UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER'S THESIS

Classification of Music Based on Machine Learning

Luka Čupić

Zagreb, July 2020.

For the people who share my last name: my family and my mentor.

CONTENTS

List of Figures

LIST OF TABLES

1. Introduction

Audio classification is an interesting machine learning problem that often doesn't get as much attention as some other contemporary problems such as computer vision or social media analysis.

This thesis will explore the problem of sound analysis with the purpose of performing real-time classification of musical instruments. An appropriate feature extraction mechanism will be employed to extract audio features that will be used to train a deep neural network. Once built, the neural network model will be deployed to a custom-built Android application. This will allow any Android device to perform predictions of recorded musical instruments in real time.

By deploying the model on to a mobile application, this thesis steps outside of the academic context and utilizes theoretical machine learning knowledge for building a standalone application capable of working in interesting real-world scenarios. Such a combination aims to bridge the gap between "theoretical" computer science and "practical" software development and show how the two are able to fully complement each other, even in the academic setting.

The second chapter of this thesis goes through some basic background theory behind digital representation of audio. Third and fourth chapters describe the dataset and some useful techniques for preparing the data. Fifth (and most important) chapter goes through the process of feature extraction in great detail. Chapters six and seven describe how the extracted features can be used to build the deep learning model and how do deploy it on to a standalone Android application. Finally, chapters eight and nine discuss some results and provide the final conclusion.

At the end of the thesis, there is a Glossary page containing important concepts which can be used as a reference point.

2. Digital Sound Representation

2.1. Brief History of Sound Recording

The earliest known device for sound recording is a phonautograph. It was invented by Édouard-Léon Scott de Martinville and patented in 1857 [24]. The device was directly inspired by the human ear and its elements are thus analogous to those of the ear. First, the outer layer of the ear (called pinna) picks up vibrations of a sound signal and funnels them through the ear canal. At the end of the ear canal is a membrane called ear drum which starts vibrating upon receiving a sound wave. Ear drum vibrations are sent through a chain of small bones which are then transferred to the cochlea — a snail-like spiral part of the ear. The cochlea finally converts these vibrations into nerve impulses which get sent to the brain for interpretation [20]. Cochlea is such a remarkable natural mechanism that we will refer to it once more in Chapter 5.

Similarly to the ear, phonautograph picks up sound signals from the environment and funnels them through a long cone-shaped tube. At the end of this tube is a membrane with a stylus attached to it. As sound wave travels through the device, the stylus moves according to the vibrations and paints a sound pattern on a moving piece of paper [16]. The drawn pattern represents a sound wave. The idea of the phonautograph was later improved upon by Thomas Edison who invented the phonograph in 1877. This device (later also known by a more recognizable name — gramophone) could be used both for recording and reproduction of sound. Decades later, first electric microphones started to appear. The steady progress of technology combined with the aftermath of World War II lead to a 'technological boom' in the second half of the 20th century. This meant digitalization of numerous technologies, including microphones and other sound devices.

Digital sound recording works in much the same way as analog sound recording, or the human ear itself, for that matter. An incoming sound wave causes the membrane of a microphone (called the diaphragm) to vibrate. This vibration is transferred to a small coil surrounding the magnet behind the diaphragm, as seen in Figure 2.1. Diaphragm's

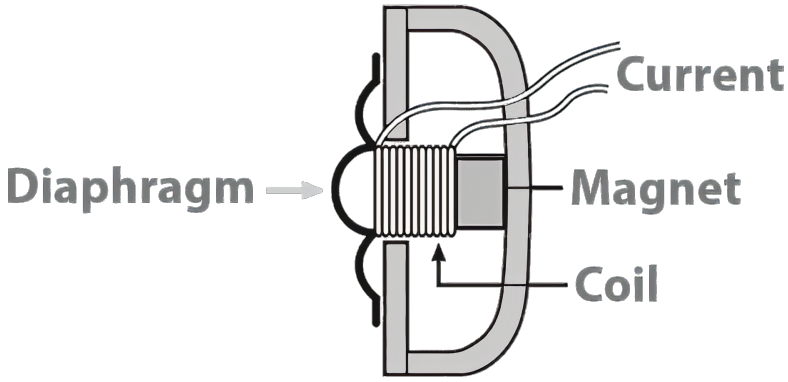


Figure 2.1: Inner workings of a dynamic microphone [6]

vibrations cause the coil to vibrate back and forth. As a consequence, electric current is induced in the coil as a result of the electromagnetic induction. This causes a voltage difference between the two ends of the coil and allows the current to flow through the connected electric circuit. This is how a microphone works.[1]

2.2. Sound Sampling

After a recording device (microphone) has converted the incoming sound wave first into vibration and then into electrical signal, one final conversion still needs to take place — analog-to-digital conversion.

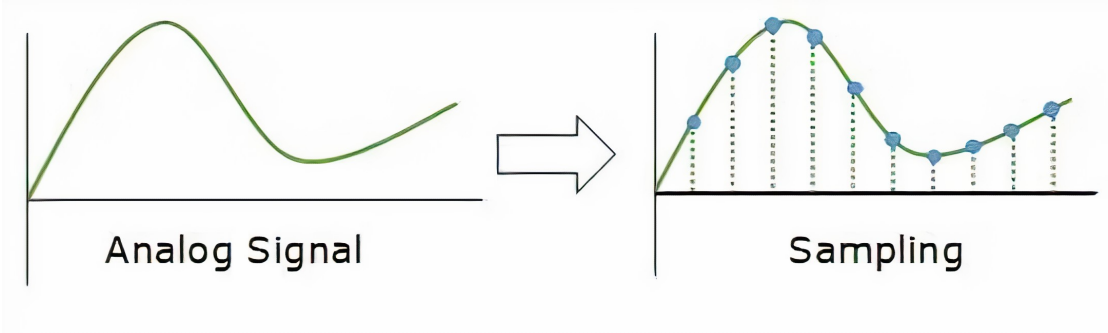


Figure 2.2: Signal sampling [8]

[1] In reality, there are many other types of microphones. The one described here is called a dynamic microphone that uses electromagnetic induction to generate electric current.

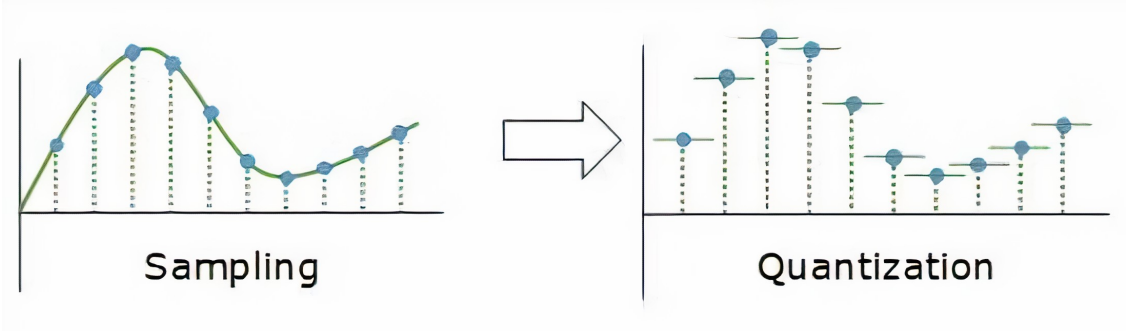


Figure 2.3: Signal quantization [8]

The basic idea behind analog-to-digital converter or ADC is that it transforms a signal of continuous time and continuous amplitude into a signal of discrete time and discrete amplitude. In other words, it converts an analog signal into a digital signal. Figure 2.2 illustrates this a bit more clearly. As the analog input signal is received by the system, it gets "vertically" sampled into discrete time points caled **samples**. How many of these samples the system is going to take depends on the frequency of sampling. The sampling frequency (or sampling rate) tells the system how many of these points it should take in any given second. For instance, a sampling rate of 16 000 Hz means that 16 000 discrete points are going to be sampled every second. This step of the conversion process is appropriately called **sampling** and can be seen in Figure 2.2.

So far, the system has converted the recorded analog signal into discrete time points, each of them representing a certain voltage picked up by the microphone. However, the system had sampled the signal only along the x-axis. The y-axis, containing the signal amplitude (i.e. voltage) has remained unchanged; the amplitude values are still represented as real numbers. Since computers are unable to work with real numbers of infinite precision, an approximation needs to be made for each of the sampled points. Similarly to how we defined sampling rate to be the precision on the x-axis (the density of vertical lines), we are going to define the precision on the y-axis (the density of horizonal lines).

The entire y-dimension will be split into a fixed number of possible amplitude values. As each point is processed by the system, the amplitude of each of the points is going to be rounded to the nearest fixed value on the y-axis. The number of these fixed values is called bit depth and is often expressed in the powers of two. The reason for this is the fact that each of the points is stored in a pre-determined number of bits. If we have only two bits then four distinct possibilities are possible: 00, 01, 10, and 11. With each increase in the number of available bits, the total number of possibilities

gets multiplied by two. If working with 16 bits, there will be 2^{16} or 65 536 possible values. This process of "horizontal sampling" is called **quantization** and can be seen in Figure 2.3.

Another way of looking at the sampling-quantization process can be imagined as placing a two-dimensional grid over the analog signal [4]. The vertical lines would correspond to sampling, whereas the horizontal lines correspond to quantization. This is visible in Figure 2.4.

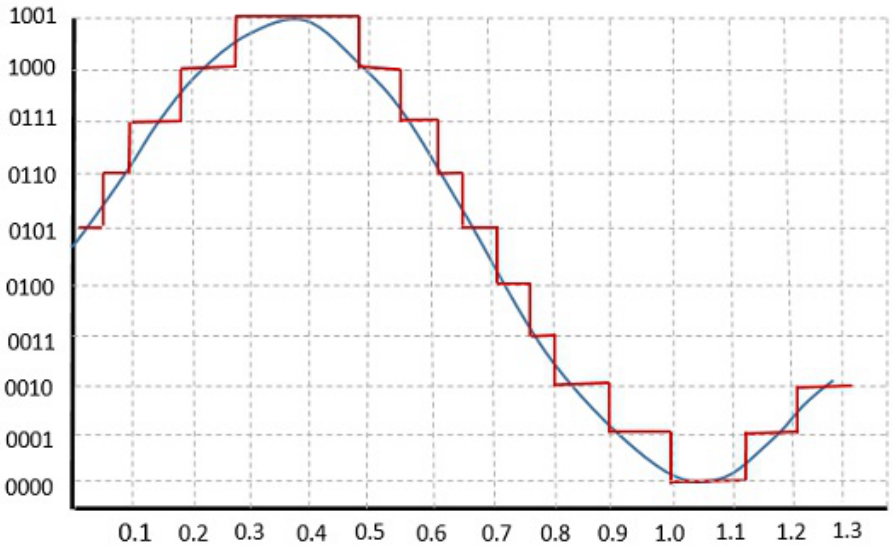


Figure 2.4: "Horizontal" and "vertical" sampling [29]

At this point, the signal has been fully digitalized and converted into a form which can be analyzed using a digital computer. A slightly simplified version of the digital audio representation (which ignores other information such as meta-data) can be though of as having a vector of numbers (e.g. `float[]` or `short[]`) containing the samples obtained through the microphone. Depending on the recording length and on the sampling rate, the vector is going to have a different number of elements. For a sampling rate of 16 000 Hz and audio length of 2 seconds, the total number of elements is going to be 32 000 (16 000 elements each second, multiplied by 2 seconds).

2.3. Fourier Transform

Before any kind of meaningful audio analysis can be performed, some pre-processing needs to be done first. The following example should clarify why this is the case.

Figure 2.5 shows the waveform of an acoustic guitar recording. It is quite similar to the waveform of a ukulele track shown in Figure 2.6.[2] By comparing the two waveforms, it is clear that it would be virtually impossible to differentiate instruments based on their time-domain waveforms alone. In other words, there is only so much information that can be inferred from raw audio.

For this reason, certain pre-processing techniques and transformations are used prior to the audio analysis itself. One such technique is called the Fourier transform, whose purpose is to decompose a signal into its basic frequencies. This is possible because every (audio) signal is essentially a composition of simpler, "pure" signals. In his original work, Joseph Fourier used a transformation of an arbitrary function (in our case, audio signal) into a series of independent cosine and sine waves [17]. Since every (co)sine wave has a constant frequency, any complex signal can be constructed by adding these simpler signals together and vice versa: any complex signal can be decomposed into those simpler signals. This is what is meant by decomposition: a signal is deconstructed into its base frequencies and each base frequency is represented by a (co)sine wave of that same frequency.

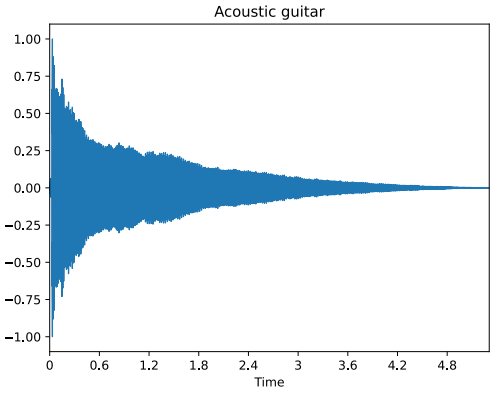


Figure 2.5: Time-domain representation of a guitar signal

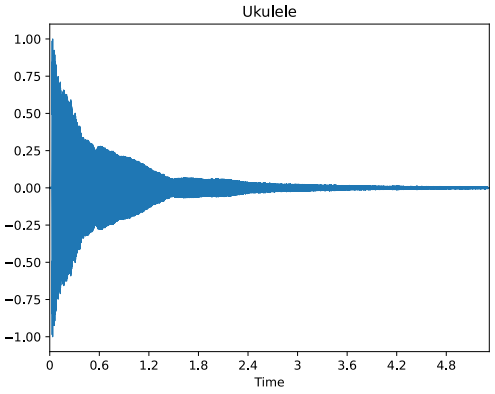


Figure 2.6: Time-domain representation of a ukulele signal

[2]Both audio tracks were normalized to make the comparison consistent.

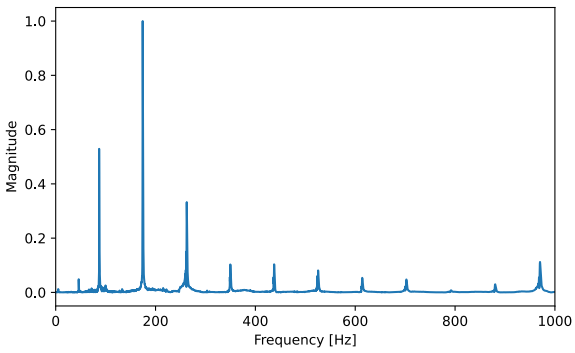


Figure 2.7: Frequency-domain representation of a guitar signal

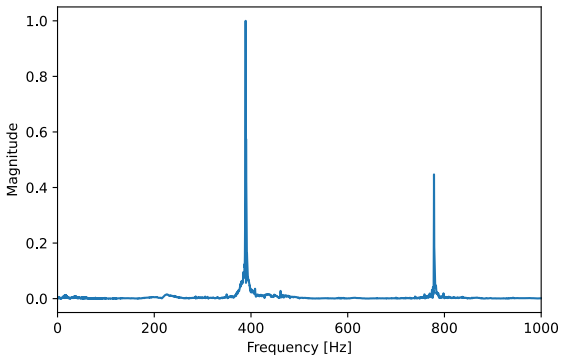


Figure 2.8: Frequency-domain representation of a ukulele signal

By revealing specific frequencies, the Fourier transform gives us a representation of the signal in the frequency domain instead of the time domain. In the previous example, we have seen that quite different signals can have similar appearance in the time domain, making them difficult to tell apart. The transformation from the time domain into the frequency domain gives another perspective on the signals in which they no longer appear as similar as before. This different perspective is shown in Figure 2.9. Instead of showing similarities between signals, the frequency domain emphasises their differences. This can be seen in Figures 2.7 and 2.8.[3] Even though both audio tracks have similar appearance in the time domain, they are much different in the frequency domain. Acoustic guitar has many frequencies all over the frequency spectrum. Ukulele, on the other hand, only has two main frequencies peaking at about 400 and 800 Hz.

Since we are working with discrete audio signals (consequence of sampling and quantization), there is a specific type of the Fourier transform to use, namely the **Discrete Fourier Transform**. Unlike the regular Fourier transform, DFT works with digital (i.e. discrete) signals, making it perfect for computer-based audio analysis. Fast Fourier Transform or FFT is an algorithm for efficient computation of DFT. Thus, in the context of digital (audio) signal analysis, the used method is FFT.

The result of a Fourier transformation of a signal is usually visualized with a **periodogram** which shows the distribution of power over the frequency components of a signal. Periodogram shows the frequency of the signal on the x-axis and the magnitude of the signal's spectrum on the y-axis. Two periodogram examples were already shown earlier in Figures 2.7 and 2.8. They are shown on a scaled scale that only goes up to 1 000 Hz but if plotted in their entirety, the frequencies would only go up to

[3]Both frequency spectrums were normalized to make the comparison more consistent.

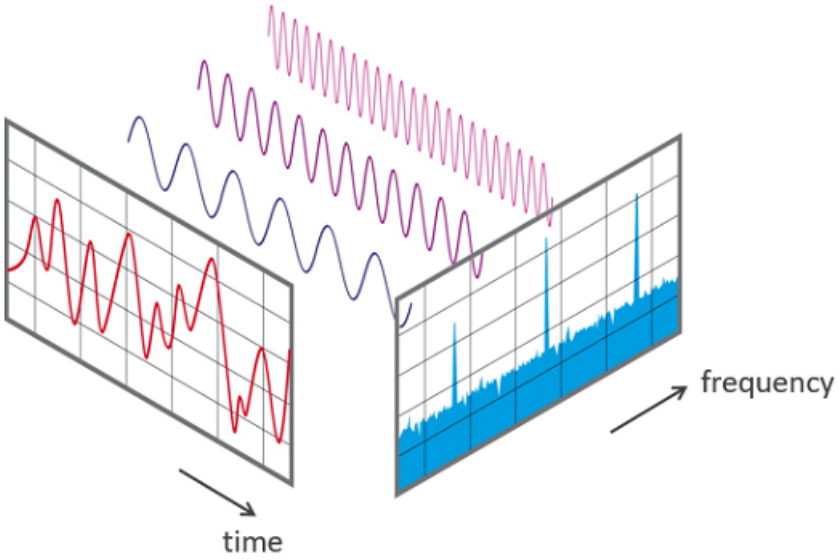


Figure 2.9: The idea behind the Fourier transform [26]

8 000 Hz. In other words, 8 000 Hz would be the highest visible frequency for these periodograms. There is a specific reason for this. In order to display a certain frequency range of a signal, there is one important condition that needs to be met: the Nyquist–Shannon sampling theorem. This theorem states that in order to successfully capture a signal, the sampling rate must be at least twice the maximum frequency that we wish to record. In other words, the highest capturable frequency of a signal is half of the sampling rate used. This is why many modern technologies use 44 100 Hz as the sampling rate for audio signals: it is about twice the frequency of the human hearing limit — roughly 20 000 Hz. Since audio signals shown in Figures 2.5 and 2.6 have been downsampled to 16 000 Hz, the highest frequency that can be obtained from them is precisely 8 000 Hz. More information on the idea of downsampling will be provided in Chapter 4.

3. Dataset

The dataset used for instrument classification comes from multiple sources and contains 12 categories in total. The categories are as follows: acoustic guitar, bass drum, violoncello, clarinet, double bass, flute, harmonica, hi-hat, saxophone, snare drum, ukulele, and violin/fiddle. There are 324 files in total among all these categories. On average, there are 24 audio files in each category with 10 and 30 being minimum and maximum, respectively. Total length of all audio files is 1544.90 seconds with the per-category average of 128.74 seconds. The distribution is shown in Figure 3.1 and in Table 3.1.

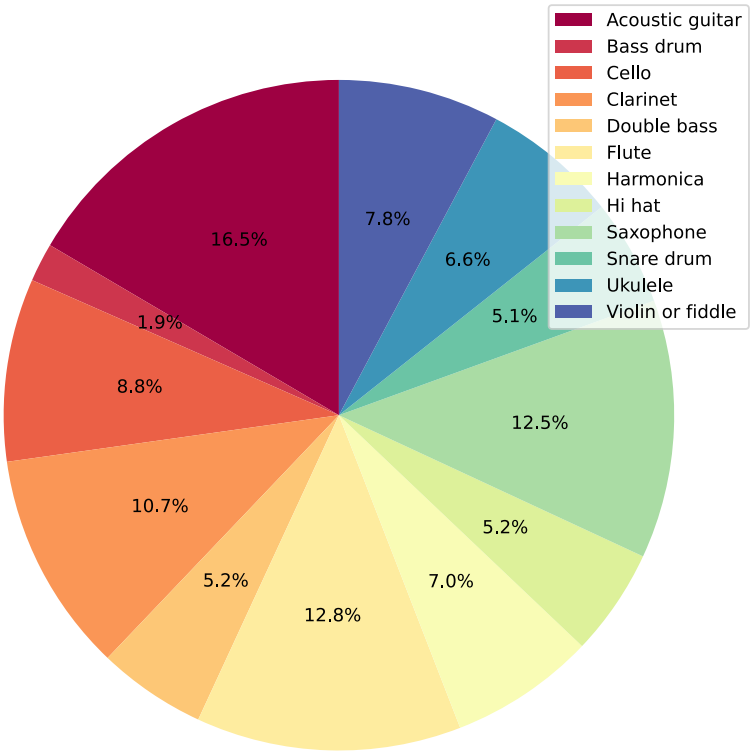


Figure 3.1: Visual representation of the class distribution

Table 3.1: Tabular representation of the class distribution

Instrument	Duration (seconds)	Percentage
Acoustic guitar	165.71	10.73%
Bass drum	30.85	2.00%
Cello	145.74	9.43%
Clarinet	176.16	11.40%
Double bass	86.34	5.59%
Flute	211.22	13.67%
Harmonica	116.06	7.51%
Hi-hat	85.44	5.53%
Saxophone	205.95	13.33%
Snare drum	84.53	5.47%
Ukulele	108.33	7.01%
Violin/Fiddle	128.58	8.32%
Total	**1544.90**	**100**%

The largest amount of audio files is obtained from a general-purpose Kaggle dataset [19] [13]. Originally containing 41 different categories, 10 distinct instrument categories were used from this dataset.[1] Other two categories were manually collected and they include harmonica and ukulele.

All audio files come from Freesound and according to the description from Kaggle's website: "...because Freesound content is collaboratively contributed, recording quality and techniques can vary widely" [14]. This means that any potential dataset bias is automatically accounted for since the large number of people providing these audio files used different instruments and different recording devices. Any potential bias to a specific instrument (i.e. its characteristic acoustics such as unique timbre) is thus greatly diminished. The same reasoning can be applied to recording instruments since any potential characteristics of a single microphone that can affect prediction become irrelevant due to a large number of various microphones used.

All audio files are stored with the Waveform Audio File Format (WAV) extension. WAV is an audio file format commonly used for uncompressed audio. Indeed, audio files contained in the dataset are uncompressed 16 bit, 44 100 Hz, mono audio files

[1] These instrument categories were extracted and made available on GitHub, courtesy of Seth Adams [3]. Other parts of the code were helpful in understanding how audio classification process is implemented.

[19]. This means that for each audio file, there are 44 100 samples per second during the entire length of the track. Moreover, each audio track has a bit depth of 16 bits. Using knowledge of sampling rate and bit depth, an interested reader is encouraged to calculate the size of such an audio file of arbitrary length — for example, their favourite song.

4. Data Preparation

Before performing any data transformations, it is important to first prepare the data. In this context, "preparation" can be described as a set of steps that need to be performed before any relevant information can be extracted from the data. This is important because raw data is not suitable for direct use in machine learning applications. Besides the obvious lack of information in the time domain, the data could have missing values, noise, or other inconsistencies. As we will see, this is particularly true for the dataset used in this thesis. In the following sections, techniques for preparing and "cleaning" the data will be shown.

4.1. Downsampling

According to the Kaggle dataset description, all audio tracks are provided as uncompressed 44 100 Hz mono audio files. Some of the manually collected files did not match this description hence they were manually processed using open-source audio sofware Audacity [5].[1]

At this point, audio files can potentially be considered suitable for feature extraction. The problem, however, lies in the previously mentioned sampling rate of 44 100 Hz. It was made clear in Chapter 2 that the sampling rate should be about twice the maximum frequency of the original audio. If we imagine using available audio files directly with this sampling rate, then the maximum available frequency would be 20 000 Hz. Needless to say, musical instruments rarely reach frequencies of this magnitude. It was mentioned that Figures 2.7 and 2.8 from the previous chapter would only show frequencies below 8 000 Hz. This means that keeping audio tracks at 44 100 Hz is redundant because essentially no additional information (or very small amounts of it) will

[1] In reality, not all files needed to be precisely at 44 100 Hz. Using the downsampling technique, any audio file with a sampling rate larger than 16 000 Hz could have been automatically transformed to the appropriate 16 000 Hz. However, the preparation of these files was done manually nevertheless for the sake of pedantic uniformity of data.

be obtained from the rest of the frequency spectrum (roughly anything above 10 000 Hz). For this reason, all audio tracks are downsampled from 44 100 Hz to 16 000 Hz. This transformation will improve general performance in the following ways: it will greatly reduce the size of the dataset; it will make training and validation faster; and it will make real-life response time quicker. There are some drawbacks to performing downsampling such as reduced quality, but the effects of it will not be considerable. Furthermore, essentially all of the original frequencies and other relevant information will be kept. Therefore, in this case the benefits greatly outweigh the cost.

Downsampling is implemented using the Librosa package and the SciPy's `wavfile` module. Librosa offers the `load()` method that loads a WAV file and automatically resamples it to the given sampling rate. The method can thus be called with two arguments: the path of the WAV file; and the downsampling rate of 16 000 Hz. At this point, each audio file is stored as a Numpy vector of floating point values representing audio tracks. Using wavfile's `write()` method, this vector of samples is written back into a new file by providing the path of the new (downsampled) file, the sampling rate, and the data vector itself. This process is iteratively repeated for all files of the dataset. The end result is a new dataset with each audio file having a sampling rate of 16 000 which is the exact value that will be appropriate for future feature extraction.

4.2. Signal Envelope

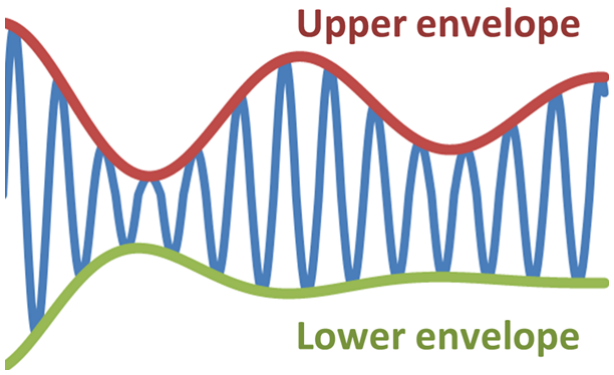


Figure 4.1: Envelope of a signal [25]

As mentioned earlier, all files used in this thesis were obtained from a variety of different sources which means there is no uniformity among them. While this is a benefit when it comes to combating dataset bias, it can be a hindrance when it comes to audio quality. Luckily, all audio files in this dataset have good playback quality.

What they're missing is a "clean" waveform including solely useful audio. In reality, many of these files contain noisy sections either before or after the actual instrument sound. This could be due to the way these files were recorded. For instance, leaving a microphone "on" for several seconds before playing the instrument means that the audio file is going to have some ambient static noise at the beginning. Ideally, we do not want this noise as it provides the model with absolutely no information. In fact, it could actually hinder the performance because many audio files might have similar ambient noise patterns that could be misinterpreted by the model and potentially cause faulty predictions. In order to minimize this potential problem, a technique known as **signal envelope** is applied to the raw data. An envelope of a signal is a curve that outlines its extremes in a smooth manner [18]. An example of an envelope is shown in Figure 4.1. It is useful for noise detection because it describes "behaviour" of a given waveform. If there is an oscillating pattern in a signal, the envelope will show it. If the intensity drops and the audio starts fading out, the envelope will show that too. For this reason, we can use the envelope to get the "behaviour" of a signal and to detect regions with low amplitude — these regions correspond to static noise. Thus, if an envelope of a signal can be obtained, all regions below a certain threshold can be considered as noise and discarded.

The envelope is implemented in the same place in the code as the downsampling technique from the previous section. After loading the WAV file and downsampling it, a vector of samples is passed to the custom `envelope()` method. This method creates a "rolling window" over the entire signal and obtains the highest value. This means that several consecutive values are considered together and a maximum is taken to be the point of the envelope. If each point is considered independently, that would have created too many distinct values and make it difficult to observe the actual pattern of the waveform. The final envelope is then filtered and all values below the threshold are ignored while keeping only those values above the threshold.[2] The method returns a computed envelope in the form of a boolean vector. After applying it over the original signal, all noisy portions of the signal are removed, leaving only relevant information to be further processed.

[2]For each set of audio files, the threshold can vary. This is something that is best determined experimentally.

4.3. Dataset Imbalance

Now that the audio files are almost ready for analysis, they are loaded into the memory. For each file of the dataset, two components are stored: its length and its class (i.e. instrument label). After processing all files, each class has a 'total' length obtained from all audio files belonging to it. For example, ukulele's 'total' length will be the sum of lengths of all ukulele audio files. For each instrument class, its total length is then divided by the sum of all audio lengths in the entire dataset. This produces a **distribution vector**.

The main reason for doing this is to counter the issue of **dataset imbalance**. This is a common problem in machine learning which can cause major issues during training. One such problem is the distortion of various metrics, such as accuracy. For example, if a dataset contains two classes with 10 and 90 percent representation respectively, the model could have an accuracy of 90 percent simply by classifying all input data as belonging to the second class. The model is trivial and hasn't learned anything, but it still has a high accuracy. This is a problem better avoided.

Such imbalance also exists in this dataset. Figure 3.1 from the previous chapter shows dataset's distribution in a pie chart. It is evident that some classes have a much lower overall representaton than others classes. For example, bass drum has 1.9% representation while acoustic guitar has 16.5%. Ignoring dataset imbalance would lead to the model being biased towards better represented classes purely because they contain more training examples. For this reason, the distribution vector is created as described earlier in this section. Each element of the vector corresponds to the distribution (i.e. percentage) of each class in the dataset. This distribution vector will later be used during model training. According to the Keras documentation, providing such distribution vector to the training method will cause the model to "pay more attention" to examples from under-represented classes, ideally alleviating the problem of dataset imbalance. In other words, the distribution vector is used to make note of those classes with low overall representation. Such classes will gain more attention, hopefully eliminating the imbalance problem.

5. Feature Extraction

Feature extraction is the process of transforming the original data into a more suitable form that can be used for machine learning. One consequence often associated with feature extraction is dimensionality reduction — reducing the amount of data while still accurately and completely describing the original dataset [10]. Feature extraction is important because original audio data often doesn't contain enough useful information for direct machine learning applications. Feature extraction is thus concerned with transforming input data into a set of structures that can be used for training the machine learning model.

Feature extraction is also used during the prediction phase. Since the model was trained on audio features, it can only predict instruments based on their features and not on the raw audio. The fact that the prediction phase also needs to have access to the feature extraction mechanism will become more relevant in Chapter 7. The main focus of this chapter will be on feature extraction during the training phase and in Chapter 7, feature extraction in the prediction context will come into focus. In subsequent sections, we will discuss steps that need to happen to transform raw audio data into appropriate machine learning features.

5.1. Specifics of Audio Features

In the previous chapter, the dataset was processed and prepared for analysis. At this point, audio files are ready to be used for feature extraction. It is important to realize that audio is a slightly different data format than other common formats in machine learning such as images or text. This is because they don't require as much processing as audio does. Images are oftentimes directly given to a (convolutional) neural network which performs the analysis of incrementally complex patterns. Starting from edges and other simple features, the model builds on the complexity of features and is eventually able to recognize the entire image. In case of audio classification, however, the data itself needs to be largely modified before the model can even begin training

on it.

Thus, audio is different because it requires additional steps *before* the data is passed to the model. Raw sound cannot be passed to the model directly because obtaining features from amplitude tracks is unfeasible. Too many types of audio contain similar representations in the time-domain, as was shown earlier in Chapter 2. That's why a different approach altogether is required for audio analysis. First, it has to be decided upon a clever way of extracting features. Only then can these features, in their appropriate shape, be passed to the neural network which will perform classification. In other words, the data needs to be prepared in such a way that differentiating various sound sources (in this case, musical instruments) can be easily done from the extracted features.

5.2. The Mel-Frequency Cepstral Coefficients

The Mel-frequency cepstral coefficients (or MFCCs for short) were introduced by Steven Davis and Paul Mermelstein in 1980 and have been state-of-the-art ever since [22]. Formally, the cepstrum coefficients are the result of a cosine transform of the logarithm of the short-term energy spectrum expressed on a Mel-frequency scale [9]. Less formally, MFCCs are a small set of features that can accurately represent original audio data. They are based on the frequency spectrum of audio (obtained through the Fourier transform) and on the conversion of frequencies from the regular frequency scale into the Mel scale. All these concepts will be explored in great detail in this chapter.

Throught modern machine learning history, different features have been proposed for audio data. Chapter 2 showed how different sources of audio have rather different characteristics which are "hidden" in the frequency domain. The goal is to employ a particular feature extraction mechanism that can "uncover" those features and emphasize them. By doing so, the model will be able capture relevant differences between different types of audio sources (in this case, musical instruments) and learn how to recognize them. The features that have proven to be very effective are the Mel-frequency cepstrum coefficients. As with both the phonautograph and the phonograph, the MFCCs were inspired by the human hearing system.

5.3. Audio Preparation

Information contained in audio files is constantly changing through time. To be able to capture those subtle changes, each audio track is divided into smaller **frames**. For those frames, it can be presumed that audio doesn't change a lot, at least on average. For example, a 20 millisecond audio frame is generally going to be stationary. The amplitudes or frequencies will change only slightly. Thus, it can be assumed that small enough audio frames are generally going to be constant. However, in order to capture enough information from them, frames need to be large enough to provide useful audio information. There is a balance at play here since frames need to be short enough to appear stationary but long enough to contain relevant information. A typically used value is 20 milliseconds [27], although this value can be somewhat different as it will be shown later in this chapter. This value is meant simply as a reference point to understand how short these frames actually are.

Before proceeding with this information, one question needs to be answered. How are these small audio frames eventually going to be delivered into the neural network? It is mentioned in the previous paragraph that each of the frames appears to be constant as no information contained in them is changing through time. However, analyzing an audio track requires the knowledge of how information changes through time. Any kind of audio is constantly changing and each small frame of audio is directly interconnected with other nearby frames. For this reason, the network is not trained on the frames themselves. Instead, the original audio is first split into 100-millisecond **blocks**. Each of those blocks is then split into frames. Such a setup accomplishes two things. First, it allows the model to learn various audio patterns (e.g. frequency) as they are changing through time. Second, it establishes a fixed-length input to the model. The second point will be discussed in more detail in the next chapter. For now, the only thing worth noting is that each 100-millisecond block of audio (composed of multiple frames) is given to the model as one input example.

5.4. Performing the Fourier Transform

Because the sampling rate is 16 000 Hz, 100 milliseconds of audio contain exactly 1600 individual samples. For reasons mentioned earlier, each block is split into smaller "constant" frames. The main idea is to now perform the Fourier transform on each frame. This allows us to extract relevant information from audio and obtain its frequency spectrum. To extract the spectrum from a frame, Discrete Fourier Transform

(DFT) is used. Briefly mentioned in Chapter 2, DFT is the discrete version of the Fourier transform. Since the audio we're working with is digital and discretized, DFT is the perfect choice. Performing DFT on each frame results with a periodogram. To recapitulate this concept from Chapter 2, periodogram represents the frequency spectrum of some signal (in this case, frame). It shows the frequency of the signal on the x-axis and the magnitude of the signal's spectrum on the y-axis. There is no time variable here since the signal is considered to be stationary — periodogram shows the entire frequency spectrum of the signal regardless of its length. An example of a periodogram is shown in Figure 5.1. Note that even though the periodogram shows an entire song, it still contains no information about how its frequency spectrum changes through time.

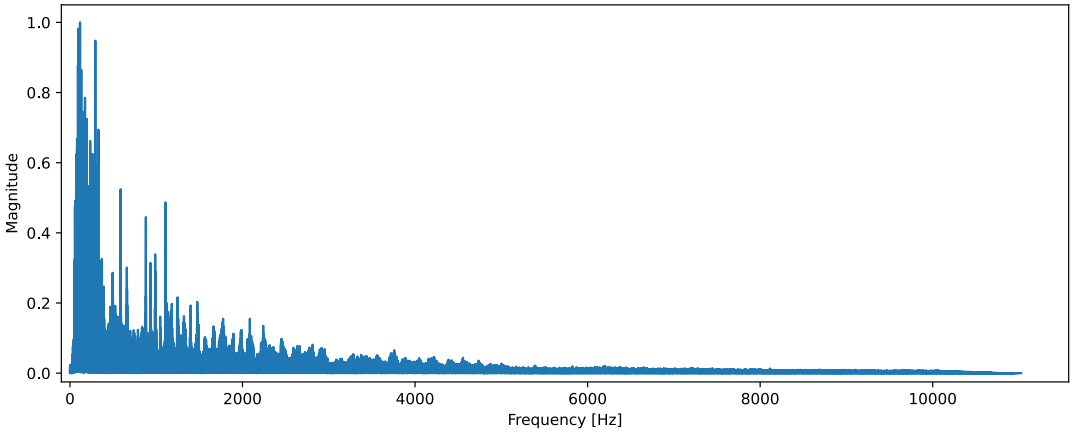


Figure 5.1: Example of a periodogram obtained from a song

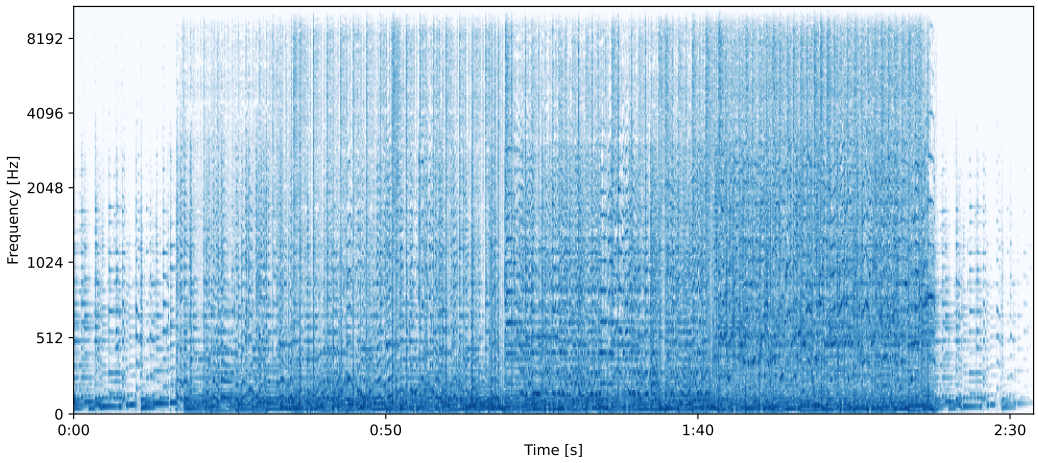


Figure 5.2: Example of a spectrogram obtained from a song

To correctly analyze audio and its frequencies, it is necessary to obtain the information of how the frequency spectrum changes through time. Since periodogram shows a stationary spectrum of frequencies for each frame, this is not enough. For this reason, the concept of a **spectrogram** is introduced. Unlike periodogram, which is stationary, spectrogram shows how a signal's spectrum (i.e. its magnitude) changes through time. An example of a spectrogram obtained from a full-length song is shown in Figure 5.2. To obtain a spectrogram, multiple periodograms can be combined to produce the representation of a signal's spectrum as it varies through time.

In other words, periodograms can be computed for short subsequent frames and then combined to produce a spectrogram for the entire block. Instead of performing DFT for each individual frame, a common technique in audio analysis is to perform something called the **Short-time Fourier transform** (STFT). The relationship between DFT and STFT is perfectly analogous to the relationship between periodogram and spectrogram. Periodogram is computed with DFT, resulting in a frequency spectrum for each frame. STFT combines periodograms for multiple frames contained in a block and produces a spectrogram — the frequency spectrum of the entire block. STFT thus represents the Fourier transform of an entire 100-millisecond block and its result is a spectrogram which shows how the frequency spectrum changes through time.

To compute STFT, **frame length** needs to be determined. Frame length is usually 20 milliseconds which corresponds to 400 samples at 16 000 Hz. However, in order to apply DFT on a frame, frame length needs to be a power of two. DFT doesn't strictly require frame size to be a power of two but the underlying FFT implementation (the Cooley-Tukey algorithm) performs in $O(Nlog(N))$ complexity while working with powers of two instead of $O(N^2)$ while working with arbitrary lengths. Specifying frame length to be a power of two makes the algorithm perform much more efficiently [7].

In this thesis, 512 samples per frame are used. To obtain 512 samples from 400, padding is performed. Padding is sometimes performed by adding zeros at the end of each frame but in this case, the underlying implementation performs **reflection padding**. Instead of padding individual frames, the entire 1600-sample block is padded by reflecting frames until reaching a length of 2112 samples. At this length, the entire block can be divided into 512-sample frames with the 160-sample overlap (10 milliseconds) between each two frames. Overlapping is introduced to avoid sharp an unnatural changes in audio. Instead of immediately jumping to another disconnected frame, overlapping smoothens out the transition between two consecutive frames. When a 2112-sample block is split into 512-sample frames with a 160-sample overlaps, 11

frames are obtained per each block. This is determined with the following expression:

$$noFrames = 1 + \text{int}(\text{ceil}((blockLength - frameLength)/frameStep)).$$

Result of the framing operation is a (512, 11) matrix which represents 11 frames, each containing 512 samples of the original audio signal. Each of the 512-sample frames corresponds to one individual frame, whereas the entire matrix corresponds to 11 frames, or an entire block. After framing the input block, STFT is applied on the (512, 11) matrix resulting in 257 samples for each of the 11 frames, i.e. an (257, 11) matrix for the entire block. The number 257 comes from the fact that the internal FFT algorithm works with complex numbers. It turns out that negative-frequency terms are complex conjugates of the positive-frequency terms and thus the negative values are discarded. The output shape is half of the input shape, or in this case 257. The resulting (257, 11) matrix represents the spectrogram of an entire block.

An example of a block spectrogram is shown in Figure 5.3. The x-axis represents time while the y-axis represents frequency. Spectrogram essentially shows how frequencies change through time. If looked closely, 11 distinct columns can be recognized. Each of the columns represents the frequency spectrum of a single frame. This spectrum is in fact a periodogram and can be recognized if looked sideways: spectrogram's y-axis represents periodogram's x-axis and the straight lines represent frequencies. Instead of each frequency having a magnitude as in Figure 5.1, in the case of a spectrogram the magnitude is denoted by the shade of the blue color. Darker blue represents higher energy frames while lighter blue represents lower energy frames.[1]

[1]Shades of the blue color are in fact a projection of three-dimensional magnitudes on a two-dimensional plot. Magnitudes are still visible as in a periodogram but a two-dimensional projection with colors is easier to visualize than a three-dimensional graph with values

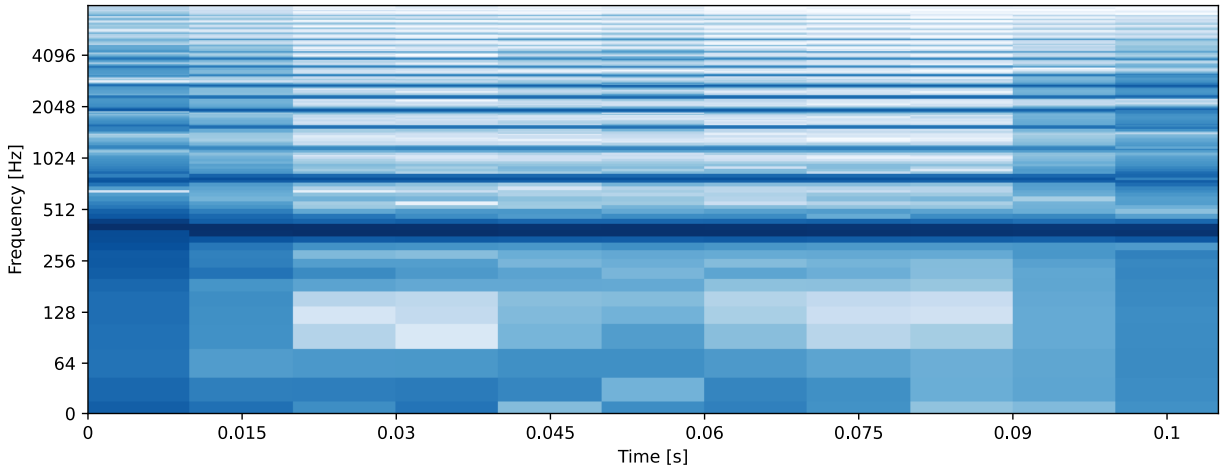


Figure 5.3: Decibel spectrogram

A regular linear spectrogram can be seen in Figure 5.4. Clearly, the linear spectrogram doesn't visually contain as much information as the decibel spectrogram. This is because most sounds in music (and in human nature in general) are limited to very narrow frequency and amplitude ranges [15]. For this reason, logarithmic scale is applied to the spectrogram, resulting in a decibel spectrogram which is shown in Figure 5.3.

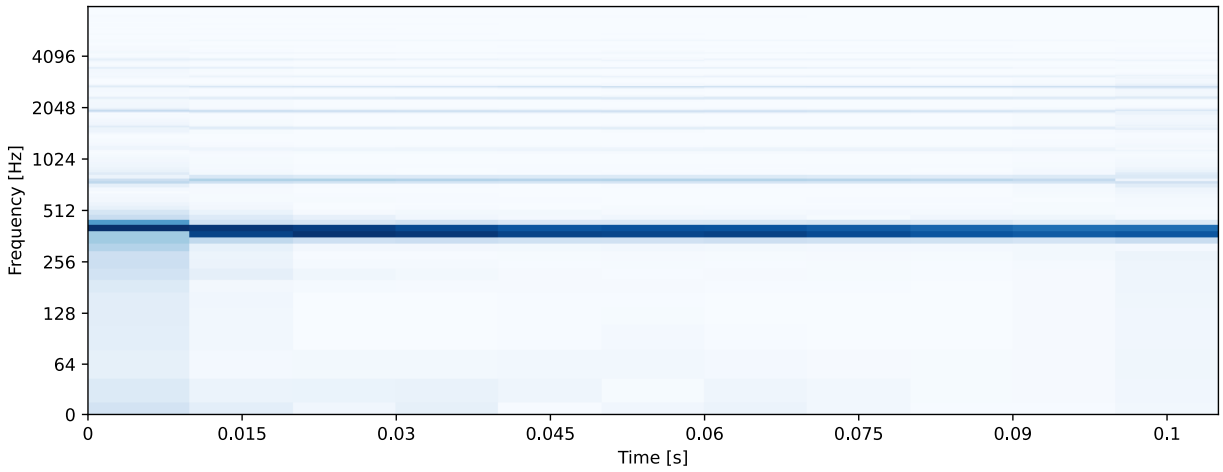


Figure 5.4: Linear spectrogram

5.5. The Mel Scale

Earlier in Chapter 2, cochlea was described as a part of the ear that converts vibrations into nerve impulses that get sent to the brain. Depending on the exact frequencies,

cochlea vibrates at different places and uses different nerve impulses to send frequencies to the brain. It could be said that cochlea's purpose is therefore to convert raw audio input into a spectrum of frequencies. In this sense, cochlea effectively acts as a real-life periodogram. The only major difference is the fact that cochlea isn't able to differentiate between two similar frequencies. This might not be obvious for low frequencies but it certainly is for higher ones. Upon hearing two low frequency-sounds (e.g. 100 and 200 Hz), the difference between the two frequencies is noticeable. Upon hearing two high frequency-sounds with the same spacing as before (e.g. 10 100 and 10 200 Hz) the difference becomes extremely difficult to discern. Generally, this effect of frequency differences becomes more pronounced as the frequencies increase. Put differently, human hearing is non-linear with respect to frequency. Audio doesn't have this property naturally as non-linearity is merely a subjective experience of the observer (in our case, humans). In order to make the audio features match more closely what humans actually hear, the audio is transformed using the **Mel scale**. The Mel scale is a pyschoacoustic scale that tries to capture distances from low to high frequencies. It makes frequencies "sound" as they are equal in distance from one another, unlike the regular linear scale where this is not the case [15].

This enables us to rescale the audio to mimic the non-linear human perception of sound by being more discriminative at lower frequencies and less discriminative at higher frequencies [12]. To perform the rescaling of audio, certain frequencies are "clumped together" into **bins** in increasing intervals [22]. These bins start as narrow (at the left — low end of the spectrum) and become wider as the frequencies increase (towards the right — high end of the spectrum).

The "clumping together" of certain frequencies is achieved with **Mel filters**. Each filter is a triangle pointing upwards that gets placed over a certain range of frequencies. These filters are placed sequentially over the entire frequency spectrum — this is known as the **Mel filterbank** and can be seen in Figure 5.5. The Mel filterbank partitions the Hz scale into bins and transforms each frequency bin into a corresponding Mel bin with the help of the triangular filters [15].

To determine shapes and positions of the filters, the Hz scale is transformed to the Mel scale using the following expression:

$$M(f) = 1125 \cdot ln(1 + f/700).$$

Furthermore, since humans don't perceive loudness on a linear scale either, logarithm is applied to the obtained filterbank energies. These two transformations (frequency and loudness) makes the audio features match more closely what humans ac-

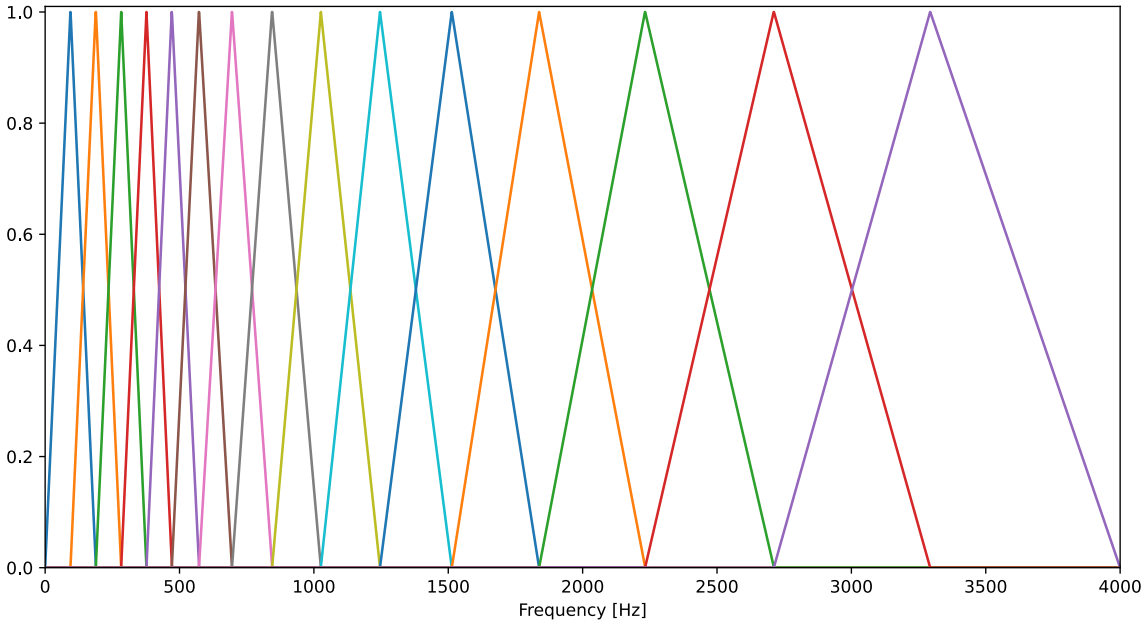


Figure 5.5: Mel filterbank

tually hear.

In total, 128 triangular filters are used in the filterbank. Each filter is a vector of 257 samples. The dimension matches the previous result of the STFT operation. This is because each filter is "placed over" the frequency spectrum of each frame in order to capture the amount of energy for each of the filters. Mathematically, this is performed by multiplying the two vectors. More precisely, to get the Mel filterbank for a particular frame, each filter is dot multiplied by each frame. This means that each of the 128 filters of size 257 is multiplied by the frame's frequency spectrum of size 257. The result is a single number determining the amount of energy present at the location of a particular filter. For each frame, a set of 128 numbers are obtained. Since there are 11 frames in total, a (128, 11) matrix holds the Mel filterbank information for the entire block in a spectrogram known as the **Mel spectrogram**. An example of a Mel spectrogram can be seen in Figure 5.6.

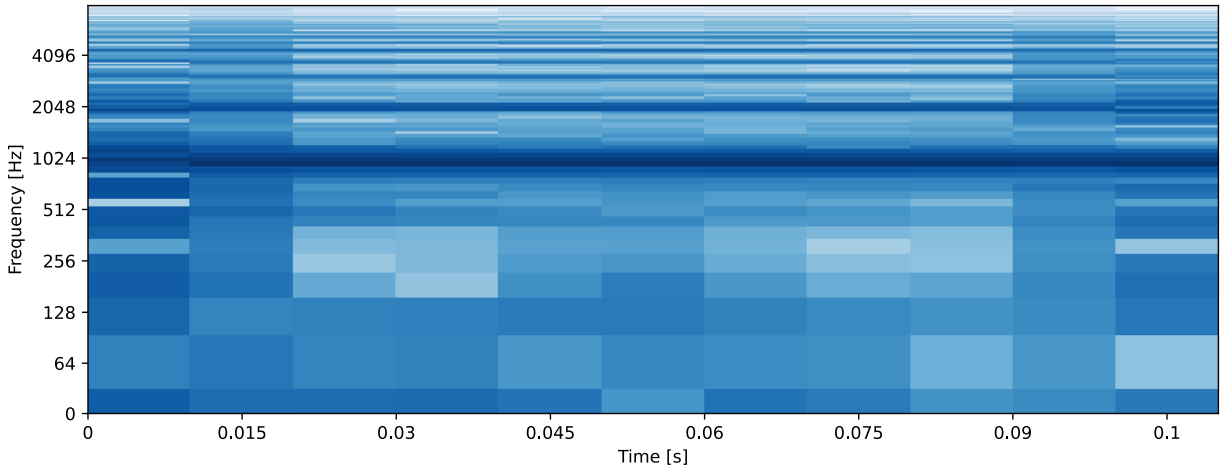


Figure 5.6: Mel spectrogram

5.6. Discrete Cosine Transform

After obtaining the Mel spectrogram, the final step is to apply the Discrete Cosine Transform (DCT) which is performed in order to reduce the number of features. In other domains, techniques such as the Karhunen-Loeve (KL) transform or the Principal Components Analysis (PCA) are used, but in the sound domain the KL transform is approximated by DCT [21]. For each of the 11 frames, 13 cepstral coefficients are obtained with the Discrete Cosine Transform. The reason why 13 coefficients are kept is because higher DCT coefficients represent fast changes in the filterbank energies. Keeping only the lower coefficients is beneficial because the higher ones can hinder the performance for sound analysis problems [22]. The result of the DCT finally represents the Mel-frequency cepstral coefficents — MFCCs. Since DCT is performed on the entire block of 11 frames, the result is a (13, 11) matrix. Each (13, 11) matrix holds the extracted features of the original 1600-sample audio block. A MFCC matrix example is shown in Figure 5.7. This matrix is the final result of feature extraction.

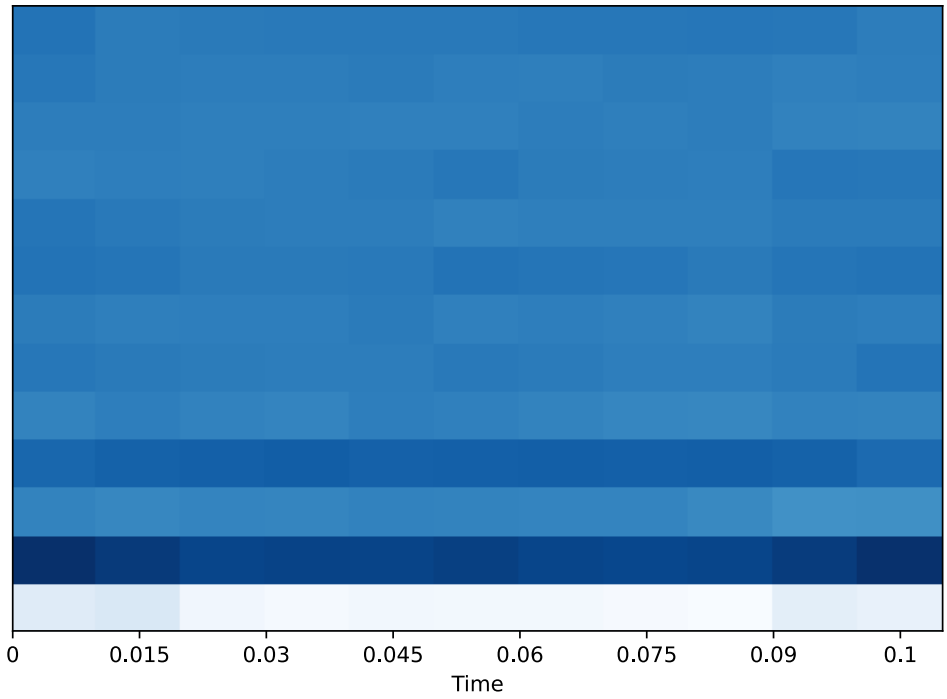


Figure 5.7: MFCC matrix — the final set of features

5.7. Performing Feature Extraction in Python

The first step of actually performing feature extraction is to load the dataset into main memory. The way each file is loaded is as follows. From the distribution vector discussed in the previous chapter, a single class is randomly chosen. From this class, a single file is further chosen randomly. After the file has been loaded into the memory, it is represented as a vector of individual amplitude samples. From these samples, a random block of 100 milliseconds in length. Since the chosen sampling rate is 16 000 Hz, 100 milliseconds of time corresponds to 1600 audio tracks. In other words, from the entire vector representing an audio file, a small sub-vector of length 1600 is selected. Such a setup should then be able to correctly classify an instrument from a tenth of a second of the original audio recording. In reality, a more robust way of performing classification would be to take a much longer audio recording, say 1 second. This recording can then be split into 10 parts, each 100 milliseconds long. Feature extraction and classification can then be performed on each 100-millisecond block and combine the results (by averaging them) to get a more robust classification. Analysis and details of classification will be discussed in Chapters 6 and 7.

At this point, each block is delivered to the feature extraction mechanism which

is based on Mel-frequency cepstral coefficients (MFCCs). The main logic behind the MFCC features comes from the Librosa library in Python. Each block is passed to the `mfcc()` method which performs the main feature extraction logic as explained in the previous sections. For each 1600-sample input, the result is a (13, 11) MFCC feature matrix. Each matrix becomes one training example for the neural network model. Since each block is extracted from an audio file (which essentially represents a musical instrument), each feature matrix can be labeled as a particular instrument. Such (feature matrix, instrument label) pairs are passed to the model as input-output pairs. The learning process is therefore based on **supervised machine learning** where each input example contains a corresponding output value. The training process is therefore an iteration over thousands of available (feature matrix, instrument label) pairs until the model has learned to discriminate among features describing each particular instrument.

The next chapter will describe the machine learning architecture used for instrument classification.

6. Building the Learning Model

In the previous chapter, features were extracted from audio resulting in one (13, 11) matrix for each 100 milliseconds of original audio. This matrix represents a 1600-sample audio block of the input audio and is used as one learning example for the model. In other words, to train the model on a 1600-sample audio, we provide it with a (13, 11) matrix containing the MFCC features. This has the great advantage of having a fixed-length input to the model. No matter how long, original audio can be split into 100-millisecond blocks to perform the prediction. Of course, the original audio needs to be longer than 100 milliseconds, but this is almost always the case in practical applications. Therefore, to predict any audio recording, it is sufficient to split it into 100-millisecond blocks. From each block a set of features is extracted, as explained in the previous chapter. The result is a (13, 11) matrix for each block. This is precisely the input dimension to the neural network model. After a series of inner-propagations, a (1, 12) vector is returned by the model. Each element of the output vector represents the probability that the input audio "belongs to" a particular instrument class. Since there are 12 classes in total, there are also 12 elements in the vector. This process is sequentially repeated for all 100-millisecond blocks of the input audio track. The result is a set of probability vectors which, after taking the mean value of, will represent the most accurate description of the audio track with respect to the instrument classes. This chapter will discuss the training and prediction phases of the machine learning for instrument classification.

The model itself is built using TensorFlow 2.0 and Keras. Several different models were considered but the decision was ultimately made to proceed with the convolutional model. A good strategy would have been to try multiple different models and choose the one with the highest validation accuracy. However, due to remarkably good results of the convolutional model, it was nevertheless decided to use this model instead of seeking potentially better alternatives. The results are displayed later in this chapter in Section 6.3.

The reason why the convolutional model is able to perform so well is because

each audio track has been transformed into a MFCC representation. Since MFCCs can be treated like an image (as shown in the previous chapter in Figure 5.7) and since the convolutional model is a standard architecture for image recognition, it makes sense to use a convolutional model to classify MFCCs. Performing feature extraction essentially converts an audio track into its image representation of MFCC features. This image representation can then be delivered to the convolutional model which performs classification. [11].

6.1. Architecture of the Convolutional Model

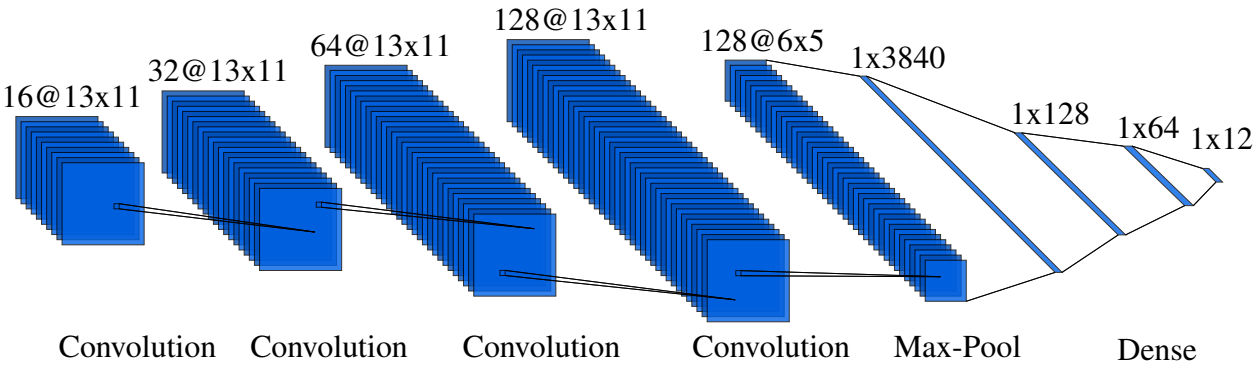


Figure 6.1: Architecture of the Convolutional Model

The architecture of the model can be seen in Figure 6.1. First, four convolutional layers are added to the architecture.[1] The input MFCC matrix (in further text — image) becomes the input of the first convolutional layer which contains 16 kernels. All kernels in the network are of size (3, 3) and each of them is convolved with the image, producing a feature map that contains various interesting patterns found in the image. Since there are 16 kernels in the first convolutional layer, 16 feature maps will be generated. These feature maps will then be used as the input for the next convolutional layer. In the second layer, 16 kernels will produce 16 new feature maps. This trend of increasingly higher number of kernels continues until the final convolutional layer with 128 kernels. The reason why the number of filters is progressively increased is because this setup allows the model to learn more abstract information in each successive layer. The first layer will be able to capture "low-level" information by combining multiple pixels. The second layer will not combine pixels but instead features generated by the first layer. This clearly shows an increase in the abstraction of features.

[1] All convolutional layers imply a (1, 1) stride, "ReLU" activation, and "same" padding.

This process continues until the final layer is operating on features that have several layers of abstraction: features of features of features, etc.

After the convolutional layers have extracted relevant features from the MFCC image, maximum pooling is performed. The purpose of the `MaxPool2D` layer is to reduce dimensionality and prevent over-fitting. After pooling, a dropout layer (left out of the diagram for the sake of brevity) is added with the value of 0.5. The dropout layer randomly "drops out" a number of units so that they are unable to contribute to further training. In this case, 50% of units are dropped out. This is done to preventing over-fitting and to make the classification more robust. If 50% of the units are disabled, then the remaining 50% need to "work harder" to compensate for that loss. The remaining units will thus try to perform the correct classification themselves which makes each of them more reliable. After returning all 100% of the units in the prediction phase, the model will display much accurate results than if it weren't for the dropout layer. The flatten layer takes the output of the previous layer (a tensor) and flattens it into a 1-dimensional array. Doing this prepares the data for subsequent classification by transforming all previously collected features into a single long vector.

Finally, a series of dense layers are added at the end. Unlike convolutional layers which increased in complexity, the purpose of dense layers is to reduce the complexity of data and create a final prediction that can be represented in simple "human" terms. Each dense layer represents a classic fully-connected layer where each unit from the current layer is connected to every unit from the previous layer. Such a setup allows the network to perform a generalization of features that has hitherto uncovered. By progressively decreasing the number of parameters in each successive layer, those features describing a certain output class are being "clumped together". This is done by performing a linear combination of units and weights from the previous layer. The result is passed through a ReLU activation function which introduces non-linearity to the network. Non-linearity allows the network to perform more complex calculations than it would have otherwise. In other words, it allows a much more accurate and abstract portrayal of features because of the complexity introduced by non-linear combinations of elements in each layer.

The final layer contains 12 elements and its output is a (12, 1) vector. This vector is the result of the softmax function which normalizes the vector and produces a probability distribution. This probability distribution is the final result of the neural network. Each element represents the probability that the original audio track belongs to a particular instrument class. In other words, each element of the output vector denotes how likely it is that the original audio track is a recording of each particular

instrument. For example, 'Acoustic guitar' corresponds to the first element since instruments were sorted alphabetically. If the first element of the vector has a value of `0.42` then there would be a 42% probability that the instrument is acoustic guitar.

6.2. The Training Process

From the entire dataset, 77 240 different examples (blocks) are used during the training process. This value was obtained by multiplying the total number of blocks in the dataset by 10. This 'total number of blocks' was obtained by splitting each audio track from the dataset into 100-millisecond blocks and then adding up the number of all such blocks in all files in the dataset. This might not be the most accurate representation, but it serves as a good approximation of "how many blocks there are in the dataset".

51 750 of these examples are used in the training set, and 25 490 of examples are used in the validation set. The validation-training ratio is thus 33%. There are 10 epochs in total and 77 240 examples are used entirety in each of the epochs. Batch size is 32 examples which means that model's weights are updated after training on 32 examples at once.

Feature extraction is performed on the CPU, while the training process has been optimized for running on the graphics card with Tensorflow GPU. The entire process (feature extraction + model training) takes no more than several minutes on Intel i5-9300H CPU and Nvidia GeForce GTX 1050 GPU.

6.3. Training Results

Model's training history is shown in Figures 6.2 and 6.3 which display accuracy and loss, respectively. From these figures, it appears that validation accuracy is higher than training accuracy, while the opposite is true for loss. This might seem counter-intuitive, but since the model uses a 0.5 dropout during training, 50 percent of the features are set to zero. During validation, all features are used which makes the model more powerful and leads to a better accuracy score. The final validation score is 97.46% which seems quite remarkable given that the model was trained on such short audio blocks of 100 milliseconds. Figure 6.4 shows the model's confusion matrix while Table 6.1 shows precision, recall, and F1 scores.

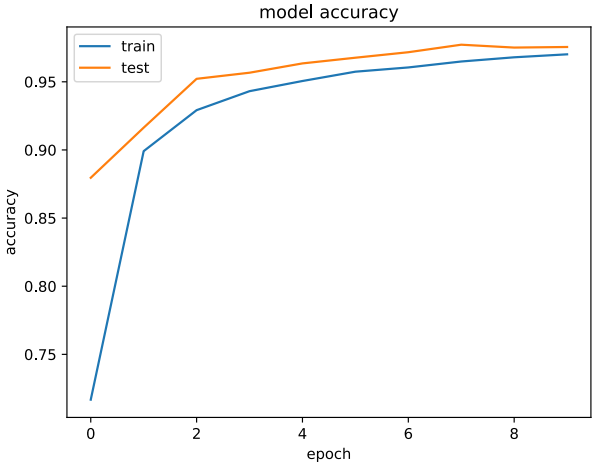


Figure 6.2: Validation and training accuracy of the convolutional model

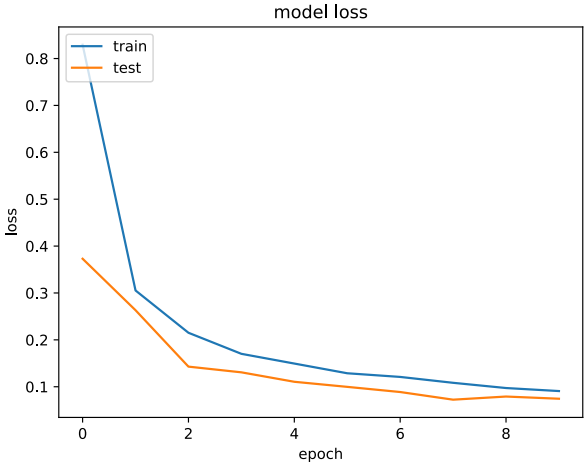
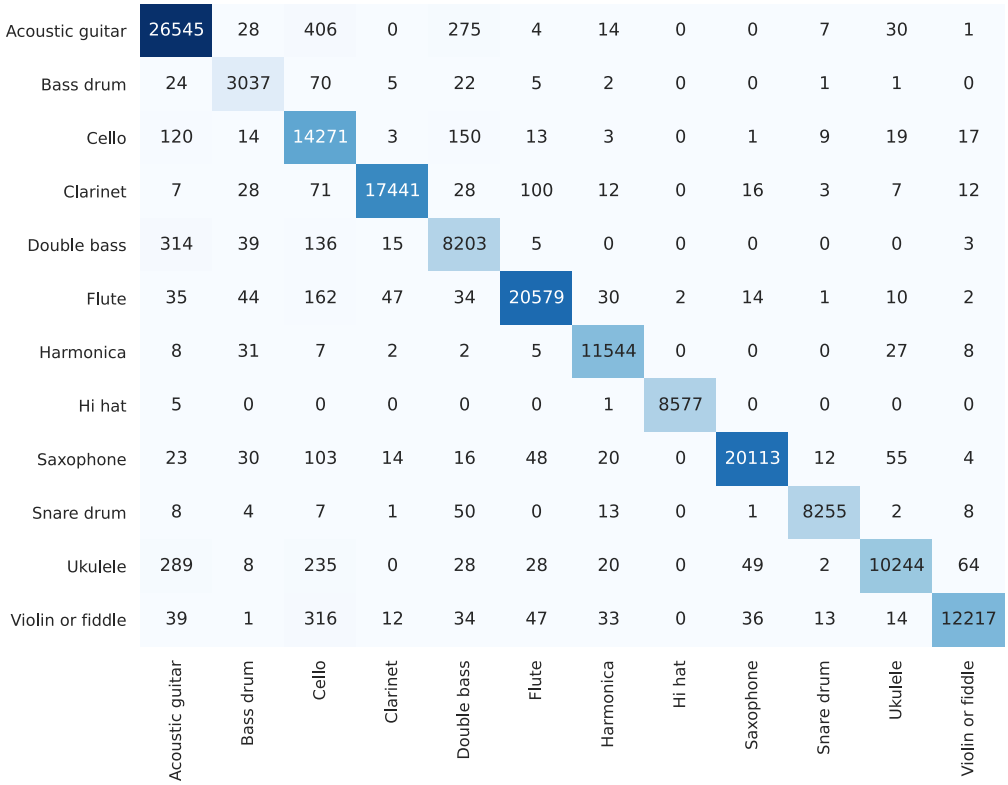


Figure 6.3: Validation and training loss of the convolutional model

Table 6.1: Precision, recall, and F1 scores

Instrument	Precision	Recall	F1
Acoustic guitar	0.9681	0.9719	0.9700
Bass drum	0.9304	0.9589	0.9444
Cello	0.9041	0.9761	0.9387
Clarinet	0.9943	0.9839	0.9891
Double bass	0.9277	0.9412	0.9344
Flute	0.9877	0.9818	0.9847
Harmonica	0.9873	0.9922	0.9897
Hi-hat	0.9997	0.9993	0.9995
Saxophone	0.9942	0.9840	0.9891
Snare drum	0.9942	0.9887	0.9914
Ukulele	0.9841	0.9340	0.9584
Violin or fiddle	0.9903	0.9572	0.9735

Figure 6.4: Confusion matrix for the convolutional model



7. Building the Mobile Application

The second part of this thesis is the Android application. The main focus here is to create a standalone mobile application that will be able to work in a real-world scenario. The goal is therefore to deploy the machine learning model in such a way that it will be able to work on the mobile platform without any additional resources. For example, a perfectly acceptable solution could have been to create a mobile application which, instead of having a local prediction model, would delegate all predictions to a dedicated server. The application would communicate with this server through an API (Application Programming Interface) which would receive an audio recording and send the prediction back to the application. Such a solution would require a dedicated server that has to be available at all times and to a number of users. In case of a large number of users, such a solution quickly becomes unscalable. Furthermore, the application would require a constant internet connection with the server, potentially using a lot of bandwidth and providing the results in a slower manner due to networking delays.

The best solution is to employ the application with a local version of the model. This way, prediction delay and server maintenance are no longer issues. Since there is no networking traffic, there is no networking delay. And since there is no server, there is no server to be maintained. The result is a standalone application able to work completely offline on any Android device.[1] Unlike many modern applications (especially those relying on machine learning), this one requires no internet connection whatsoever.

The first step in building the application is to decide upon the technologies that will be used. The most obvious distinction is that between Android and iOS. Developing iOS applications is often considered tedious due to a substantial amount of logistics. One needs to have an Apple operating system (i.e. MacOS), an Apple developer licence which is not free, knowledge of Apple's specific technologies, etc. Developing

[1] Since the minimum API level used in the application is 21, the device needs to be running at least Android 5.0 which became available on June 25, 2014.

Android applications is free, can be done on any operating system, and requires knowledge of Java. The decision was therefore made to focus on Android instead of iOS. However, Android applications can be developed in a number of different frameworks and the most common one is Java-based Android. This allows the developer to write Android applications in pure Java. Another way would be to use one of the lesser common frameworks such as Xamarin or Flutter. Several demo versions have been written in Flutter, however it quickly became obvious that the system wouldn't work as expected. This is due to a lack of support for the Dart language in the context of machine learning, and especially audio feature extraction. Flutter itself was only released in 2017, having a rather small number of such specific libraries. The final decision was therefore made to build the application in Java.

7.1. Research and Preparation

For developing the application, JetBrains' IntellijIDEA was configured for Android development. Before starting with the development, it is important to include relevant dependencies. In Python, Librosa library is used for MFCC feature extraction. Since there is no official Librosa implementation for Java, a thorough search (lasting multiple weeks) was done to find an appropriate implementation capable of working on the Android platform. Eventually, an MFCC implementation by NG et. al. was discovered in their book 'Mobile Artificial Intelligence Projects' [23]. Unlike other implementation, this one is a directly rewritten Librosa MFCC module from Python into Java. Feature extraction is therefore identical in Java and Python, making it possible to perform MFCC extraction locally on Android devices and deliver the results directly to the neural network. The results will be the same as if the feature extraction and prediction were done through Python.

Once the feature extraction mechanism is set-up, a machine learning framework needs to be added to allow for the model itself to run on the device. For this, a nightly-build of TensorFlow Lite is used. TensorFlow Lite is a deep learning framework that works on mobile devices and performs on-device inference. With TensorFlow Lite, no additional resources are needed (such as an external server) which makes it perfect for this thesis.

Using the rewritten Librosa library and the TensorFlow Lite framework, instrument classification can be performed in real-time on the mobile device itself. In the next section, we will go through the architecture of the Android application. Important parts of the codebase will be discussed as well as some key implementation parts.

7.2. Architecture of the Application

The application can be divided into three main parts (i.e. classes): `MainActivity`, `AudioRecorder`, and `AudioPredictor`. `MainActivity` represents the logic behind the main screen. `AudioRecorder` is the class used for recording the audio through the microphone. Finally, `AudioPredictor` is used for making predictions about a recorded audio track. `MainActivity` can be seen as a mediator between `AudioRecorder` and `AudioPredictor`. Oncer the recording process is started by the user, `MainActivity` delegates the recording process to the `AudioRecorder`. After recording, an audio track is returned to the `MainActivity`. Then, the recorded sound track is delegated to the `AudioPredictor` which returns a prediction vector with probabilities for each instrument. `MainActivity` then uses this vector to create its visual representation on the main screen. In the rest of this section, two most important parts of the architecture will be discussed in detail: `AudioRecorder` and `AudioPredictor`.

7.2.1. AudioRecorder

Recording is performed through the device's main microphone. This process is performed in a separate thread to avoid unnecessarily blocking of the main thread during audio recording.

For recording itself, Android's `AudioRecord` class is used. An `AudioRecord` object requires audio source information (microphone type), sampling rate (16 000 Hz), and several other configuration settings. Once started, the `AudioRecord.read()` method starts reading audio samples from the hardware. Since the method also requires a size, it can return an arbitrary number of samples. As in the training process, 100-millisecond block is used as the basic unit (block) which corresponds to 1600 samples when recorded at 16 000 Hz. However, the **inference length** can be modified in the application's preferences as will be shown in Section 7.4. Inference length is defined as the subset of an audio recording that will be treated as a single unit for making the prediction of an instrument. The default is 1 second, which means that 10 separate 1600-sample blocks will be processed before delivering the final prediction. The longer the inference length, more accurate the prediction. By deafault, `AudioRecorder` delivers a 16 000-sample recording to the `MainActivity`. The `MainActivity` receives the recorded array using the Observer design pattern. Once the `AudioRecorder` has finished recording, it updates the `MainActivity` with the freshly recorded audio track.

7.2.2. AudioPredictor

Once the recorded audio track has been delivered to the MainActivity, it is delegated to the `AudioPredictor` object. Audio predictor contains references to the TensorFlow Lite `Interpreter` and to the `MFCC` object. The MFCC object is constructed with the same parameters as during the training process. This is important because otherwise the resulting matrices would be different, making the prediction inaccurate.

By default, 16 000-sample recordings are delivered to the `AudioPredictor` (as per the default inference length). Internally, each audio recording is split into 1600-sample blocks which makes it possible to perform prediction as usual. Each block is delivered to the MFCC object which performs the extraction of features and returns a (13, 11) matrix. Each value of the matrix is normalized. This is important because each value needs to be scaled according to the minimum and maximum values from the training process. Otherwise, the final prediction would be inaccurate. The matrix is then reshaped to match the exact specifications of the input tensor to the model: (1, 13, 11, 1). Finally, the matrix (now tehnically a tensor becauase it has more than two dimentions) is passed to the TensorFlow's Interpreter which performs the actual inference. The result of the Interpreter is a softmax prediction vector with per-class probabilities for each of the instruments.

As mentioned earlier, 1 second is used as the inference length. After the model has made a prediction for a single 1600-sample block, 9 more predictions need to be made. This is done in a `for` loop which goes over all 1600-sample blocks that `AudioPredictor` received. When all blocks have been processed, a mean vector is computed from all prediction vectors and is returned to the caller — `MainActivity`. Once receiving the prediction vector, `MainActivity` calls appropriate methods for updating the visual aspects of the application.

7.3. Designing the User Interface

The user interface, like the codebase itself, is built from within IntelliJ. Since the design didn't require many complex features, most of the visual layout was done manually from within the code editor, while the control and inspection of elements was done through a built-in visual editor. The main screen can be seen in Figure 7.1. There are deliberately no complex features on the main screen because the minimalist design offers a simple and intuitive usage of the application [2]. There is a wrench icon which

[2]The design was inspired by the Apple's Music Memos application.

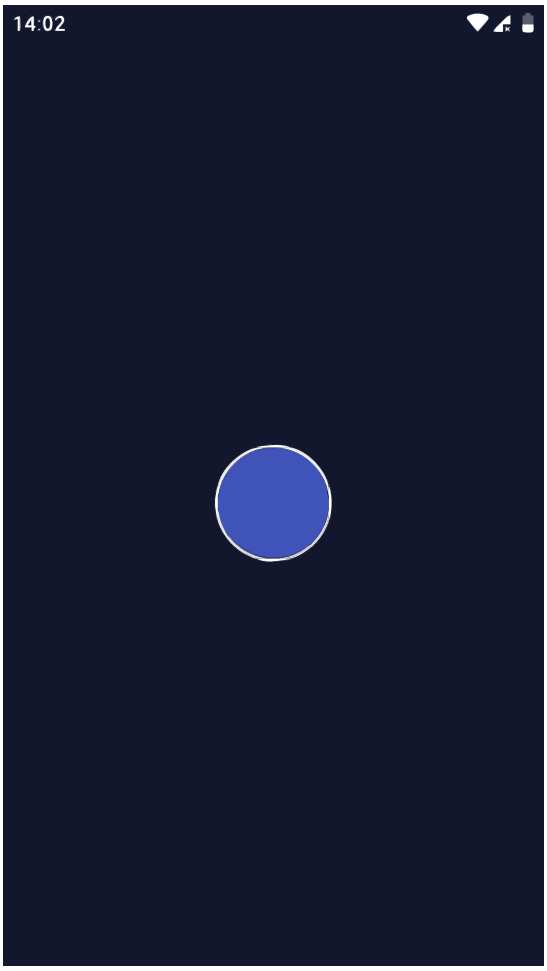


Figure 7.1: Main screen before audio recording

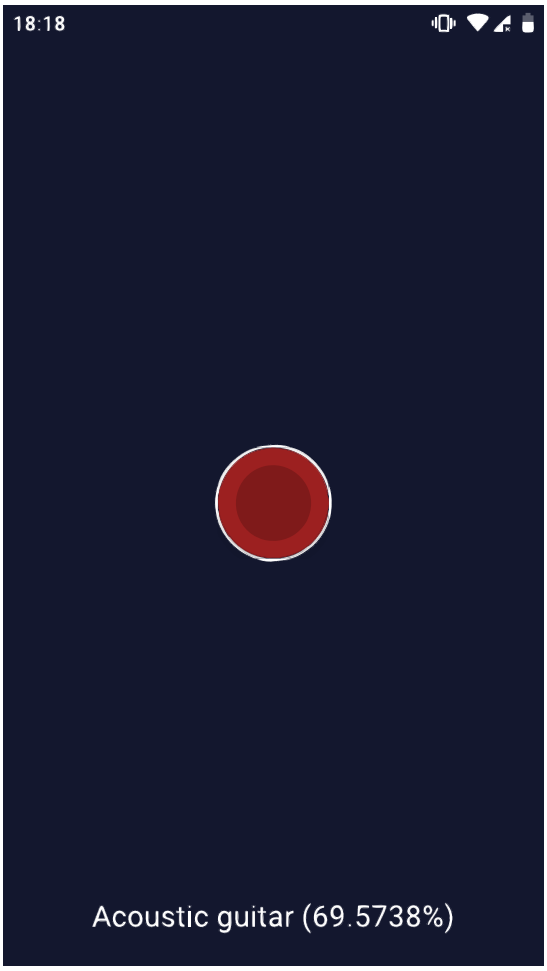


Figure 7.2: Main screen during audio recording

allows the user to navigate to the preferences screen — this will be explored in Section 7.4.

Audio recording is started by pressing the blue button in the middle of the screen. Once pressed, the button turns red to indicate that the recording process has begun. This can be seen in Figure 7.2. The bottom of the screen shows the current results of the prediction model.

7.4. User Preferences

After experimenting with different inference lengths, a decision was made to implement a ‘preferences’ screen in the application. At this screen, users would be able to customize the inference process by selecting one of the available inference lengths or keeping the default value of 1 second. Minimum inference length is 100 milliseconds

seconds since the model is trained on audio tracks of precisely 100 milliseconds in duration. This is therefore the shortest possible duration of any audio recording to be classified. There is no maximum duration as any (longer) audio track can be divided into 100 ms blocks. Each of the blocks would then be independently passed to the feature extraction mechanism and subsequently to the model to obtain the probability vector for each.

7.5. Converting the TensorFlow Model

So far, the learning model has been trained and a user-friendly Android application was created. The final piece of the puzzle is to combine the two and create a fully-functioning application that will be able to classify musical instruments from the palm of a hand. The main idea is to take the current model file (obtained by running the Python training script) and convert it into a format that can be run on the Android platform. This simple conversion is done through a TensorFlow utility method `tf.lite.TFLiteConverter.from_keras_model(model)` that converts a TensorFlow model (`.h5` format) into a TensorFlow Lite model (`.tflite` format).

The `.tflite` file is placed inside the Android assets folder, along with a file containing the output labels. The labels file is a `.txt` file containing the name of each musical instrument in a separate row. The model itself is merely 2.3 megabytes in size making the memory footprint completely insignificant.

8. Results and Observations

This chapter discusses results and some interesting observations made during the development of this thesis. It should be mentioned that some of the predictions were tested with actual physical instruments, since the author owns several different instruments. Acoustic guitar, ukulele, and harmonica were thus tested directly by placing the instruments in front of the phone's microphone.

At first, real-time prediction didn't seem promising but eventually the application was configured to work extremely well. After modifying the MFCC implementation and its parameters on Android, a configuration was finally found that worked exactly like the one during the training process in Python.

8.1. Inadequate Training Data

One interesting fact was observed about the acoustic guitar category in particular. While playing the guitar in front of the microphone, it became clear that the classifier had more success with finger picking than with the strumming style of playing. Former is a classical-like style of playing where each string is plucked individually, such as with arpeggio. Latter is a style of strumming multiple strings at the same time, such as when playing chords. The model therefore recognized finger picking much better than it did strumming chords. Upon closer inspection of the dataset, it turned out that essentially all original files for acoustic guitar were recordings of finger picking style. The model had no means of learning strumming patterns since they sparsely exist in the dataset. Several new strumming recordings were obtained from Freesound and integrated into the existing model. After testing the new model with guitar strumming, a noticable improvement was immediately visible. The model had recognized essentially all types of acoustic guitar playing, including finger picking and strumming.

8.2. Inference Length

The most interesting observation relates to the accuracy of the model. Despite high accuracy results of the model, real-world results are often different from the ones obtained during the training-validation process. Due to the nature of audio data, there is always a set of random variables involved in the process. Microphone quality, static noise, background ambience — these are all unknown factors that can hinder the classification process. Validation accuracy of 97.46% mentioned in the previous chapter is an idealized metric that cannot fully encompass real-world examples. This is why results recorded with a microphone don't necessarily align with those obtained from the training and validation sets. This is precisely the conclusion after running the application on an actual Android device. Lower inference lengths (shorter than 1 second) rarely work as intended, and even though the model was trained on 100-millisecond blocks of audio, it turns out that predictions on such short durations of time aren't precise enough for real-time classification. Inference length of at least 1 second was shown to work with relative success.

8.3. Discarded Instrument Categories

Two additional instrument categories were added and experimented with: human vocals and ambient noise. Human voice is capable of performing wonderful melodic sounds so it made sense to include it as a separate instrument. Several audio recordings were collected through Freesound. They include male and female singers singing in English and Mandarin. Training and validation seemed to work well, scoring high on both training and validation accuracy. However, as soon as testing was performed with an actual mobile device, it became clear that the model had no way of recognizing another voice that was never heard before. Even though the model used a separate validation set, the blocks on which it was trained came from the same recordings that were used in the training set. This means that even though the model never "heard" exactly the same block of audio both during training and validation, it still "heard" the same *voice*. It is assumed that this is the reason why the model was unable to generalize vocals outside of the ones it already "heard".

If the dataset contained dozens or hundreds of different singers, the model would probably be able to generalize and correctly classify new unheard singers. However, this is just an assumption.

9. Conclusion

In this thesis, 12 different instrument categories were collected and used to train the model on MFCC features extracted from instrument recordings. The model was included in an Android application which was then used for performing real-time instrument predictions without internet connection or any other additional resources. Even though the model performed remarkably well on training and validation sets, real-life predictions didn't work as expected, at least not on short-length recordings. Increasing recording (inference) length to one second or longer improved predictions substantially.

The results of this thesis show how a sound classification problem can be solved using machine learning techniques. Even more so, it shows how a well-trained model can be included in an application (or some other software system) to serve as a real-time automatic sound predictor. Such a system would have a number of potential use cases, such as helping people with hearing impairment or classifying animal sounds in the wild. A sound prediction system can even be used in diagnosing and predicting possible fault of physical engines [28].

In addition to its use cases, building such a sound-classification system shows how interesting multifaceted projects can be created by a single author without relying on traditional specialization in a narrow field of research. Data collection, feature extraction, machine learning, software development, testing the application while playing musical instruments — these are all interesting and important segments that can be worked on simultaneously. By showing how theoretical and practical perspectives are able to complement each other, the hope of the author is to inspire others to take similar path forward in their own lives. Human existence on this blue planet called Earth is far too short to pursue a narrow discipline when so many fascinating things exist outside of it.

BIBLIOGRAPHY

[1] Let's Enhance. URL https://letsenhance.io/v2/boost.

[2] Vector Magic. URL https://letsenhance.io/.

[3] Seth Adams. Audio Classification. https://github.com/seth814/Audio-Classification, 2018.

[4] Ali Grami. *Introduction to Digital Communications*. Academic Press, 2015.

[5] Audacity. https://www.audacityteam.org/.

[6] Teach Me Audio. Dynamic Microphone. https://www.teachmeaudio.com/recording/microphones/dynamic-microphone, 2020.

[7] James W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. 1965. URL https://www.ams.org/journals/mcom/1965-19-090/S0025-5718-1965-0178586-1/S0025-5718-1965-0178586-1.pdf.

[8] Cosima. The Analog to Digital Conversion Process, 2014. URL https://worship1.wordpress.com/2014/07/31/the-analog-to-digital-conversion-process/.

[9] Steven Davis and Paul Mermelstein. Comparison of Parametric Representations for Monosyllabic Word Recognition in Continuously Spoken Sentences. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(4):357–366, 1980.

[10] DeepAI. What is Feature Extraction? https://deepai.org/machine-learning-glossary-and-terms/feature-extraction.

[11] Daniel Falbel. RStudio AI Blog: Simple Audio Classification with Keras, 2018. URL https://blogs.rstudio.com/tensorflow/posts/2018-06-06-simple-audio-classification-keras/.

[12] Haytham M. Fayek. Speech Processing for Machine Learning: Filter banks, Mel-Frequency Cepstral Coefficients (MFCCs) and What's In-Between, 2016. URL https://haythamfayek.com/2016/04/21/speech-processing-for-machine-learning.html.

[13] Eduardo Fonseca, Manoj Plakal, Frederic Font, Daniel P. W. Ellis, Xavier Favory, Jordi Pons, and Xavier Serra. General-purpose Tagging of Freesound Audio with AudioSet Labels: Task Description, Dataset, and Baseline. 2018. URL https://arxiv.org/abs/1807.09902.

[14] Freesound. https://freesound.org/.

[15] Dalya Gartzman. Getting to Know the Mel Spectrogram, 2019. URL https://towardsdatascience.com/getting-to-know-the-mel-spectrogram-31bca3e2d9d0.

[16] Sound Recording History. Édouard Léon Scott de Martinville. http://www.soundrecordinghistory.net/inventors-of-sound-recording-devices/edouard-leon-scott/.

[17] IEEE Engineering in Medicine & Biology Society. Highlights in the History of the Fourier Transform. https://web.archive.org/web/20181008214131/https://pulse.embs.org/january-2016/highlights-in-the-history-of-the-fourier-transform/, 2016.

[18] C. Richard Johnson Jr, William A. Sethares, and Andrew G. Klein. *Software Receiver Design: Build your Own Digital Communication System in Five Easy Steps*. 2011.

[19] Kaggle. Freesound General-Purpose Audio Tagging Challenge. https://www.kaggle.com/c/freesound-audio-tagging/, 2018.

[20] Hearing Link. How the Ear Works. https://www.hearinglink.org/your-hearing/about-hearing/how-the-ear-works/, 2018.

[21] Beth Logan. Mel Frequency Cepstral Coefficients for Music Modeling. URL https://pdfs.semanticscholar.org/55af/c2d63fd410a719c3bbe9772d9bbc6bc565a6.pdf.

[22] James Lyons. Mel Frequency Cepstral Coefficient (MFCC) tutorial. http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs/.

[23] Karthikeyan NG, Arun Padmanabhan, and Matt R. Cole. *Mobile Artificial Intelligence Projects*. 2019.

[24] NPR. 1860 'Phonautograph' Is Earliest Known Recording. https://www.npr.org/templates/story/story.php?storyId=89380697&t=1589107003267, 2008.

[25] Brews ohare. Signal envelopes, 2017. URL https://upload.wikimedia.org/wikipedia/commons/3/31/Signal_envelopes.png.

[26] Phonical. FFT Time Frequency View, 2017. URL https://commons.wikimedia.org/wiki/File:FFT-Time-Frequency-View.png.

[27] K Sreenivasa Rao and KE Manjunath. *Speech Recognition Using Articulatory and Excitation Source Features*. Springer, 2017.

[28] Vrijendra Singh and Narendra Meena. Engine fault diagnosis using dtw, mfcc and fft. stranice 83–94, 2009.

[29] Chuanxun Wu. Quantization Error, 2015. URL https://wuchuanxun.github.io/2017/10/23/Quantization_error/.

NOTES

To improve image quality, some images were either vectorized or quality-enhanced. This was done with the help of online services Vector Magic and Let's Enhance [2] [1].

GLOSSARY

Block 100-millisecond segment of original audio. Since sampling rate in this thesis is 16 000 Hz, each block contains 1600 audio samples. 18, 20, 24–28, 31, 36, 37, 39, 41, 47

DFT Discrete Fourier transform, a technique applied on each frame to obtain a periodogram. 7, 19, 20

Periodogram visual representation of a frame's (stationary) frequency spectrum. vi, 7, 19–21

Sample a single discrete point of audio. It is characterized by a particular time value on the x-axis on a particular amplitude value on the y-axis. 4, 5, 11, 13, 14, 20, 21, 36

Spectrogram visual representation of a block's (time-changing) frequency spectrum. vi, 19–22, 24

STFT Short-time Fourier transform, a technique applied on each block to obtain a spectrogram. 20, 21, 24

Track a short piece of audio that can be an audio recording or an audio file stored on the disk. 6, 7, 11–13, 17, 18, 28–31, 36, 37, 39

Classification of Music Based on Machine Learning

Abstract

Audio classification is an interesting machine learning problem that often doesn't get as much attention as some other problems such as computer vision or social media analysis. This thesis presents a system based on machine learning capable of classifying musical instruments. For the model to recognize instruments, a specific set of features is extracted from the original audio files. These features, MFCCs, represent state-of-the-art technique for representing audio in a way that can be used in a machine learning environment. After training, the model is deployed on to an Android application where it can be used for instrument prediction in real time.

Keywords: instrument classification, machine learning, mfcc features

Klasifikacija glazbe temeljena na strojnom učenju

Sažetak

Klasifikacija glazbe zanimljiv je računalni problem koji često ne predstavlja velik interes poput ostalih problema kao što su računalni vid ili analiza društvenih mreža. Ovaj rad predstavlja sustav baziran na strojnom učenju koji može klasificirati glazbene instrumente. Kako bi model to mogao učiniti, karakterističan skup značajki izvučen je iz izvornih zvučnih zapisa. Ove značajke, MFC koeficijenti, predstavljaju suvremenu tehniku za predstavljanje audio zapisa kako bi mogli biti korišteni u kontekstu strojnog učenja. Nakon učenja modela, isti je uključen u Android aplikaciju gdje se može koristiti za klasifikaciju instrumenata u stvarnom vremenu.

Ključne riječi: klasifikacija instrumenata, strojno učenje, mfcc značajke