thesis has been uploaded to a public GitHub repository https://github.com/manuelminca/Automatic-Music-Transcription-Thesis.

The code has been developed in Google Colab so the code is optimised to their hardware. It may be necessary to install manually some libraries such as Librosa in order to run the code.