Capstone Project - Dr. Davide Guidi

1 - Introduction

In this project I intend to replicate part of the research efforts carried over by Darren Tio In [1] and Andy Wiggins in [2]. The goal of the project is to extract a dataset from GuitarSet and use it to train a model to recognise note patterns in an audio file. The model should learn to detect the notes being played in a song with a certain accuracy.

While the final goal of both [1] and [2] is to map the actual finger position in the fret board corresponding to the notes, for this project I am only interested in detecting the notes being played in the audio file.
For this purpose, GuitarSet will be used for both training and validating the model.

The project's goal is to develop a system that can identify notes being played in an audio file. The audio file consist of a mono audio recording containing a tune played by a guitar. To achieve this goal I will first create a dataset from the audio files in GuitarSet. Each audio file will be split into a chunk of appropriate length (20ms) and then a Constant-Q transformation will be applied on it.

These transformations, which can be represented as images, denote the frequencies information related to the audio chunk. From GuitarSet I will also extract the annotations and convert them into the notes detected in each audio chunk. The resulting dataset will contain images (the audio features) and the associated detected notes (the ground truth).

Such training dataset will be used as input and validation for a Convoluted Neural Networks. The CNN model will output a 6-dimensional output of 21 possible results - for each string the output will correspond to a classification of one label out of a maximum of 21. For each string, the 21 labels correspond to the closed position (the string is not played), the open position (the string is played open, no fingers on the fret) and the 19 different frets available on a standard guitar neck.

This paper is organised as follow. Section 2 contains some background work and empirical review of the state of the art libraries for audio analysis on Python. In addition, it contains a few experiments of real-time audio analysis inside Jupiter Notebook. Section 3 details the design and implementation process related to the project implementation. It contains information about data exploration and analysis, feature extraction, data refining process and the implementation of the CNN model. Finally, Section 4 details a further attempt I've done to improve accuracy and terminates with some conclusions and lessons learnt.

2 - Background on Audio Analysis

This section contains some basic research I've conducted to evaluate the available Python libraries for audio analysis. These empirical tests provided very important information and definitely helped shaping the design of the project.

Audio analysis using Python

A number of audio-related libraries are available for Python. The following libraries have been evaluated (for more information see Python in Music wiki pages - https://wiki.python.org/moin/PythonInMusic):
Aubio - Aubio is a tool designed for the extraction of annotations from audio signals. Its features include segmenting a sound file before each of its attacks, performing pitch detection, tapping the beat and producing midi streams from live audio.
MusicKit - The MusicKit is an object-oriented software system for building music, sound, signal processing, and MIDI applications. It has been used in such diverse commercial applications as music sequencers, computer games, and document processors. Professors and students in academia have used the MusicKit in a host of areas, including music performance, scientific experiments, computer-aided instruction, and physical modeling. PyObjC is required to use this library in Python.
pyAudio - PyAudio provides Python bindings for PortAudio, the cross-platform audio I/O library. Using PyAudio, you can easily use Python to play and record audio on a variety of platforms. Seems to be a successor of fastaudio, a once popular binding for PortAudio
pyAudioAnalysis is a Python library covering a wide range of audio analysis tasks.
librosa - A python module for audio and music analysis. It is easy to use, and implements many commonly used features for music analysis.

For the purpose of audio analysis, I found librosa [3] a quite mature project, frequently updated and providing a wide range of functionality.

Audio Experiment 1

This is a very basic experiment where I learned how to record live data from the sound card and how to set various parameters, such as audio format, sample rate and so on. In addition, I investigated a few libraries for visualising the result.

For this project I used the PyAudio library for audio sampling and BokehJS for the visualisation part.
The experiment source code can be found in the "AudioExperiement1.ipynb" file. The output of the experiment is presented above in Figure 1, which depicts the live audio stream plotted using BokehJS.

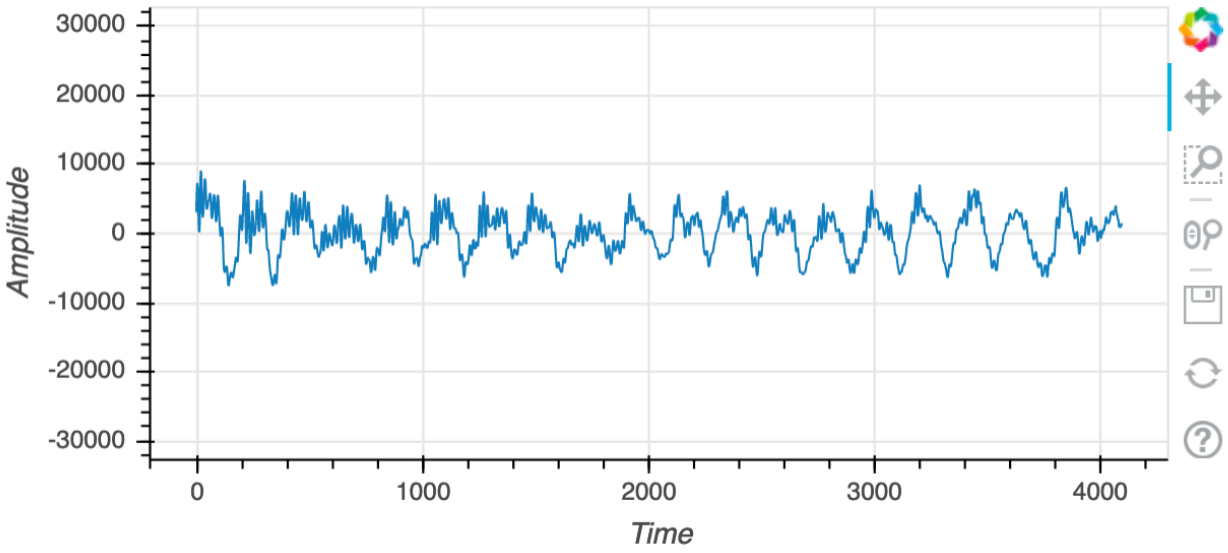

Figure 1 – Live audio output sampled from the sound

Audio Experiment 2

The next step was to explore audio analysis in Python. One important tool for audio analysis is the Fast Fourier Transform. Many implementation of the FFT exists for Python and for this experiment I used the FFT function provided by NumPy. Before experimenting with FFT, however, I had to figure out hot this can be integrated in Jupiter notebook in a useful way.
The main problem with live audio analysis is of course the limited computation available to process data on the fly. In addition, because I've decided to use Jupiter Notebook for my tests, I had to find a way to process data without blocking the user interface. In fact, just processing data in the main thread will lead to an unresponsive notebook cell that eventually need to be terminated manually.
This problem was solved by using a background process to do the calculation, while letting the main UI thread complete successfully. However, I still have an issue – how can the user stop the background process?
To solve this new problem I decided to introduce control widgets that can be used to start and stop the background process. At the beginning I tried using Bokeh widgets, but I eventually ended up using ipywidgets because they can easily communicate with the running python kernel.
The user experience when using ipywidgets and a background process was very good.
I had just a last problem to solve. My machine was unable to calculate the FFT fast enough in real time for all the sampled data. As a result, the program was unable to pull the audio data from PyAudio in a timely manner, and after a while an overflow exception was thrown.
I solved this issue by creating a callback function to pull the data from PyAudio (without processing), so that the data was always retrieved in due time.
The audio processing function was instead delegated to a different process, computing the analysis and the visualisation as fast as possible (but always less slower realtime) but without interfering with the audio collection side.
The source code for this second audio experiment is available in "Audio Experiment2 .ipynb" and the output is shown in Figure 2 below.

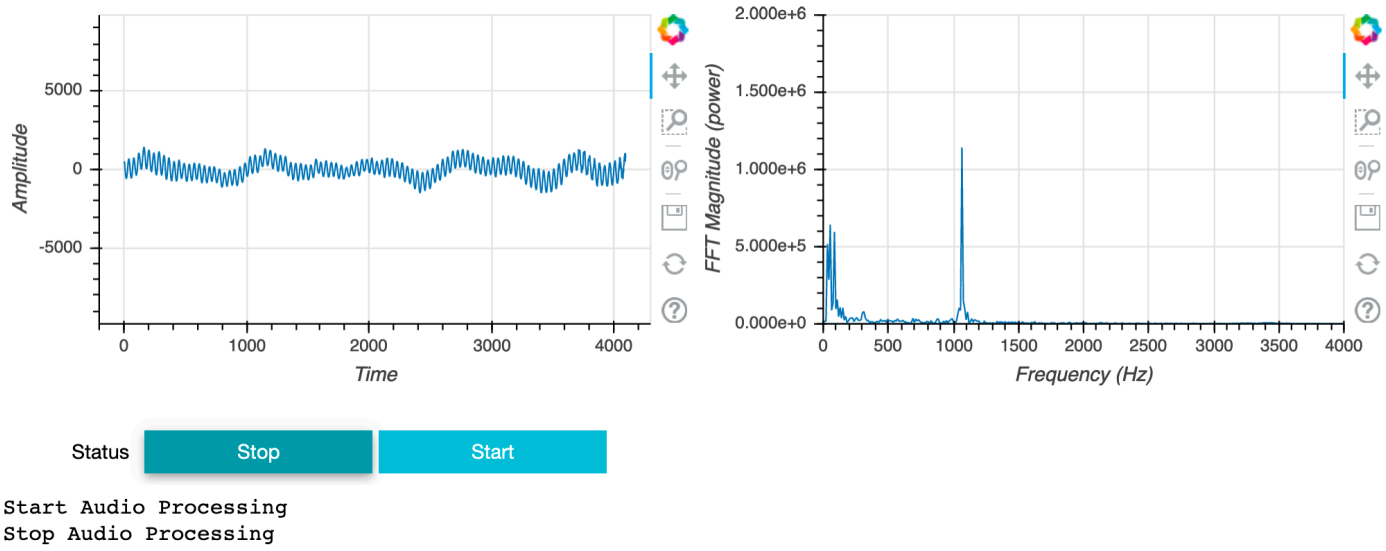

Figure 2 - Audio signal and its representation in the frequency domain

Audio Experiment 3

In this experiment I wanted to extend the last notebook replicating some ideas taken from the *polyphonic_track* GitHub project, by Jay Miller [5]. In this project, the author collects FFT fingerprints of each note, which are then used to track chords in real time. What interested me was the part related to the learning server. In order to simplify the learning process, the author implemented a workflow where the program communicates to the user the expected note to be played provides a feedback whether that note is detected or not. I wanted to replicate this workflow, and see if I could write a program that can detect a note, guiding the user in the learning process.
To achieve this goal, I've extended my last experiment in the following way:

- I've added some code, extracted from [5], that transform a given frequency into a note, for example 440 (Hz) to A4.
- I've added a cycle that continuously monitor the frequency domain of the live signal for a given frequency.
- Finally, I've added more widgets to allow entering the Learning Mode.

The user interaction works as follow: when the user click the "Learn Notes" button, the first note will be displayed (in our case E2, corresponding to the lowest possible note on the guitar neck), and the program will continuously scan the live audio. As soon as a signal above a certain threshold is detected, the FFT is calculated and the output is presented in the "Detected Note" graph. If the system detects the right note, then a positive message is displayed and after 1 second, the cycle will repeat for the next note, until the highest note is processed.
Otherwise, if the detected note differs from the expected one, an appropriate message is displayed, and after 1 second the cycle restarts for the same note.
This allows the program to semi-automatically collect FTT snapshots of single notes relative to the specific instruments, which can later be used to detect chords.

The resulting program worked quite well even when using a standard nondirectional microphone such as the default mic on a standard laptop, and I was able to record all the notes of the guitar.

The source code for this third experiment is available in the "AudioTest3.ipynb" file, and an example of it's output is given in Figure 3 below.

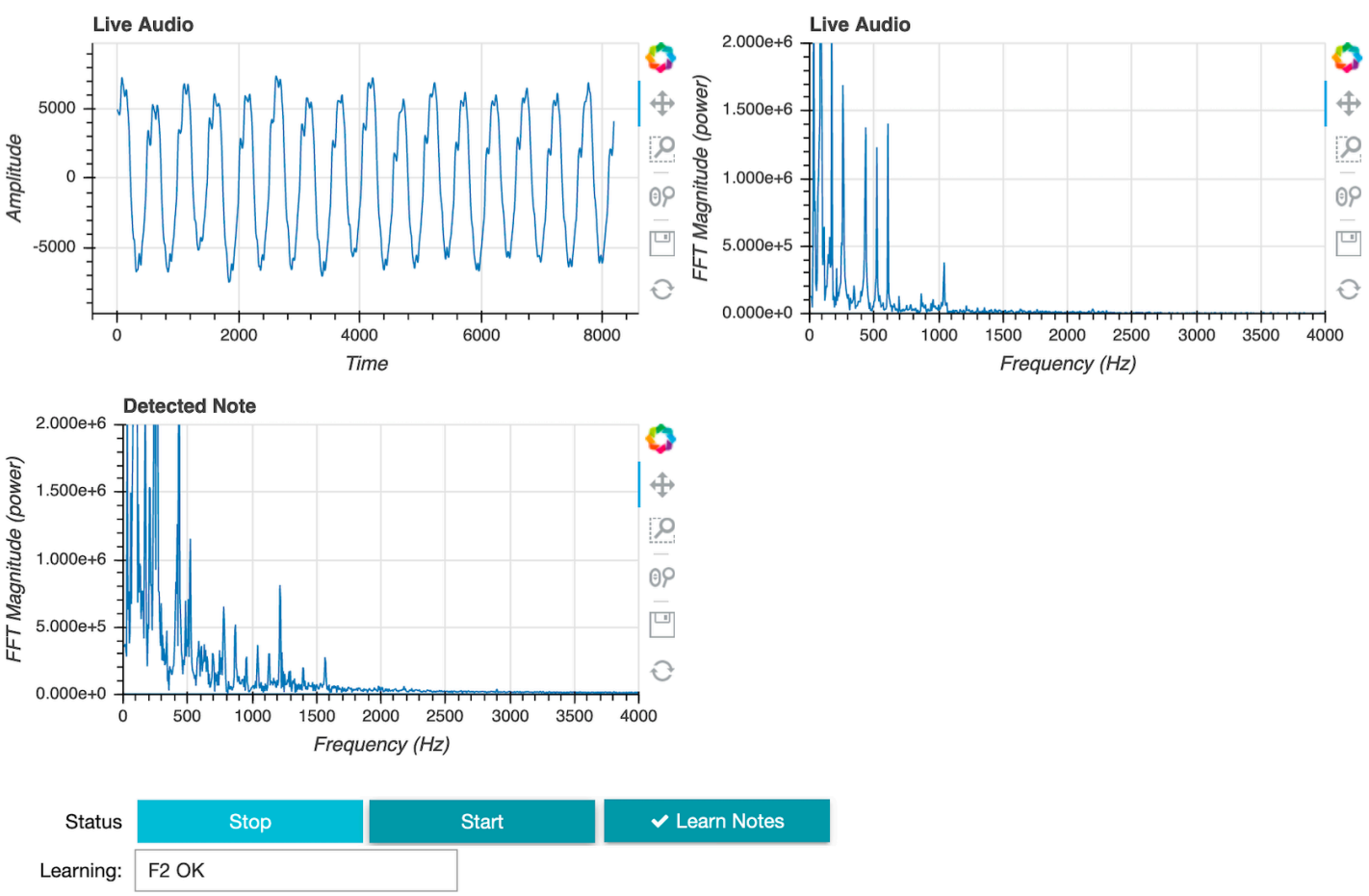

Figure 3 - Learning note F2

In the figure above the top 2 graphs represent the live audio signal and its representation in the frequency domain. The "Detected Note" graph act as kind of screenshot that is refresh whenever a signal above a certain threshold is detected. In this case the note to learn is F2, its screenshot is provided in the "Detected Note" graph, and as the note has been correctly detected the "Learning" text box is updated with a positive feedback ("OK")

This last experiment concludes my background research on audio analysis in Python/ Jupyter Notebook.

3 - Design and Implementation

In this part I will detail the design and implementation of the project.

Data Exploration and Analysis

The dataset used in this project is GuitarSet [3], a dataset that provides high quality guitar recordings alongside rich annotations and metadata.
The dataset is composed of 360 excerpts close to 30 seconds in length. Audio are recorded with the help of a hexaphonic pickup, which outputs signals for each string separately, allowing automated note-level annotation. For the recordings, players are provided with lead sheets and backing tracks reflecting the correct style that includes a drum kit and a bass line. Three audio recordings are provided with each excerpt with the following suffix:

- *hex*: original 6 channel wave file from hexaphonic pickup
- *hex_cln*: hex wave files with interference removal applied
- *mic*: monophonic recording from the reference microphone

Each of the 360 excerpts has an accompanying .jams file that stores 16 annotations.
The following annotations are provided:

Pitch:
- 6 pitch_contour annotations (1 per string)
- 6 midi_note annotations (1 per string)

Beat and Tempo:
- 1 beat_position annotation
- 1 tempo annotation

Chords:
- 2 chord annotations (instructed and performed)

The notebook associated to this part can be found in the file "Data Exploration and Analysis.ipynb". The waveform plot of an audio fragment of 20ms from each of the 6 pickup and the associated monophonic audio recording can be found in Figure 4. The first 6 graphs show the waveform for each guitar string, while the last graph contains the audio recording from the mono reference microphone.

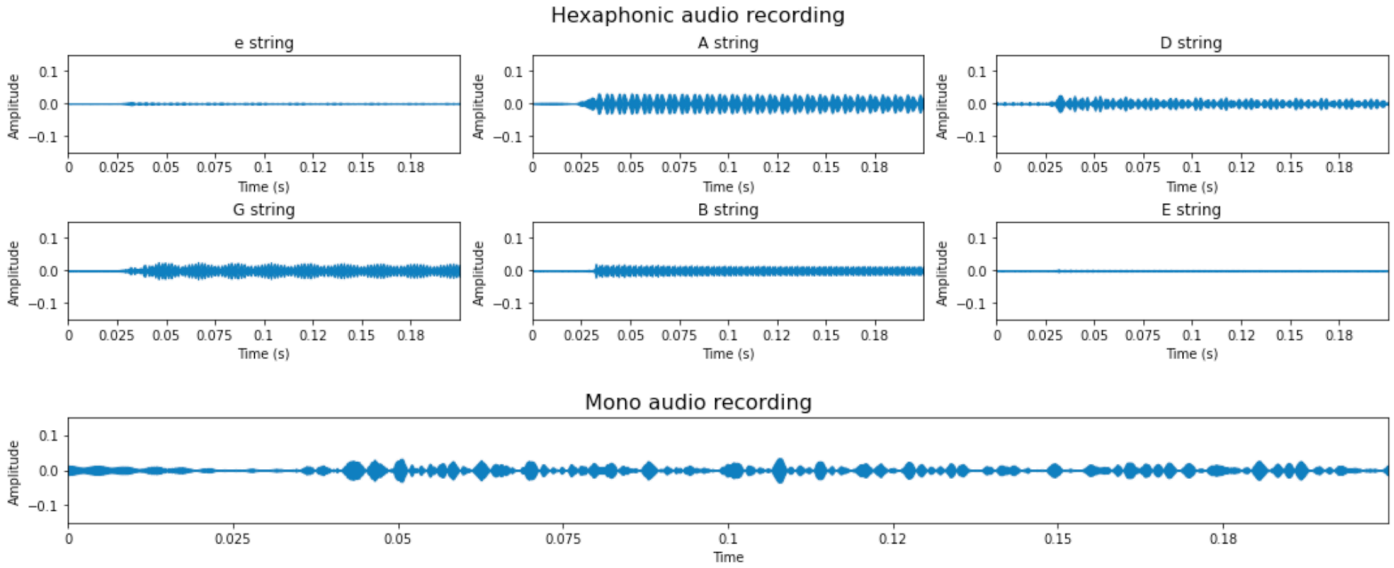

Figure 4 - Waveform plot of a test audio segment

Figure 5 shows the frequency domain related to the same chunk audio. The first 6 graphs shows the frequency analysis of each separate string, while the final one shows the frequency analysis on the mono recording.

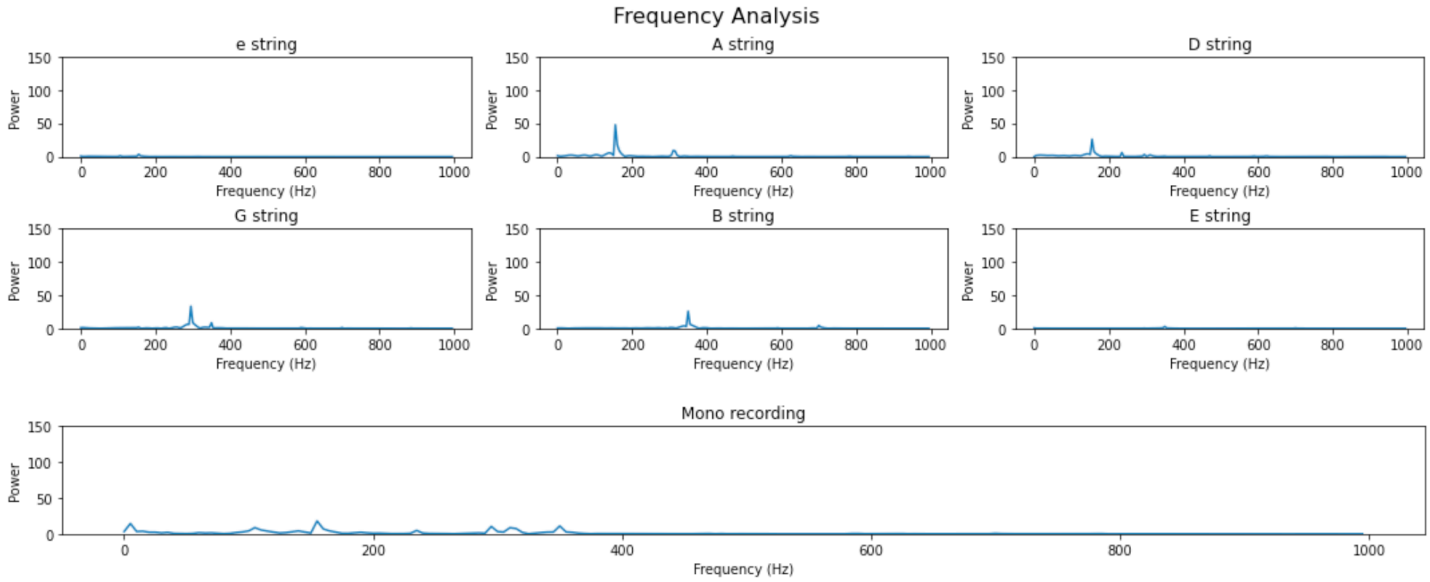

Figure 5 - Frequency analysis for the audio segment

To validate the dataset, I've calculated the frequency related to the strongest signal for each string in a test audio segment. Note that in this case I haven't set a threshold for the signal strength, so 6 frequencies are found in output, one for each string, irrelevant of the signal strength. The detected frequencies are then converted into notes by using librosa's `hz_to_note()` function. The output is presented in Table 1 below.

```
e string - detected frequency  155.0 D#3
A string - detected frequency  155.0 D#3
D string - detected frequency  155.0 D#3
G string - detected frequency  295.0 D4
B string - detected frequency  350.0 F4
E string - detected frequency  350.0 F4
```

Table 1 - Frequencies detection from the audio segment

I've then compared the detected frequencies with the annotations provided in GuitarSet. The frequencies extracted from the annotations are provided in Table 2 below.

```
A string  - annotated frequency 155.989 D#3
G string  - annotated frequency 295.282 D4
B string  - annotated frequency 349.936 F4
```

Table 2 - Annotated frequencies (ground truth)

The annotated frequencies are very close to the frequencies detected by applying FFT. In addition, the associated notes matched perfectly. We can further observe that the annotation only provides 3 notes instead of the 6 notes detected by the naive detection approach. This is because the signal strength for the e String and E String is very low (as can be seen in Figure 5) and for this reason they should be discarded. For the D String , however, the situation is different. The FFT graph in Figure 5 clearly shows a relatively strong signal, but why is that not present in the annotations? The reason is that the D string simply resonates with the G String, it's not being actively strummed.

This is something to be expected, as I've used the original hexaphonic recordings for this example. Much better results can be obtained using the hex_cln dataset, where the interference is filtered out.
The waveform graphs from the hex_cln dataset for the same audio segment can be found in Figure 6. It's easy to see that the waveforms differ from the previous one but no tmuch more information can be extracted.

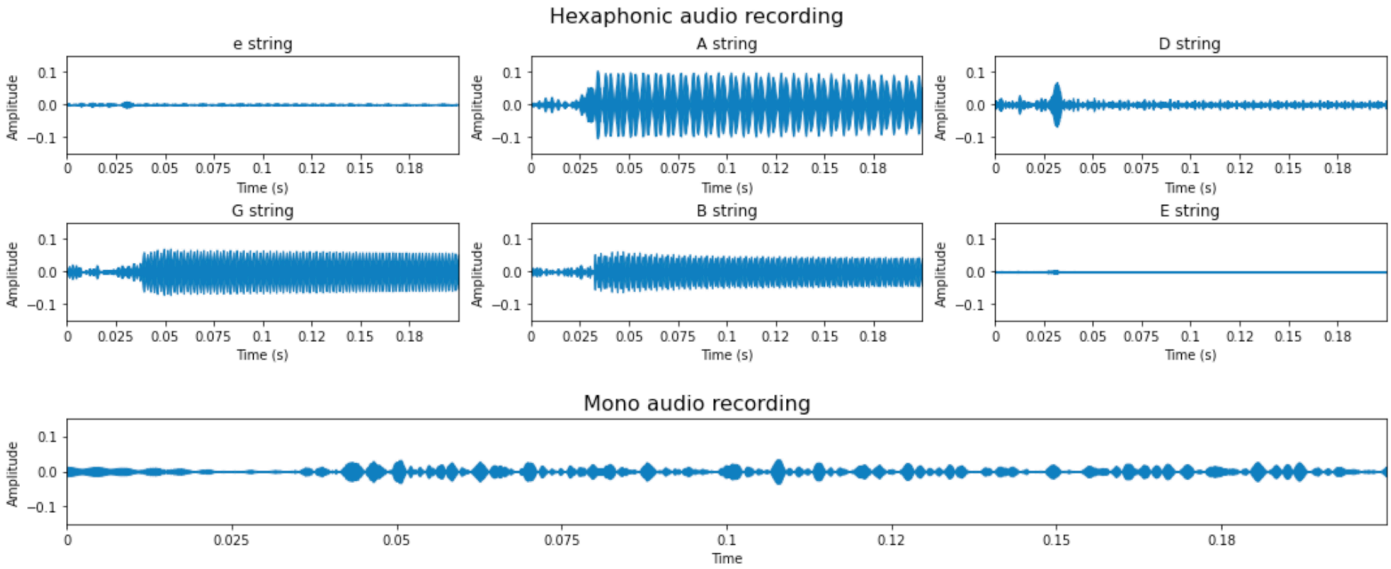

Figure 6 - Waveform plot of the cleaned audio segment

The related FFT signal is presented in Figure 7 below. Here the situation becomes interesting, we can see that there is a strong signal associated to A string, G string and B string, while the signals related to the other strings are quite faint.

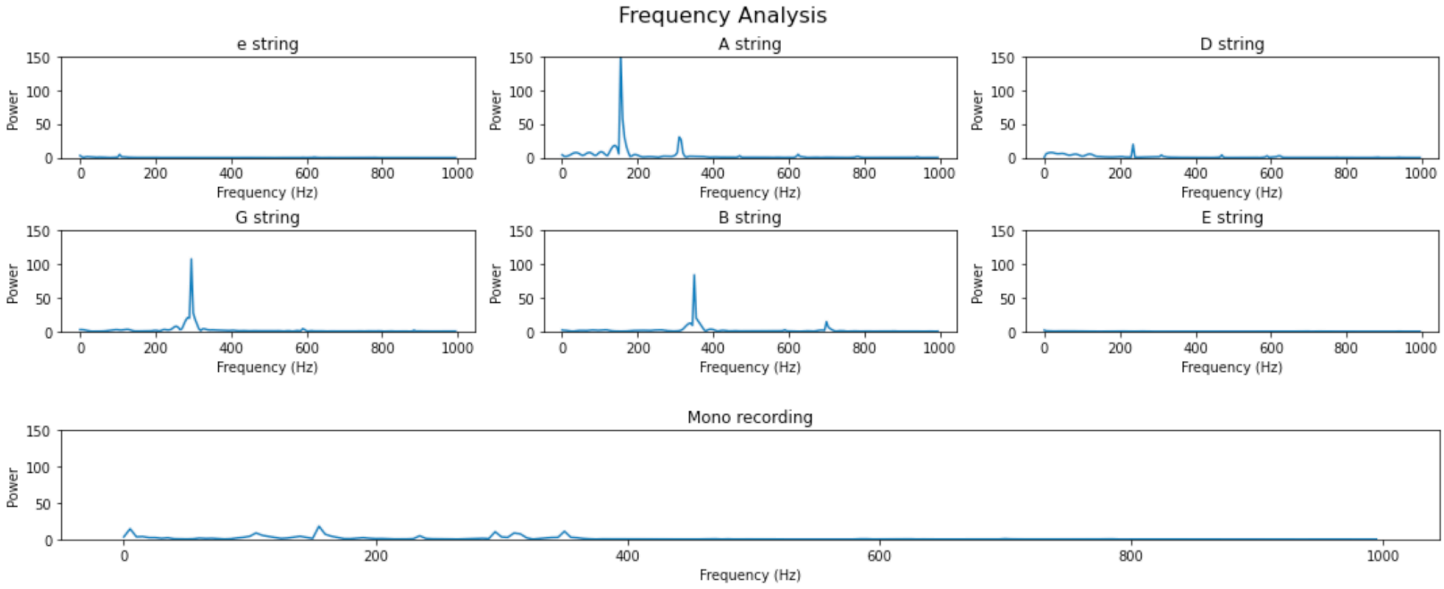

Figure 7 - Frequency analysis of the cleaned audio segment

Running the frequency detection algorithm on this new data, we still obtain the same results for A string, G string and B string. In addition, simply specifying an appropriate signal threshold is enough to filter out the faint signals. The detected frequencies are presented in Table 3.

```
A string - detected frequency  155.0 D#3
G string - detected frequency  295.0 D4
B string - detected frequency  350.0 F4
```

Table 3 - Frequencies detection from the cleaned audio segment

As expected, these notes perfectly match the ground truth annotations of Table 2.

Feature Extraction

In this project the feature extraction is divided in two distinct parts. The first part involves segmenting each audio file in small chunks, with a duration of 20ms, which will contain a maximum of one note per string. After that, a Complex-Q Transformation will be applied to the chunk audio data, resulting in an image containing the frequency information.
The second part is related to the extraction of the annotation information, the ground truth, for each audio chunk.

Complex-Q Transformation

The code implementation related to the first step was not particularly complex.
My first implementation involved splitting every audio file into chunks and creating an actual PNG image from the Complex-Q Transformation. The image was then saved in a file inside a folder named after the specified audio file.
The parameters for the `librosa.cqt()` function have been calculated in this way:

- The minimum frequency (`fmin`) has been set to C2, just two tones below the lowest possible note in a standard tuned guitar (E2)
- The number of `bins_per_octave` have been set to 12, as we want to classify a note into one of the 12 possible values per octave (every semitone step from C to B)
- `n_bins` has been set to 7*12 as we are interested in a maximum of 7 octaves.

A few special parameters have been included to allow Matplotlib to save a figure without borders and without axis information (`bbox_inches = 'tight', pad_inches=0, set_axis_off()`.

A Further code fragment would load back every PNG image file, converting it to RGB (i.e. ignoring the alpha channel) and save the final result as a NumPy array.
In a further optimisation I unified these 2 steps, so that the extracted image was directly aggregated int he training dataset.
Finally, the data was further processed by normalising the matrix containing the image data

Annotation extraction

Extracting the information from the annotation required some work. Annotations are stored in a .jams file, which uses a JSON file format.

The key problem to solve here is that while the audio file is composed of sequential audio chunks, annotations are stored in a non-sequential way.
The annotation file contains 6 sets of observations, one for every string. Every observation contains a number of information, including a time and an annotated frequency. The annotations do not contain information for every audio chunk, which means that the number of observation per string differ. While parsing the annotation, we have to infer the fact that, if an annotation is not present for a specific time interval, it would mean that the specified string was not strummed.

In addition, a few edge cases needed to be handled:
- There is at least a track which does not contain any observation for a specified string. This simply means that the specified string was never strummed when performing the song, which is uncommon but very possible
- Some annotated frequencies have a value of 0 Hz, which broke the computation of the associated note

Finally, the annotated frequency was converted to its note representation (using `librosa.hz_to_note()`), and the result added to a note vocabulary.
The data is then processed further and converted into a one-hot encoding matrix.

Processing and refinement

While validating the extracted annotation, I discovered that some annotated frequency were due to the string resonating with another string, instead of the string being actively strummed. The easiest way to detect this is to check the annotated frequency against the list of frequencies allowed for a particular string. In the guitar, each string can only be be sound a specific note range. To improve accuracy I've decided to ignore any annotation where a string was resonating with another one. To do this I had to explicitly define which note range is possible for every string, as in Table 4 below.

```
notesInString = [
[‘e string OFF', 'E2', 'F2', 'F#2', 'G2', 'G#2', 'A2', 'A#2', ‘B2', 'C3', 'C#3', 'D3',
'D#3', 'E3', 'F3', 'F#3', 'G3', 'G#3', 'A3', 'A#3', 'B3'],
['A string OFF', 'A2', 'A#2', 'B2', 'C3', 'C#3', 'D3', 'D#3', 'E3','F3', 'F#3', 'G3',
'G#3', 'A3', 'A#3', 'B3', 'C4', 'C#4', 'D4', 'D#4', 'E4'],
['D string OFF', 'D3', 'D#3', 'E3', 'F3', 'F#3', 'G3', 'G#3', 'A3', 'A#3', 'B3', 'C4',
'C#4', 'D4', 'D#4', 'E4', 'F4', 'F#4', 'G4', 'G#4', ‘A4'],
['G string OFF', 'G3', 'G#3', 'A3', 'A#3', 'B3', 'C4', 'C#4', 'D4', 'D#4', 'E4', 'F4',
'F#4', 'G4', 'G#4', 'A4', 'A#4', 'B4', 'C5', 'C#5', ‘D5'],
['B string OFF', 'B3', 'C4', 'C#4', 'D4', 'D#4', 'E4', 'F4', ‘F#4', 'G4', 'G#4', 'A4',
'A#4', 'B4', 'C5', 'C#5', 'D5', 'D#5', 'E5', 'F5', 'F#5'],
['E string OFF', 'E4', 'F4', 'F#4', 'G4', 'G#4', 'A4', 'A#4', 'B4', 'C5', 'C#5', 'D5',
'D#5', 'E5', 'F5', 'F#5', 'G5', 'G#5', 'A5', 'A#5', ‘B5']]
```

Table 4 - Specifying the available note range for each string

The `notesInString` list contains 21 possible notes for each string, corresponding to the string being muted (OFF), the string being played open and each note available in the 19 frets.

Processed data

The result of the data processing step is as follow:

- `x_train_opt` array, containing an array of over 50000 images taken from the Complex-Q Transformation
- `y_train_one_hot` array, consisting of 6 arrays of the same x_train_opt length, containing annotations for every audio chunk for each string

Feature validation

The extracted feature appear to be consistent. Figure 8, below, shows a random entry where the annotation data is consistent with the CQT image - 3 notes are annotated and 3 bands can be seen in the associated CQT image.

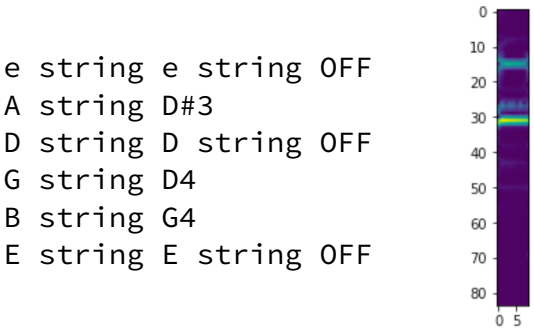

Figure 8 - The annotated data is consistent with the CQT image

Model

The approach use by both [1] and [2] is to apply a Convolutional Neural Network.
Convolutional Neural Networks (CNN) are a class of Deep Neural networks, inspired by biological processes.
A key feature of a CNN, and the reason of their popularity, is dimensionality reduction. CNN are very good at reducing the number of required parameters without affecting the quality of the model. They do that by using a sliding window technique, where the window size is smaller than the input.
In a generic CNN the input is usually connected to a number of hidden layers which are composed of a feature learning part and a classification part.
The feature learning is made up of one or more convolutional and pooling layers. The convolutional layer convolve the input to the next layer, while the pooling layer performs the dimensionality reduction.

The classification part includes a fully connected layer and a softmax classification technique that can learn a possibly non-linear function. CNN are very popular for detecting features in images and as such they should perform very well when presented with data from Complex-Q transformations.

The model I've created is based on the works done in [1] and [2]:

- The input is composed of 192 (frequency bins) x 9 (time frames) CQT images, representing 200ms of isolated acoustic guitar audio.
- I've added three convolutional layers, each with a filter size of 3 x 3. The first convolutional layer has 32 filters, and the latter two each have 64. Each convolution is immediately followed by a Rectified Linear Unit (ReLU) activation.
- The feature maps are then subsampled by a max pooling layer. Both the filter size and the stride for this operation are 2 x 2.
- The structure is then flattened and followed by a dense layer of dimension 128, which includes a ReLU activation. This is connected to a second dense layer of dimension 126 with no activation.
- Finally, the vector is reshaped to 6 x 21, and a 6-dimensional softmax activation is applied. The output shape represents the 6 guitar strings and the 21 different classes related to each string

The results obtained after 30 epochs are presented in Figure 5 below.

```
e string_accuracy: 0.8860
A string_accuracy: 0.8152
D string_accuracy: 0.7692
G string_accuracy: 0.7582
B string_accuracy: 0.7063
E string_accuracy: 0.7790

Average accuracy: 0.7856
```

Table 5 - Accuracy results

Result analysis

The results obtained appear to be decent, but somehow worse than the accuracy level of the works I used as reference. The results obtained in [1] and [2], used as benchmark, are much better, with an average accuracy of 0.842 and 0.845 respectively.

The source code related to the feature extraction part can be found in "Feature Extraction2.ipynb", while the model source code can be found in "CNN Model2.ipynb"

4 - Accuracy improvement and conclusions

As I was not satisfied with the accuracy level, I went back to study the process I've created to see if something could be improved.

I've tested with a number of different parameters: smaller or bigger audio segments, differ model parameters, etc, but I've always obtained worse or similar results.

In this project I assumed that a single note would last at least 20ms, but when analysing the annotations I've noticed that a number of observations were recorded in the same 20ms period.
This was interesting. In my first attempt I was segmenting the audio file in chunks of 20ms and I was creating a single CCQT image for every audio segment.
A more detailed analysis of those images revealed that while in some of them I could see a straight line across the whole width of the image, in others I could see a partial one, as presented in Figure 9 below.

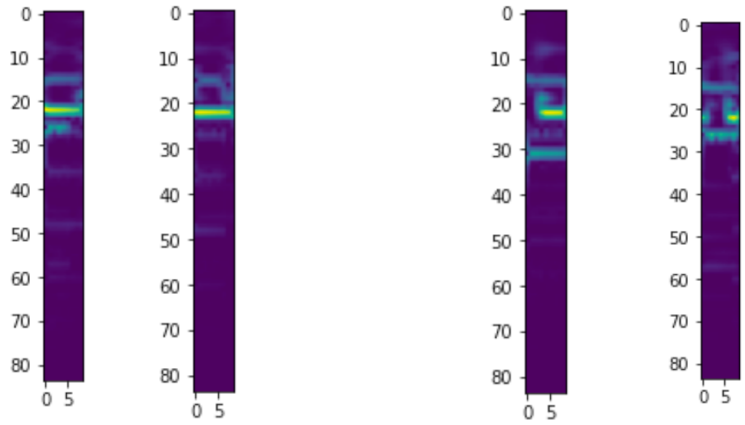

Figure 9 - Different CQT images

In the left side of Figure 9 we can see 2 examples were the detected frequencies are represented as straight lines across the whole image width, meaning that those frequencies were detected in the whole 20ms. On the right side we can see 2 examples were the lines are not visible across the whole width of the image, but only partially. This means that the frequencies were not detected for the whole 20ms segment. In other words, the CQT snapshot was taken during the attack or decay of the note; the frequency was not detected consistently in the 20ms audio chunk.

Following this consideration, I've decided to rewrite my feature extraction algorithm, so that, whenever an annotation is found, not just one CQT was created per single audio segment, but a number of them. Intuitively, this should help the model to perform better as it provides additional information for every note. I've started creating 2 CQT image per audio segment and noticed an improved accuracy, I then moved to 4 and finally I created 10 CQT per audio segment.
While in my first approach I created only 1 CQT image for an audio segment of 20ms, in this second approach I created 10 CQT images, taken every 2ms, for the same audio segment.
The results presented in Figure 6 definitely confirmed my hypothesis - the accuracy is much improved!

```
e string_accuracy: 0.9321
A string_accuracy: 0.9021
D string_accuracy: 0.8788
G string_accuracy: 0.8785
B string_accuracy: 0.8816
E string_accuracy: 0.9314

Average accuracy: 0.9007
```

Table 6 - Improved accuracy

The source code my second attempt can be found in "Feature Extraction5.ipynb" and "CNN Model5.ipynb".

Conclusions

The results I've obtained are in line with the results in [1] and [2]. The same observations valid for the [1] are still valid here: the current model does not take into account the duration a note is held and for this reason the model is not yet ready to begin creating full length guitar tablature.
However, the results confirm that a CNN model can be well suited for audio analysis.
Further work could involve translating the detected notes to the related finger positions, for example using a Python library such as `fretboard`.

References

[1] Darren Tio, Automated Guitar Transcription with Deep Learning. Available at https://towardsdatascience.com/audio-to-guitar-tab-with-deep-learning-d76e12717f81
[2] Andy Wiggins and Youngmoo E. Kim, Automatic Guitar Tablature Transcription with Convolutional Neural Networks. Available at http://nemisig2019.nemisig.org/images/kimSlides.pdf
[3] Brian McFee et al, librosa: Audio and Music Signal Analysis in Python. Available at: https://dawenl.github.io/publications/McFee15-librosa.pdf
[4] Q. Xi, R. Bittner, J. Pauwels, X. Ye, and J. P. Bello, "Guitarset: A Dataset for Guitar Transcription", in 19th International Society for Music Information Retrieval Conference, Paris, France, Sept. 2018.
[5] Jay Miller, polyphonic_track project. Available at https://github.com/jaym910/polyphonic_track