Harmony Is All You need

Tin Ferković, Dorian Smoljan
University of Zagreb
May 2023

Contents

List of Figures

List of Tables

List of Equations

1 Business Understanding

It is often said that the most important asset of the 21st century is data. Collecting, analyzing, and selling data has become the core business model of many companies, and many others have earned billions in profit by taking advantage of the various insights data can provide. However, when thinking about the most common types of data in our everyday lives, most of us think about data in text or image form, such as social media posts we write, pictures we take, or videos we capture. It is no surprise that the majority of research tends to focus on applications in computer vision or natural language processing. However, one type of data is perhaps often overlooked, even though its presence in our lives is just as important – audio. Ever since the appearance of the first audio recorders in the late 19th century, sound has become one of the most important ways we store and experience information. From music albums and phone calls to podcasts and radio, audio captures the richness and nuance of our world in a unique and powerful manner. Therefore, it makes perfect sense to give audio the focus it deserves in the data science world.

When it comes to extracting data from audio, there are a lot of different audio-related tasks to choose from – such as speech-to-text, sound generation or even uses in medical audio analysis. However, with the rise of streaming services such as Spotify, Deezer, and Apple Music, one area of audio analysis has gained more and more prominence – information extraction from music. Knowing what instruments are playing in any given song, being able to detect the presence of vocals, and understanding the overall structure and dynamics of the music is the crucial first step in many of the algorithms used by leading streaming services to enhance their user experience and increase the amount of time their users spend listening.

This provides an excellent opportunity to develop solutions for instrument and voice detection in audio files, such as the one we present in our work.

1.1 Challenges

Even though today's computers are more powerful than ever before, it is widely known they have trouble with many tasks humans find easy – for example detecting objects in images, extracting information from text, or knowing the difference between a pedestrian and a traffic light in a video. The task of instrument recognition in audio files, however, can prove to be challenging even for humans – although you may easily recognize the sound of a single electric guitar in a rock song solo, or the human singing voice in an opera aria, consistently recognizing all instruments that are playing in a set of songs can prove to be a challenging task for all but the most experienced musicians. Add to that the many complexities of audio files, such as high entropy or noise, and it is no surprise this is a very challenging task for computers to accomplish.

There are a few reasons audio files are generally harder to process and analyze compared to, let's say, image and text files. For a start, audio signals are of temporal nature, with their features changing continuously over time. This can add a high degree of variability to a single audio recording, making its analysis more difficult, especially compared to static images or text of fixed length. Furthermore, audio signals are highly prone to variability and noise, especially when audio is not recorded in studio conditions. Dealing with this noise without reducing the quality of the original audio can be challenging. Finally, audio files lack an explicit or well-defined structure, compared to, for example, text, which often has a well-defined syntactic and semantic structure that makes analysis easier.

Instrument recognition in audio files also introduces its own challenges. One of these challenges lies in the fact that there is rarely a single instrument present in an audio file; most often multiple instruments come together to create a single song. And while this creates a new and wonderful sound for the listener, it creates nothing but a headache for an automatic instrument recognizer, as overlapping instrument sounds interfere with each other and make it harder to isolate and identify specific instruments. Furthermore, there is often a degree of variability between genres; the same instrument played in songs of one genre often sounds somewhat different when playing a song of a different genre. And while for humans discerning this difference usually is not a problem, the added variability is an issue for computers. Finally, there exist hundreds of different instruments, many of them sounding alike. This makes recognizing all possible instruments that could appear in a song virtually impossible, so solutions in this area usually focus on recognizing a small number of instruments.

All of these challenges usually lead to the need for complex solutions and algorithms to tackle this problem. In recent years, deep-learning approaches have stood out as the most promising solution for this problem, and this is the main approach we also choose to take in our work.

1.2 Possible applications

There are many useful applications for an instrument-recognition system. One previously mentioned application is a starting point for various algorithms used by different streaming services. For example, an algorithm that suggests new songs to the users of the streaming service would find the information about instruments present in a song to be useful, as it could use that information to match profiles of users with similar tastes based on the instruments they most often listen to; or it could use the information as a starting point in building a genre-classification algorithm. It could also enhance search; allowing users to search and filter songs by different instruments present in songs.

Of course, there a plenty of other uses for such a system that go beyond streaming services. Such a system could aid music producers and artists when editing their songs, helping them isolate specific parts of the song containing only a certain instrument or voice. It could also help in creating and indexing a library of audio samples, which could then be easily searched by artists wanting to use a specific instrument sample in their new song. Finally, various applications for teaching instruments that already exist, such as Simply Piano[1] or Tonestro[2] could benefit from a robust system capable of detecting the specific instrument(s) these application teach, especially in the presence of multiple different instruments.

1.3 Existing approaches

A number of different methods for instrument detection already exist. These methods could broadly be split into two categories: traditional approaches, and machine-learning oriented approaches.

1.3.1 Traditional approaches

Under traditional approaches, we consider all algorithms which do not rely on machine learning for any part of their predictions. One example of such an approach would be **template matching**, in which a series of filters is created, each corresponding to a pitch of a certain instrument. Then, the input signal is passed through each filter and the correlation between the original and filtered signal

[1] https://www.hellosimply.com/
[2] https://www.tonestro.com/

is calculated. Based on this value, a specific source for the input signal is determined. An example of this approach can be found in [3]. Source separation is another method, in which the original audio signal is decomposed into its constituting signals, ideally one signal per instrument. These split signals can then be classified using a different algorithm.

Even though methods for instrument recognition without using machine learning do exist and have been used in the past, in the last decade, machine learning methods, especially deep learning have long surpassed traditional approaches.

1.3.2 Machine learning approaches

Most of the approaches for instrument recognition nowadays feature some kind of machine learning, usually deep neural networks which are, due to their large capacity especially suited for tasks usually too difficult for computers and traditional algorithms. There are too many different approaches using machine learning to count, but almost all of them follow a similar idea:

1. Extract features from the input audio signal.
2. Use a fully-connected layer (or layers) to make the final decision based on the extracted features

Different methods and architectures could be used to perform either of these steps; for example, a CNN, ViT or even a simple fully connected layer could be used to extract audio features from the input signal, and fully connected layers of various sizes and architectures could be used to make the final prediction. It is impossible to highlight any specific approach, as the sheer amount of papers published on a daily basis makes this impossible. However, some examples include:

1. Linear regression, SVM and XGBoost models trained on Mel-frequency-spectral-coefficients features extracted from raw audio [4]
2. A transformed-based ensemble model trained on additional data created by WaveGAN [5]
3. A CNN trained on spectrogram features [6]

Of course, these approaches highlight only a small fraction of all research, with new papers and approaches published almost on a daily basis.

2 Data Understanding

2.1 Audio representation

In order to understand data, we need to first understand audio. A majority of this theoretical audio overview is done according to [7]. Task of audio classification deals with acoustic waves, e.g. sounds. Sound is transmitted through the medium (usually air) as pressure oscillations. To be completely correct, the term audio is used to refer to transmission, reception, reproduction of sounds that lie within the limits of human hearing. An audio signal is a representation of sound. Unlike sheet music (musical notes and pitches) or symbolic representation (piano-roll, MIDI, score representations), an audio representation encodes all the information, such as temporal, dynamic, and tonal deviations, which make up a musician style. However, in audio representations, these mentioned parameters are not given explicitly, but need to be inferred using audio analysis. This task becomes even more difficult in case of polyphonic music, where multiple voices and/or instruments are present. Polyphonic music is indeed at the center of this work. What's more, the perception of sound is subjective and depends on human ear and brain, which often remains forgotten when performing an ML approach on audio data.

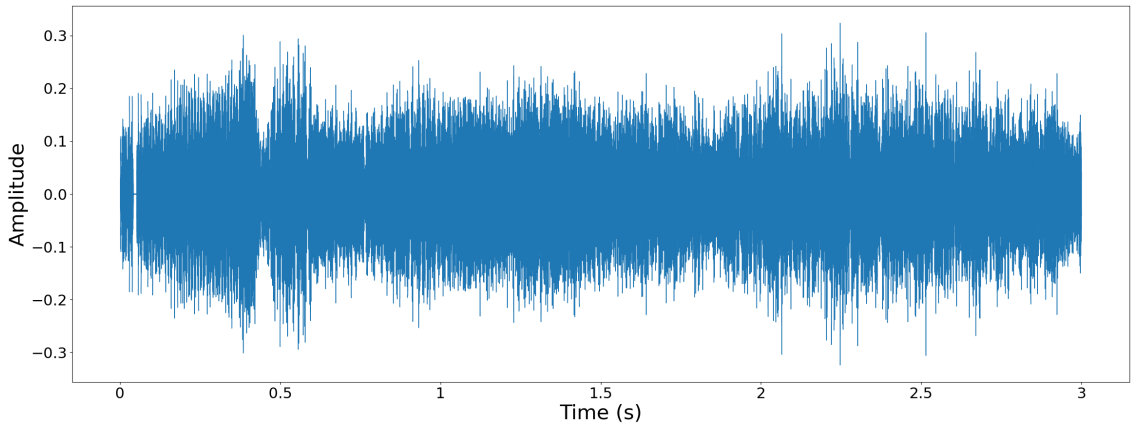


Figure 2.1: Audio waveform – violin.

A sound is produced by a vibrating object, e.g. vocal cords or the string and soundboard of a violin. These vibrations modify the air molecules, resulting in local regions of compression and rarefaction. This alternation pressure travels via air to a listener or microphone, where it is either perceived by a human or converted into an electric signal. Figure 2.1 shows a waveform of a violin from our training set. The oscillations are of the same direction as propagation, which makes sound a longitudinal wave.

If the points of high and low air pressure repeat in a regular alternating fashion, the resulting waveform is periodic. The period is defined as the time required for a cycle. The frequency is the reciprocal of the period and is measured in Hertz (Hz). The audible frequency of a human is between 20 Hz and 20 kHz. These ranges vary for different species. For example, bats can detect frequencies beyond 100 kHz. Pitch is a subjective position of a sound in a complete range of a sound. In music, standard pitches have long been used to facilitate tuning among various performing groups. A4 (440 Hz) is considered a reference pitch.

Sample rate is the number of audio samples carried within a second and is measured in Hz. Namely, an audio with 44,100 samples per second is considered to have a sample rate of 44.1 kHz. Bandwidth is a difference between the highest and the lowest frequency within an audio stream. Both sample rate and frequency are expressed in Hz, which could point into thinking that they denote the same thing. That is, however, not the case. The sample rate determines the maximum audio frequency that can be produced. In theory, maximum frequency that can be represented is half the sample rate. In practice, maximum frequency is a little lower. Thus, if we want an audio sound whose every segment (almost) every human will be able to hear, we should ideally use a sample rate of 44,100 kHz.

2.1.1 Fourier Transform

Fourier transform is a mathematical tool that transforms signals from the time domain to the frequency domain. Fourier transform is based on Fourier's theorem, which states that a continuous and periodic function can be represented as a sum of sine and cosine waves with different frequencies. The transform generates a set of complex values that describe the frequency spectrum of the input signal.

The Fourier transform is a vital tool in signal processing, and several approaches and algorithms have been developed for it. One of the most commonly used algorithms is the fast Fourier transform (FFT), which is a computationally inexpensive variant of the Fourier transform. It is used when the input signal is discrete, which is the case in signal representation as samples and frames. The FFT is a discrete Fourier transform (DFT), as defined in Eq 2.1, where m is ranges from 0 to $K-1$, K is the number of samples in a frame, and x_k represents the k-th input amplitude, i.e. the k-th sample in the frame. Each output X_m is a complex number with a real-valued part $\mathrm{Re}(X_m)$ and an imaginary part $\mathrm{Im}(X_m)$. The magnitude and phase are obtained from these numbers as $|X_m| = \sqrt{Re(X_m)^2 + Im(X_m)^2}$ and $\phi(X_m) = \tan^-1\frac{Im(X_m)}{Re(X_m)}$.

$$X_m = \sum_{k=0}^{K-1} x_k \cdot e^{-\frac{2\cdot\pi\cdot i}{K}\cdot k\cdot m} \tag{2.1}$$

Equation 2.1: Discrete Fourier transform

The Cooley and Tukey algorithm [8], proposed in 1965, is used to reduce the computational complexity of the Fourier transform. It runs in $\mathcal{O}(Klog(K))$ and requires K to be a power of two. The discrete cosine transform (DCT) is another variant of the Fourier transform, which uses only real values instead of complex ones. The short-time Fourier transform (STFT) is a technique used in audio signal processing, which computes several DFTs over different segments of the input signal. This is an especially useful and often used technique in audio processing. The computed segments correspond to windowed frames in temporal order, and the resulting frequency domain representations are concatenated and depicted along the time axis to yield a spectrogram (see Section 2.2.12). The overview of this subsection was done according to [7].

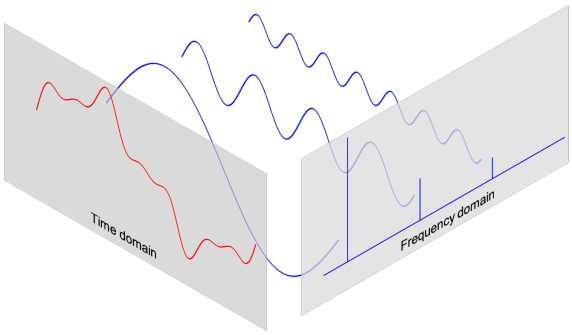


Figure 2.2: Discrete Fourier transform [1]

2.2 Audio Features

In order to perform analysis or ML approaches on the the audio, we first need to extract features from it. Namely, raw audio waveform previously shown in Figure 2.1 is not a good candidate for performing the analysis directly. Thus, we need to create meaningful numbers (audio features) from the marginally meaningless numbers (raw audio waveform). Audio features are numerous and going over all of them is not the point of this work. Instead, we will go over some of which we found interesting and included into at least one of our experiments or analyses.

2.2.1 Zero-Crossing Rate

Zero-Crossing Rate (ZCR) measures the number of time the amplitude value of a raw audio waveform changes its sign within the frame t under consideration, as shown in Eq. 2.2.

$$ZCR_t = \frac{1}{2} \cdot \sum_{k=t \cdot K}^{(t+1) \cdot K - 1} |\text{sgn}(s(k)) - \text{sgn}(s(k+1))| \tag{2.2}$$

Equation 2.2: Zero-crossing Rate

ZCR is a simple low-level time domain feature mostly used in speech recognition and music information retrieval to detect percussive sounds and noise [9]. This feature could be useful in differentiating between classical piano music and distorted percussive instruments, such as electric guitar [7]. An example of ZCR is shown later in Figure 2.15.

2.2.2 Root-Mean-Square-Energy

Root-Mean-Square-Energy (RMSE) is a time domain feature whose computation is shown in Eq. 2.3.

$$RMSE_t = \sqrt{\frac{1}{K} \cdot \sum_{k=t \cdot K}^{(t+1) \cdot K - 1} s(k)^2} \tag{2.3}$$

Equation 2.3: Root-Mean-Square-Energy

RMSE is related to perceived sound intensity and can be used for loudness estimation. Again, classical music tends to have lower RMSE values compared to other genres. An example of RMSE is shown later in Figure 2.15.

2.2.3 Spectral Centroid

Spectral centroid (SC) is a frequency domain feature which represents the frequency where most of the energy is concentrated, i.e. where the center of mass of the spectrum is located. SC is used to measure the *brightness* of a sound. As it can be seen from Eq. 2.4, high frequency bands are given more weight than low frequency ones.

$$SC_t = \frac{\sum_{n=1}^{N} m_t(n) \cdot n}{\sum_{n=1}^{N} m_t(n)} \tag{2.4}$$

Equation 2.4: Spectral Centroid

Thus, SC is sensitive to low-pass filtering, such as down-sampling. If we first calculated SC on an audio sampled at 44.1 kHz, down-sampled the audio to 16 kHz, and calculated the SC again, it would change significantly.

2.2.4 Spectral Bandwidth

Spectral bandwidth (BW) is also a frequency domain feature extracted from the SC (Section 2.2.3). BW indicates a spread of the spectrum around the centroid. A more casual definition could be that BW is a variance from the mean frequency of the signal. Its formulation is given in Eq. 2.5.

$$BW_t = \frac{\sum_{n=1}^{N} |n - SC_t| \cdot m_t(n)}{\sum_{n=1}^{N} m_t(n)} \tag{2.5}$$

Equation 2.5: Spectral Bandwidth

A higher SB value indicates a wider spread of the spectral energy, which can be used to distinguish between different timbres or instruments. For example, classical music tends to show smaller bandwidths than electronic music.

2.2.5 Spectral Contrast

Spectral Contrast is a feature that measures the difference between peaks and valleys in the spectrum. Each frame of a spectrogram is divided into frequency sub-bands. For each sub-band, the energy contrast is estimated by comparing the mean energy in the top quantile (peak energy) to that of the bottom quantile (valley energy). High contrast values generally correspond to clear, narrow-band signals, while low contrast values correspond to broad-band noise [10].

2.2.6 Spectral Rolloff

Spectral Rolloff (SR) is a frequency domain feature that measures the frequency f_t below which a certain percentage of the total spectral energy lies, and can be calculated using Eq. 2.6. It is calculated for each frame t as the center frequency for a spectrogram bin such that at least roll percent (p_r) of the energy of the spectrum in this frame is contained in this bin and the bins below.

$$\arg\min_{f_t \in \{1,\dots,N\}} \sum_{i=1}^{f_t} m_t(i) \geq p_r \cdot \sum_{i=1}^{N} m_t(i) \tag{2.6}$$

Equation 2.6: Spectral Rolloff

SR is useful for characterizing the high-frequency content of an audio signal. For example, classical music tends to have a lower SR compared to rock or electronic music, which has a higher SR due to its reliance on high-frequency energy.

2.2.7 Spectral Flatness

Spectral Flatness (SF) measures the degree of spectral flatness of an audio signal, and can be calculated using Eq. 2.7.

$$SF_t = \frac{\sqrt[N]{\Pi_{n=0}^{N-1} m(n)}}{\frac{\sum_{n=0}^{N-1} m(n)}{N}} \tag{2.7}$$

Equation 2.7: Spectral Flatness

SF is a frequency domain feature that can be used to distinguish between harmonic and noisy sounds. For example, harmonic sounds, such as those produced by string instruments, tend to have a higher SF value compared to noisy sounds, such as those produced by percussive instruments.

2.2.8 Polynomial Features

Polynomial Features (PF) are the coefficients of fitting an n-th order polynomial to the columns of a spectrogram. An example (adapted from [11]) of fitting the 0-th (constant), 1-st (linear), and 2-nd order polynomial on the columns of a spectrogram, along with the spectrogram, can be seen in Figure 2.3. A shown example is from the IRMAS training set and labeled as `vio`.

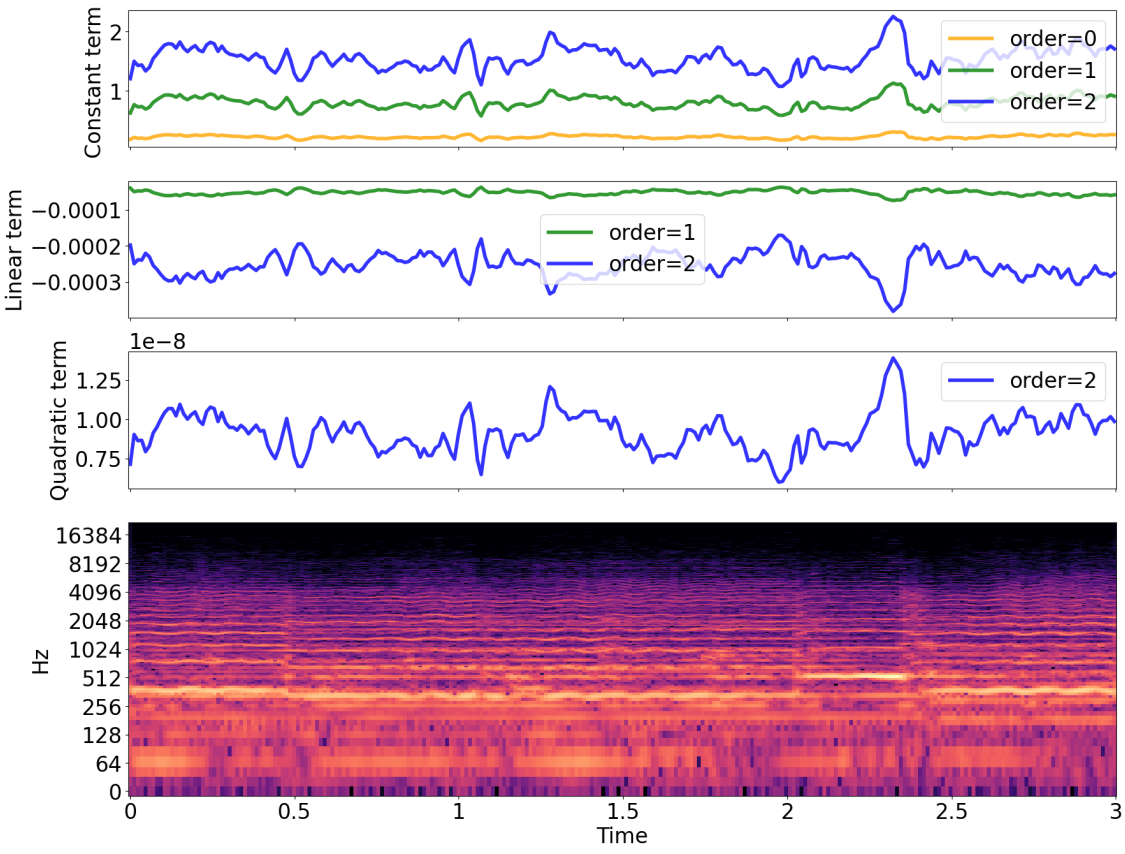


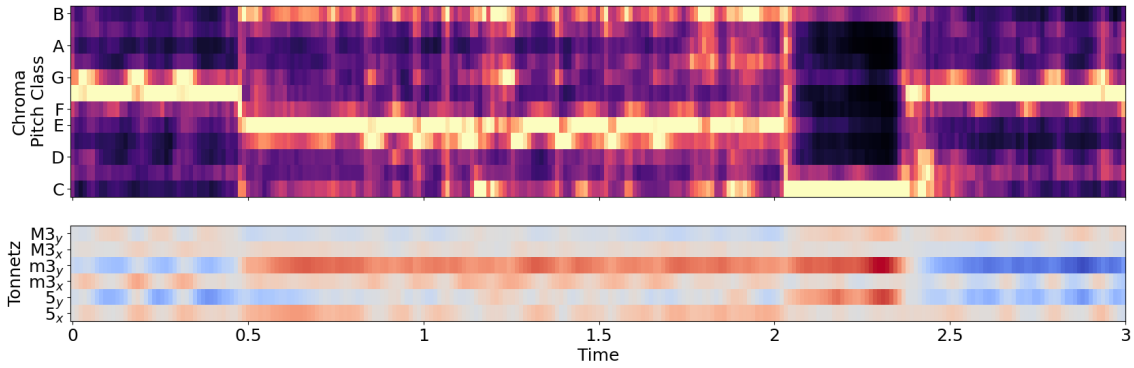
Figure 2.3: 0-th , 1-st, and 2-nd polynomial fit on the spectrogram columns.

2.2.9 Chroma Features

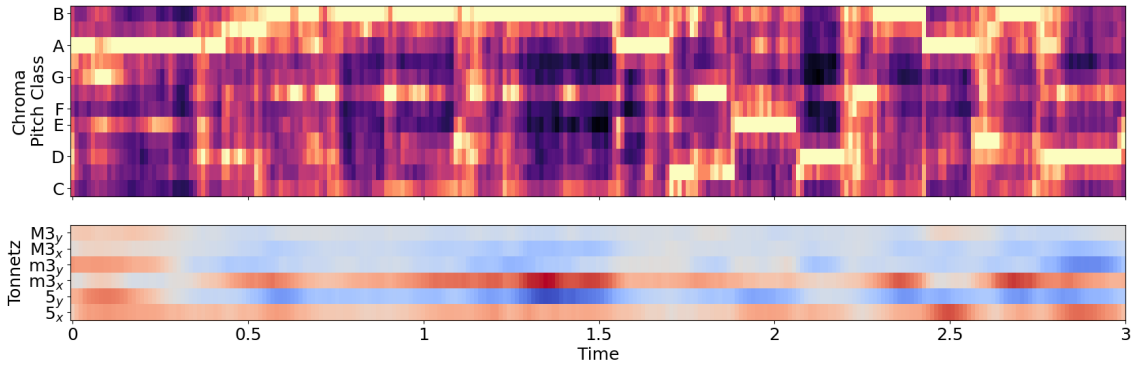
Chroma features are a set of 12 pitch class features that represent the relative intensity of each of the 12 chromatic pitches in a musical piece – C, C#, D, D#, E, F, F#, G, G#, A, A#, and B. Chroma features are invariant to changes in timbre and octave, making them useful for music information retrieval tasks. Chroma is especially powerful in analyzing music whose pitches can be meaningfully categorized. The computation of chroma features involves computing the Short-Time Fourier Transform (STFT) of an audio signal and mapping the resulting spectrum to a 12-dimensional pitch space using a bank of triangular filters. A visualization of chroma features for two examples labeled as `vio` and `gel` is shown in Figure 2.4.

2.2.10 Tonnetz

Tonal centroid features, usually referred to as tonnetz, are a projection of chroma features onto a 6-dimensional basis representing the perfect fifth, minor third, and major third each as two-dimensional coordinates - x and y axis [11]. They capture tonal relationships between the pitches. Regarding any further details about the algorithm, we refer readers to the original paper [12]. A visualization of tonnetz features for two IRMAS training examples labeled as `vio` and `gel` respectively is shown in Figure 2.4. Projection patterns are observable.



(a) Violin



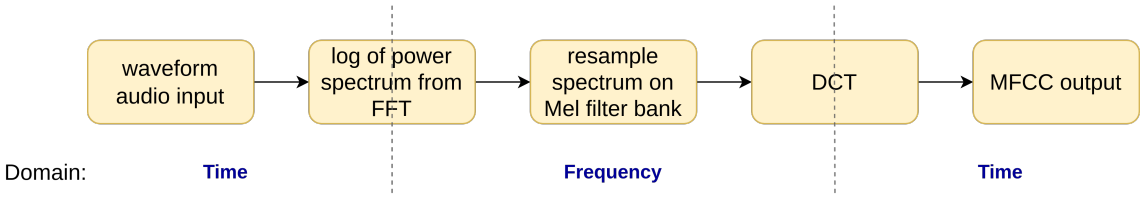
(b) Electric guitar

Figure 2.4: Chroma and tonnetz features of two IRMAS training examples.

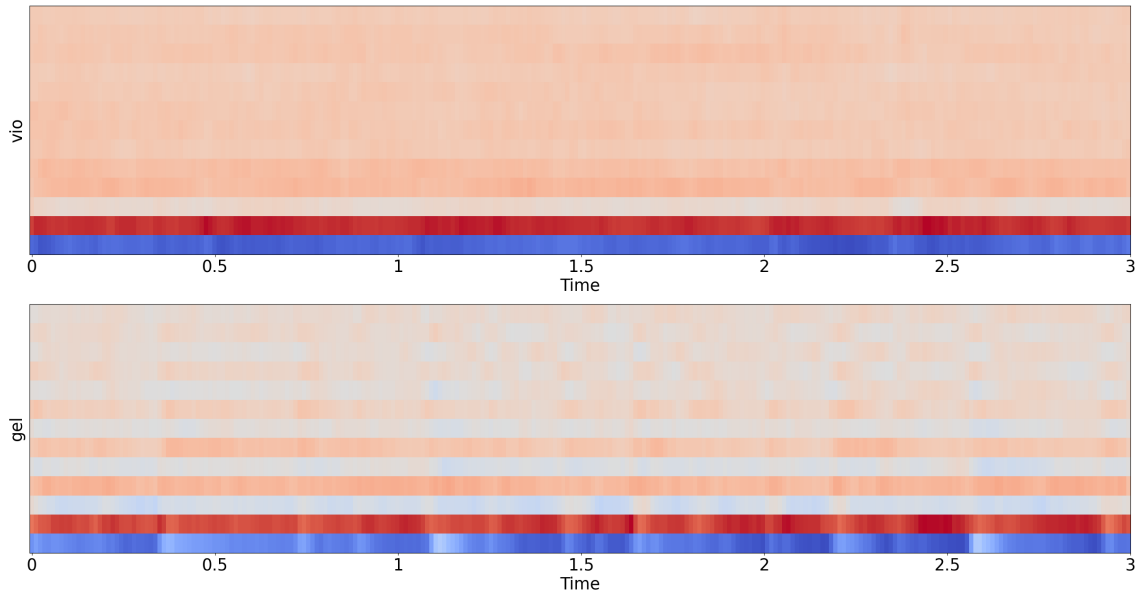
2.2.11 Mel-Frequency Cepstral Coefficients

Mel-Frequency Cepstral Coefficients (MFCC) are a small set of features (usually 10-20) that represent the spectral envelope of an audio signal. A spectral envelope is a curve in the frequency-amplitude plane, derived from a Fourier magnitude spectrum, which describes one point in time (one window, to be precise). MFCCs are commonly used in speech recognition and music information retrieval tasks

such as genre classification, instrument recognition, and emotion recognition etc. The computation of MFCC involves mapping the frequency domain of an audio signal to the mel scale, which is a non-linear frequency scale that approximates the human auditory system. The resulting mel-frequency spectrum is then transformed using the Discrete Cosine Transform (DCT) to obtain a set of cepstral coefficients. A more refined process of obtaining MFCCs is presented in 2.5(a) and the coefficients themselves are shown in Figure 2.5(b) for IRMAS training examples containing `vio` and `gel` respectively. The first few coefficients typically capture the overall spectral shape of the signal, while higher-order coefficients capture finer spectral details. This can be seen in Figure 2.5(b), where the first two coefficient have similar values, while the coefficients above differ significantly.



(a) Process of computing the MFCC from audio waveform



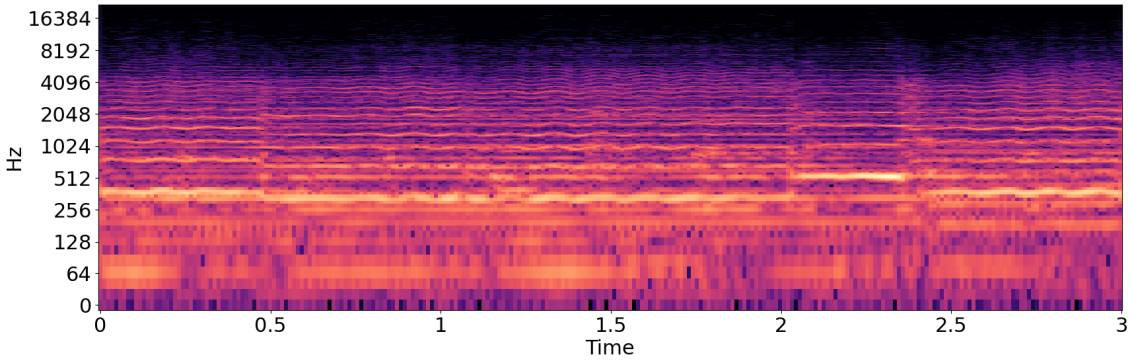
(b) MFCC examples of two different IRMAS training set examples

Figure 2.5: MFCC process and examples

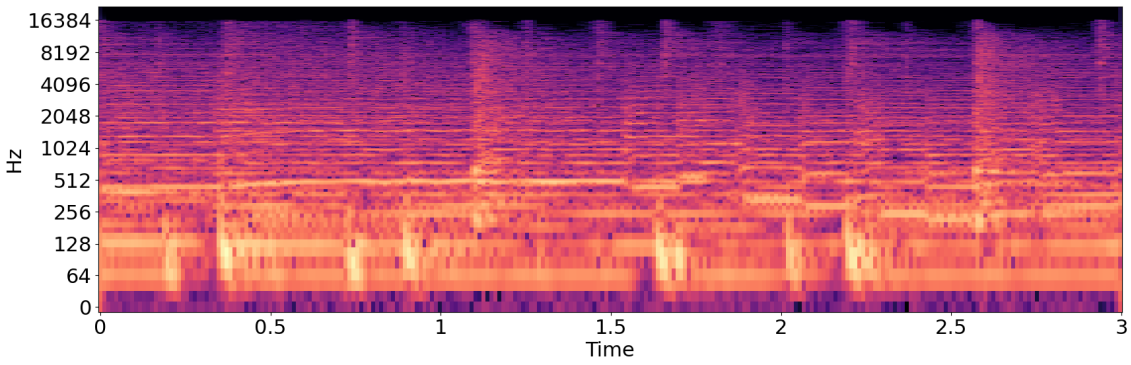
2.2.12 Spectrogram

A spectrogram is a visual representation of the time-varying frequency content of an audio signal. To compute a spectrogram, a time-domain signal is divided into shorter segments of equal length. Then, the fast Fourier transform (FFT) is applied to each segment. The spectrogram is a plot of the spectrum on each segment. The Frame Count parameter determines the number of FFTs used to create the spectrogram and, as a result, the amount of the overall time signal that is split into independent FFTs. Thus, spectrograms are useful for analyzing the spectral content of an audio signal over time. Spectrograms of the same `vio` and `gel` examples are shown in Figure 2.6 (a) and (b) respectively. The log scale is used to better visualize the frequency content of the signal in a way that is more similar to how human perception works. This is due to the fact that our perception of pitch is roughly logarithmic. By using a log-scale, a log-scaled spectrogram compresses the high frequency range while expanding the lower frequency range. This makes the visualization more suitable for signals with a wide frequency range, which is especially useful when using a sample rate of 44.1 kHz, which we do. One can observe smoother frequencies through time in the violin example, as

opposed to visible frequency increases caused by electric guitar strums. A spectrogram representing a violin reveals individual notes and overtones, i.e. the horizontal lines in the spectrogram. On the other hand, spectrogram representing an electric guitar shows a rather vertical characteristic of a spectrogram, where distinctive repeating patterns emerge.



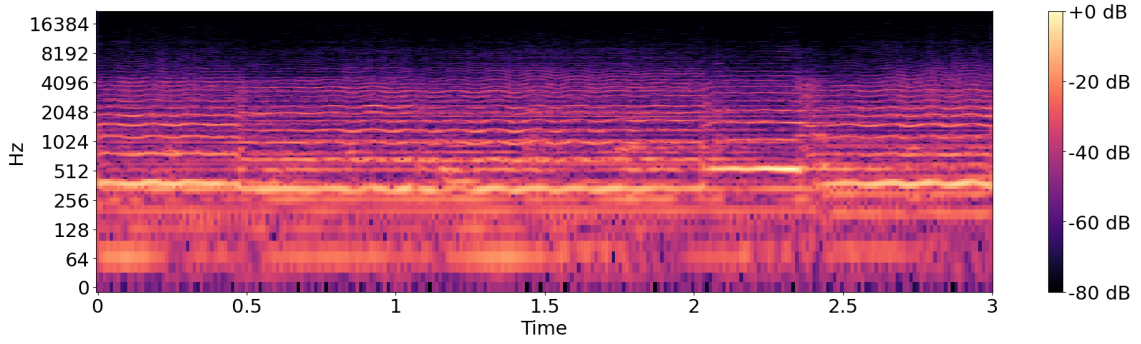
(a) Violin



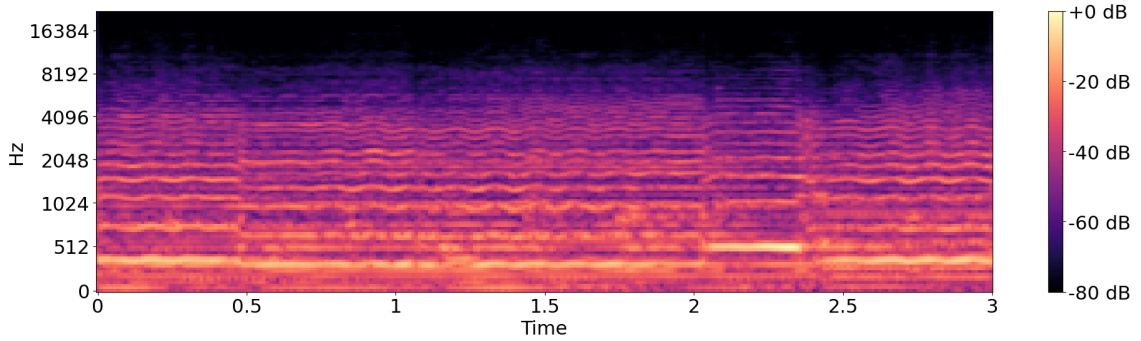
(b) Electric guitar

Figure 2.6: Spectrogram IRMAS training set examples

Another variant of a spectrogram is the mel spectrogram. Mel spectrogram uses a non-linear scaling that is based on the Mel scale, which is a perceptual scale of pitch based on the concept of critical bands. Mel spectrograms map the frequency axis to a set of fixed, non-linearly spaced frequency bands that better reflect how human hearing perceives different frequency regions. This is particularly useful for speech and music signals, as it allows features that are more relevant to human perception to be extracted. The Mel scale is often used in speech and music recognition applications, where it has been found to perform better than linearly-spaced frequency representations. A comparison between a log-scaled and mel-scaled power spectrogram is shown in Figure 2.7. First, we observe a difference in frequency scale (in Hz) on the left. The mel mapping of the frequencies causes them to *start* lower on the scaled compared to a log-scaled spectrogram. As a result, this causes the mel-scaled spectrogram to appear blurred, as the wider range of frequencies is covered on the shorted section of y axis, compared to a log-scaled spectrogram.



(a) Log-scaled



(b) Mel-scaled

Figure 2.7: Comparison between log- and mel-scaled spectrograms

2.3 IRMAS Dataset

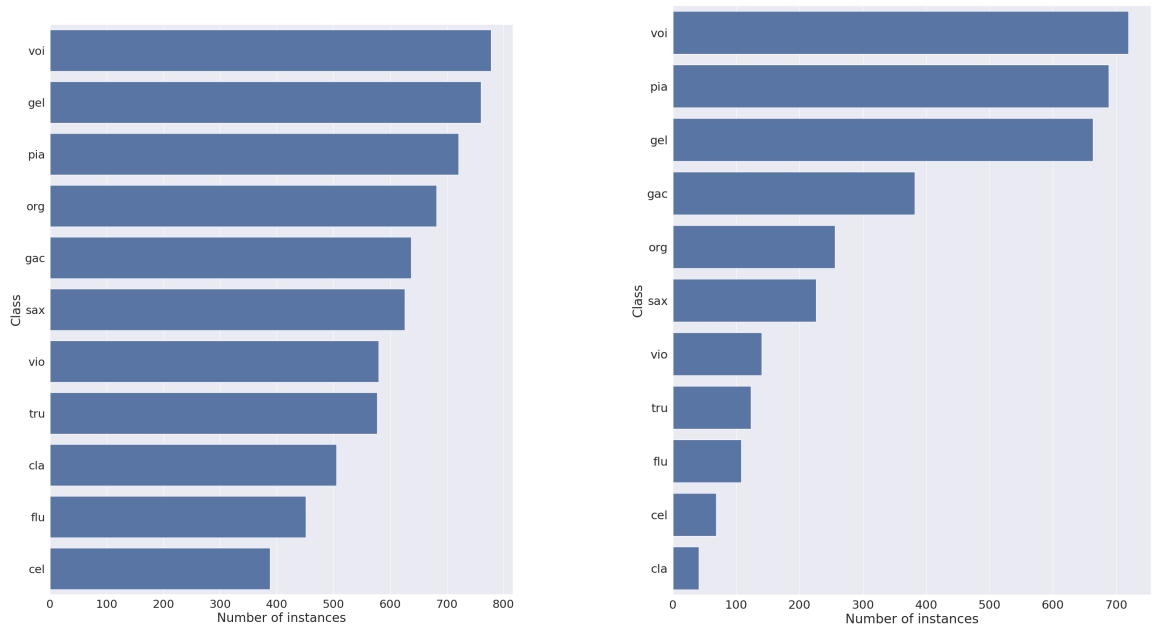
2.3.1 Overview

The dataset we were provided with for the competition is IRMAS [13]. It is a collection of audio files with 11 labels in total – cello, clarinet, flute, acoustic guitar, electric guitar, organ, piano, saxophone, trumpet, violin, and human singing voice. These labels will appear throughout the work with their respective abbreviations – `cel`, `cla`, `flu`, `gac`, `gel`, `org`, `pia`, `sax`, `tru`, `vio`, and `voi`. Training set contained 6705 examples, while the validation set had 2874 examples. Both sets were sampled at the frequency of 44,1 kHz and were saved in a 16-bit stereo wav format. Training examples are 3 seconds long and are labeled with one instrument only. That instrument is guaranteed to appear throughout the whole duration of the audio. There might be other instruments appearing, but they will not be labeled as positive. By inspecting the dataset, we have noticed that the presence of other instruments is not rare, which could degrade the performance of the model. The reason for this performance degradation is the validation set itself, which labels an instrument as positive as soon as it appears within the audio for any duration. Additionally, validation set contains files with the duration between 5 and 20 seconds.

2.3.2 Visualizations

The best way of understanding a dataset is by visualising it in various ways. In order to get the impression of the dataset, we took an approach of visualising its numerous distributions. To start with, Figure 2.8(a) shows the distribution of instances per class. We observe that the dataset is not perfectly balanced, but the number of `voi` examples, which is the most represented class, is roughly 2 times greater than the number of `cel` examples, which is the least represented class. At first, this does not call for any up-/down-sampling, but in case of a significantly worse performance on the

minority classes, we would consider doing so.



(a) Training set **(b)** Test set

Figure 2.8: IRMAS distribution of classes

Due to the fact that the training set is always of the 3 seconds duration and is labeled with a single label, the validation set is more interesting to visualize. Notice how we did not say that the training set is always single-label, as that would be incorrect. Other instruments are present during some segments, but remain unlabeled. As we have randomly split the given validation set into validation and testing sets using the ratio 30:70 (more on this in Chapter 3), we only visualize the test set here. Keep in mind that due to the random split, the distributions would (and do – we've checked!) look the same, but we take the test set as it contains twice as many examples.

To start with, we again visualize the distribution of instances per class in Figure 2.8(b). Although the order of instruments by its number of instances is very similar as in Figure 2.8(a), this distribution is much more skewed. In the test set, the factor difference between the number of instances in most and least represented class is close to 15, which is a big change compared to training set's factor of 2. This shows that our validations and testing might be less trustful in macro or macro average metrics. The reason for this is that evaluating on the test set containing only 40 `cla` examples might yield less trustful `cla` class metrics than they would be if we had evaluated on the set containing, for example, few thousand `cla` examples. Unlike the training set, up-/down-sampling of testing or validating examples cannot be performed. Nevertheless, a balanced training set is, in our opinion, more important, as it will yield a high quality model with the weights properly adjusted. This model can than be evaluated on a high-resource dataset, and if the training set data was of a high quality, it would produce good results. It does not hold the other way around – training a model on a highly imbalanced training set would produce a low quality model which, when evaluated on a high-resource dataset, produces poor macro and macro average results.

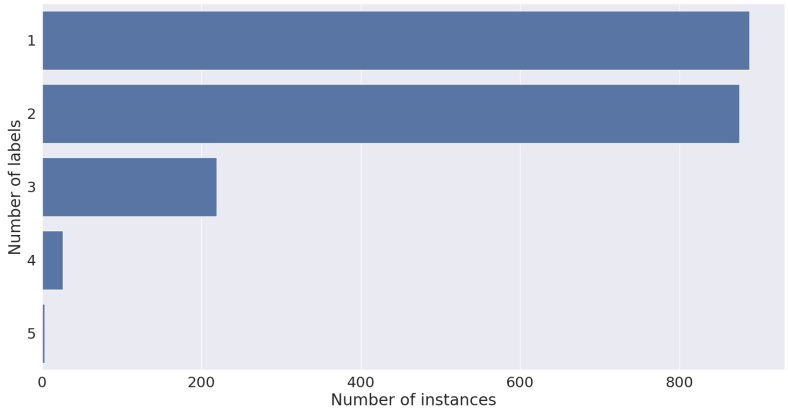


Figure 2.9: IRMAS test set – distribution of the number of labels

Since IRMAS test data is multi-labeled, it is interesting to visualize the distribution of the number of labels. Figure 2.9 shows that a high majority of instruments is labeled with either one or two classes, while only few examples are labeled with four or five classes.

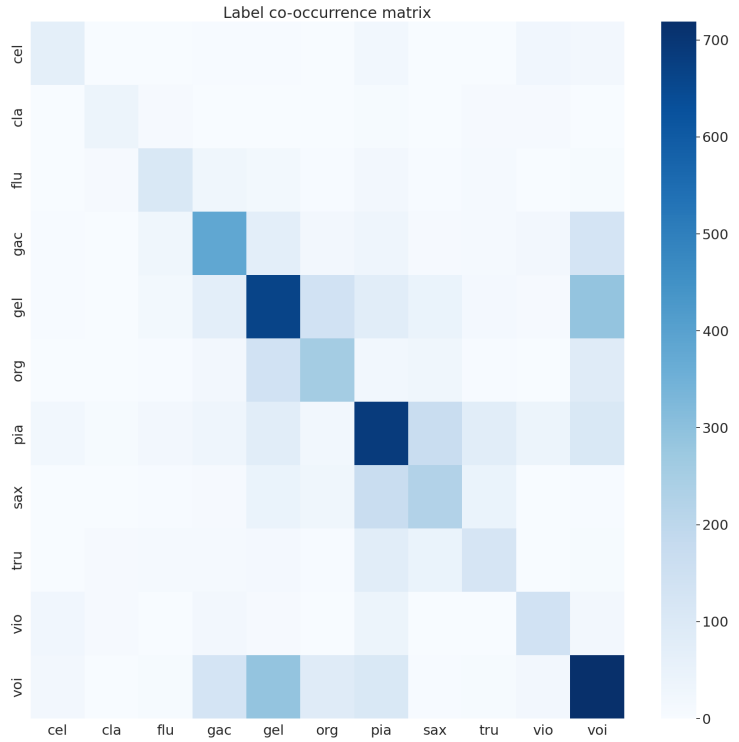


Figure 2.10: IRMAS test set – label co-occurrence matrix

It is also interesting to show which instruments tend to appear together. Figure 2.10 shows a symmetric label co-occurrence matrix on the IRMAS test set. Co-occurrences worth mentioning are `voi-gel`, `pia-sax`, and `org-gel`. However, the graph generally favours the instruments occurring numerous times.

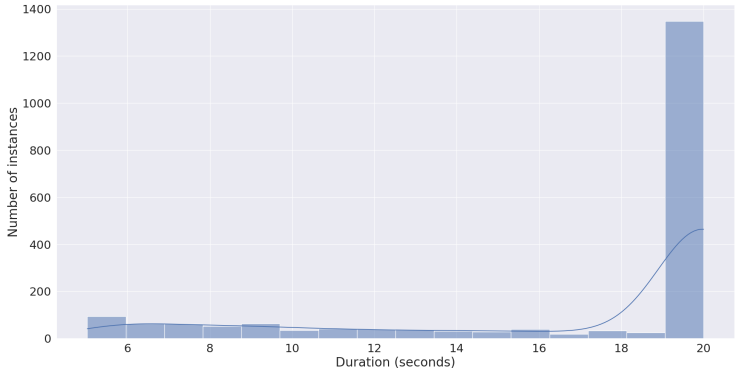
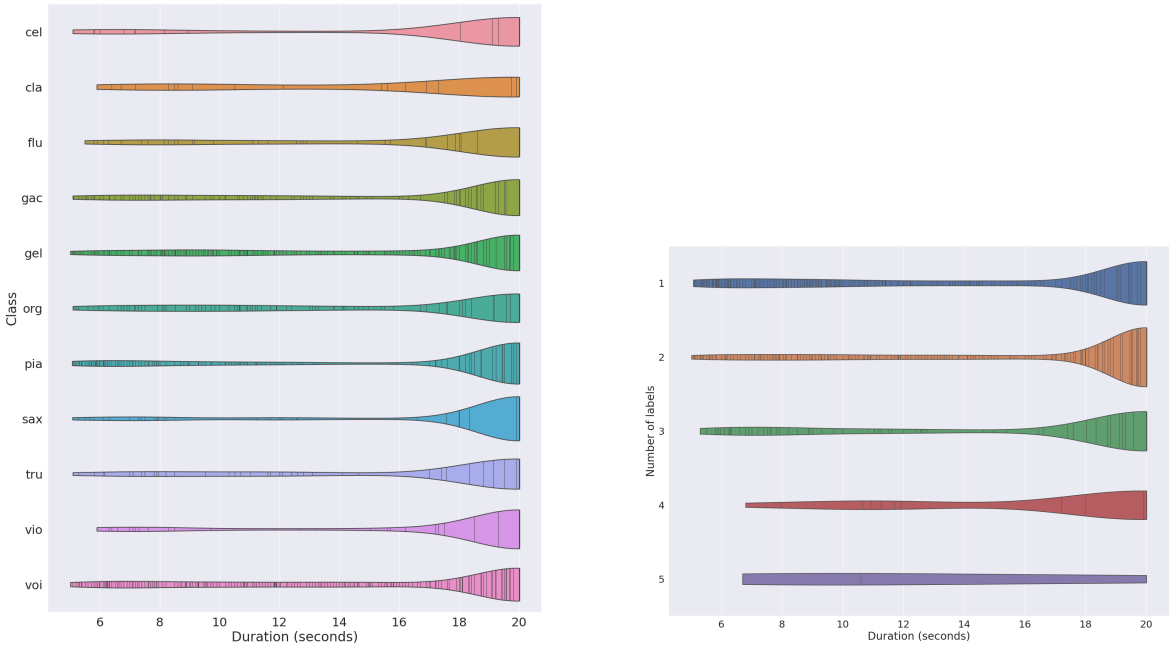


Figure 2.11: IRMAS test set – duration distribution

Since the duration of examples ranges from 5 to 20 seconds in the IRMAS test (validation) set, we visualize the distribution on a histogram in Figure 2.11. A very interesting pattern is observed, showing that a huge majority of evaluation examples is 20 seconds long. At first, this poses a concerning discrepancy between the training and validation/test set. Our way of handling this discrepancy is shown in Sections 4.6 and 4.7. Furthermore, we found it interesting to visualize how does the duration distribution act when compared to individual classes or when compared to the number of labels per example. Figure 2.12(a) shows distribution of the example duration per each dataset class. At first, `cla` appears to be the most uniformly distributed class in terms of example duration. However, we need to take into account that it does not have enough examples for its distribution to be trustful. `voi` and `pia` have *the thickest left tails*, meaning that, among 11 classes, they are the ones with the most percentage of examples of a short duration. Additionally, `voi` is relatively equally distributed, apart from a thick right tail which is present in every class. `sax` is an instrument with the largest percentage of examples of a long duration, e.g. thickest right tail. Figure 2.12(b) shows the distribution of duration per number of labels. The distribution for 5 labels appears uniform, but it is so due to an insufficient number of examples. There are no unusual patterns emerging, apart from the aforementioned distribution of duration, where the majority of examples is 20 seconds long.



(a) Duration per class

(b) Duration per number of labels

Figure 2.12: IRMAS test set – duration per class and number of labels

2.4 Audioset Dataset

2.4.1 Overview

Due to the quality concerns, label imbalance in the validation set, and the ever insufficient number of examples in the IRMAS dataset, we have decided on using the Audioset dataset as well [14]. In Section 3.2 we describe how we have obtained the dataset. In this section we will instead talk about the characteristics of the Audioset. To start with, we've ensured that the same 11 labels present in IRMAS are present here as well. For that reason, in this work, we will refer to *Audioset* as our modified version of the dataset. Audioset is counting $20,885$ examples which, after splitting the dataset using the $0.7/0.1/0.2$ train/val/test split, produces $14,619$ training examples, compared to 6705 in IRMAS (Section 2.3.1). The videos downloaded from YouTube were sampled at 44.1 kHz. However, we cannot know with which sample rate they were originally uploaded, so there was most likely some up-sampling done in the process. Another important difference is that the training samples are now exactly 10 seconds long and possibly contain multiple labels. However, unlike IRMAS, the voice or instrument is not required to appear for the entire duration of the audio. This will later me important for splitting the training or evaluating examples into smaller windows, which is described in Section 4.6.

2.4.2 Visualizations

The best way to understand this new dataset is again by visualizing it. We first plot the distribution of instances per class in Figure 2.13. Compared to IRMAS (Figure 2.8), this dataset is more balanced, especially compared to IRMAS evaluation data.

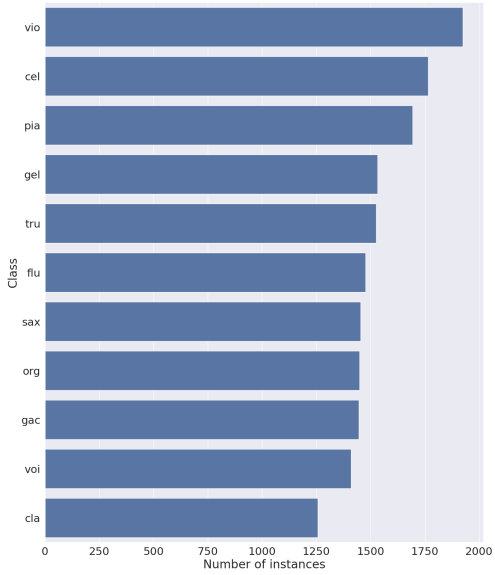
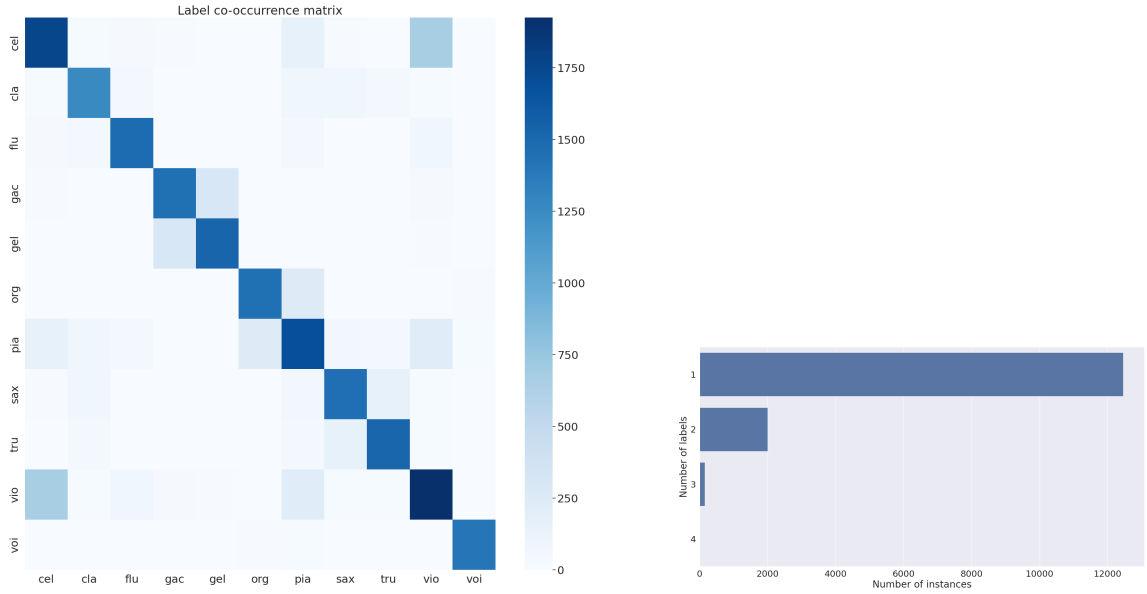


Figure 2.13: Audioset train set – distribution of classes

Since all the files are of the 10 seconds duration, we only visualize the label co-occurrence matrix and the distribution per number of labels in Figure 2.14 (a) and (b) respectively. `cel - vio` is the most often co-occurrence, followed by `gel - gac`, `org - pia`, and `pia - vio`. As for the distribution of the number of labels, Audioset has a large majority of examples labeled with one class only.



(a) Label co-occurrence matrix **(b)** Distribution of the number of labels

Figure 2.14: Audioset training set – label co-occurence matrix and distribution of the number of labels

2.5 Statistical Testing

In order to quantify the quality of previously discussed features (Section 2.2) in the context of the data available to us (Sections 2.3, 2.4), we conduct a one-way ANOVA (ANalysis Of VAriance). It is a technique of testing if all the means of k different populations are equal. In our case, we chose 11 IRMAS training set instruments groups as populations. To be exact, means of all the features described in Section 2.2 for each instrument group. In simplified terms, for each instrument within a group, e.g. `cel`, we calculate a desired feature, e.g. spectral centroid (Section 2.2.3) and calculate its mean. Thus, for each audio example we obtain a single number in a case of a one-dimensional feature, such as spectral centroid. In case of a multi-dimensional feature, e.g. MFCC (Section 2.2.11), we calculate mean for each dimension and obtain the number of features equal to the number of dimensions. Now, in case of an n-dimensional feature, we get an n-dimension vector of means per each instrument group. We are interested if means across each dimension vary for any two instruments within the group.

In the statistical language, this translates to the following hypotheses:

$$H_0 : \mu_1 = \mu_2 = \cdots = \mu_k,$$
$$H_1 : \text{At least two of the means are not equal.}$$

It is assumed that the k populations are independent and normally distributed with means μ_1, μ_2, ..., μ_k, and common variance σ^2. To start with, a common variance σ^2. might not always be justified. Previously mentioned features, such as Zero-Crossing Rate (Section 2.2.1) and Root-Mean-Square-Energy (Section 2.2.2) may vary in a greater amount for electronic music, compared to classical music. A comparison of a classical (`vio`) and electronic (`gel`) Zero-Crossing Rate and Root-Mean-Squared-Energy is shown in Figure 2.15. One can observe a higher variance in the Zero-Crossing Rate in the `gel` example. The difference is even more exaggerated in case of the Root-Mean-Squared-Energy. However, the lack of the justification of the common variance assumption does not necessarily call for the immediate stop of the procedure. Instead, the obtained p-values will be less precise and trustful. With that in mind, we continue the procedure and opt for a very low p-value for making any conclusions. A common mistake in statistical testing is choosing the p-value after already conducting the experiment and obtaining the p-values. Instead, we here

define the p-value to be 0.001. As a reference, a common p-value is, for example, 0.05. Due to the mentioned questionable existence of the common variance, we opt for a much lower number in order to avoid any type 1 errors of rejecting the H_0 when it's actually true. When it comes to the assumption of independent and normally distributed populations, that assumption is not infringed. For more technical details of the one-way ANOVA, we refer readers to [15].

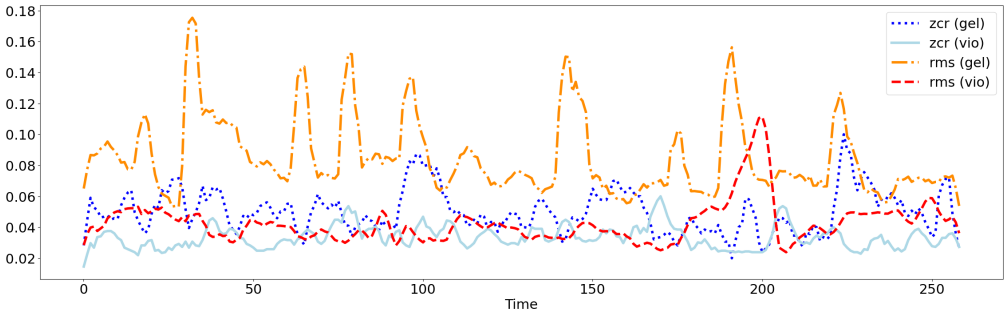


Figure 2.15: Zero-Crossing Rate and Root-Mean-Squared-Energy of the `vio` and `gel` audio examples.

After conducting the one-way ANOVA, we've obtained *extremely* significant results for every feature from Section 2.2, apart from tonnetz (Section 2.2.10). By *extremely*, we refer to the values of around 1e-150, some even smaller. In fact, some p-values were even smaller than 1e-300 and Python described them numerically as Zeros. For tonnetz, $\texttt{m3}_x$, $\texttt{m3}_y$, $\texttt{M3}_x$, and $\texttt{M3}_y$ were not significant or were very close to not being significant, which is also worth mentioning with the sizes of other p-values considered. Such small p-values are, of course, due to the difference in means, but also due to the sample size. Namely, one-way ANOVA uses the sample size in calculating the F statistics, which is used to calculate the p-value.

It is important to mention that averaging the feature across the time dimension causes a major information loss and would not be considered an actual advanced technique of classifying the instruments. Besides, ignoring the feature interactions and treating them independently is also a simple and ineffective technique. This one-way ANOVA experiment was conducted just to show that the values of the chosen features generally differ for our instrument groups and are a good starting point for the musical information retrieval.

3 Data Preparation

One of the most important parts of any data science pipeline is data preparation. If the acquired data is not prepared in the right way, no matter how good the models are, optimal performance will be unattainable. Therefore, in this section, we are going to present all the steps we took to prepare our data for training.

3.1 Feature selection

Samples in the IRMAS dataset consisted of three features, alongside the raw audio itself - length of the raw audio, the genre the song from the audio belongs to, and a Boolean value indicating the presence or absence of drums. However, due to the fact that these values only appear in the train set, and not in the given evaluation set, and without a way to extract them, we decided not to use features genre and drums, leaving us only with raw audio as the base feature.

However, upon researching previous works in instrument and audio classification, we knew that models working with raw audio probably will probably not achieve the best results. As a result, we extracted a number of different features from the available raw audio and used those features as inputs to our models. Exact definitions of all the listed features have already been explained in the previous section, so this section will only focus on the exact way these features were extracted from raw audio and used in our models. Librosa [3] library was used to extract each of these features.

3.1.1 MFCC

The first features we decided to try using were Mel-frequency cepstral coefficients (MFCC). The choice of using this feature was motivated by a couple of previous works achieving good results using MFCC, such as [4] or [16]. MFCC features were calculated using the default Librosa's get_mfcc method with the following parameters: the length of the Fast-Fourier transform (FFT) window was set to 2048, hop length was 512, and the sampling rate was usually set to the default sampling rate of our audio, 44100 Hz. We choose 40 as the number of MFC coefficients, which is a bit higher than the usual number (13). We chose a higher value as we believed the model we used had enough capacity to be able to fully benefit from the additional information provided by using more coefficients. The output of the method was a 2D feature matrix which was resized to the dimensions 256x40, simply to standardize the dimensions of inputs to our model.

3.1.2 Mel spectrograms

The second type of features extracted from audio we decided on using were mel-spectrograms (further referred to as just spectrograms), as they were by far the most prevalent features used in most previous work we analyzed and seemed to be the most promising ones. They also enabled us to treat the problem of instrument detection as a computer vision problem, instead of an audio one, as a generated spectrogram is nothing more but an image. This allowed us to approach the problem from another angle, as well as to use a larger spectre of models and algorithms. When generating spectrograms, the following values were used: number of mel bands was set to 256, yet again in

[3] https://librosa.org/

hopes that a higher value will allow our models to extract more information from the input audio. FFT window was set to 2048 and hop length to 512 (25% of the length of the FFT window). The sampling rate was set to the default sampling rate of input audio, 44100 Hz. The resulting image was resized to 256x256 pixels to standardize the dimensions of spectrograms entering the model. As the resulting spectrogram was a single-channel image, the values of the channel were repeated 3 times in order to acquire a 3-channel image which our model expected.

3.1.3 General audio features

Finally, we decided to try and combine multiple audio features into a single matrix which would then be provided as input to our models. We assumed that by combining multiple features our model would be able to extract more information about the input audio and thus work better than with only one of the features. The features we selected for combining were, as follows:

- MFCC
- Chroma features
- Zero-crossing rate
- Spectral centroid
- Root-Mean-Square-Energy
- Spectral contrast
- Spectral bandwith
- Spectral roll-off
- Spectral flatness
- Polynomial features

Tonnetz features were also the ones we initially considered, but finally decided not to use, as our ANOVA in Section 2.5 showed that its mean values do not significantly differ across different (instrument) groups. Each feature was created using the default values from the corresponding Librosa method. This meant that in this instance we used 13 for the number of Mel coefficients when generating MFCC, as it is the default value. In this case, we felt that we already increased the amount of information by a large margin by incorporating all the other features, so there was no need for the extra information gained by using 40 as the number of Mel coefficients when generating MFCC. Every feature was generated as a matrix of dimensions $M_{n_i} \times N$, where n_i represents the number of rows for the i-th feature, and N depends on the length of the input audio. All of the features were then combined into a single matrix of dimensions $40 \times N$, whose rows (40) were then passed into the model as separate channels. This approach of combining multiple features into a single vector is similar to the one used in [4], although we use more features in our approach and take the entire MFCC feature matrix as input, whereas authors from [4] calculate the mean of each Mel coefficient column to create the final input, resulting in a 1×13 vector, compared to our $259 \times N$ matrix.

3.2 Collecting new data – Audioset

As already described in section 4, due to quality concerns with the IRMAS dataset, and in an attempt to acquire more data, we downloaded a subset of the Audioset dataset containing only examples from target classes. We've used Google's original files `class_labels_indices.csv`, `balanced_train_segments.csv`, and `unbalanced_train_segments.csv`[4]. The file

[4] https://research.google.com/audioset/index.html

`balanced_train_segments.csv` contains mappings from their string unique IDs to actually recognizable classes, such as the mapping from `/m/01xqw` to "Cello". The files `balanced_train_segments.csv` and `unbalanced_train_segments.csv` both contain columns YTID, start_seconds, end_seconds, and positive_labels. The only difference is that, as the names suggest, the labels are balanced in one file, and unbalanced in the other. A small example of actual values from `balanced_train_segments.csv` is shown in Table 3.1. YTID is the unique YouTube identifier that needs to be appended to `"https://www.youtube.com/watch?v="`. Columns start_seconds and end_seconds suggest the start and end time of a 10 second window. These columns were used to crop the audio after downloading it. We downloaded just the audio from the video. Unfortunately, we did not find a library with the option of downloading a YouTube video (audio) with the given time range. Instead, after downloading the video using the pytube[5] library, ffmpeg[6] library for Python was used to crop the audio and save it to the .mp3 format. This saving to the .mp3 format turned out to be a mistake, as librosa, which uses libsndfile[7] in the background, does not support .mp3 files, and switches to using audioread[8] instead, which takes much longer. Thus, all the file were converted to the vaw format first using a single channel, 44.1 kHz sample rate, and `pcm_s16le` audio codec.

YTID	start_seconds	end_seconds	positive_labels
−−PJHxphWEs	30.000	40.000	`"/m/09x0r,/t/dd00088"`
−−ZhevVpy1s	50.000	60.000	`"/m/012xff"`
−−aE2O5G5WE	0.000	10.000	`"/m/03fwl,/m/04rlf,/m/09x0r"`
−−aO5cdqSAg	30.000	40.000	`"/t/dd00003,/t/dd00005"`

Table 3.1: A small sample from the `balanced_train_segments.csv` file

When downloading the dataset, we occurred numerous VideoNotAvailable warnings, due to the videos being made private or removed. Surprisingly, we did not occur any blacklisting from YouTube due to a large number of GET request in a short period of time. We did, however, notice problems from our service providers. Namely, as soon as we would lose the connection to the internet (for whatever reason), pytube would throw a large number of URLError class exceptions. This is because it would continue looping over the YTIDs, trying to fetch them, but would fail so (because of the loss of connection) and would interpret it as the URLError. This proved to be a problem even for short-timed disconnections, such as few seconds, because a computer can loop over many examples in a second. At first, we thought that this is the short-time blacklisting from YouTube side. However, it was not. Finally, we solved this problem by setting up a try-except block and sleeping for 60 seconds whenever catching a URLError class exception, which happened rarely, but caused skipping a large number of YTIDs.

[5] https://pytube.io/en/latest/
[6] https://github.com/kkroening/ffmpeg-python
[7] https://github.com/libsndfile/libsndfile
[8] https://github.com/beetbox/audioread

Audioset ID	Description	Class
"/m/02qldy"	narration	`voi`
"/m/01h8n0"	conversation	`voi`
"/m/02zsn"	female speech	`voi`
"/m/05zppz"	male speech	`voi`
"/t/dd00005"	child singing	`voi`
"/t/dd00004"	female singing	`voi`
"/t/dd00003"	male singing	`voi`
"/m/0l14jd"	choir	`voi`
"/m/07y_7" w/o "/m/0d8_n"	violin, fiddle w/o pizzicato	`vio`
"/m/07gql"	trumpet	`tru`
"/m/06ncr"	saxophone	`sax`
"/m/05r5c" w/o "/m/01s0ps"	piano w/o electric piano	`pia`
"/m/013y1f"	organ (hammond and electronic)	`org`
"/m/02sgy"	electric guitar	`gel`
"/m/042v_gx"	acoustic guitar	`gac`
"/m/0l14j_"	flute	`flu`
"/m/01wy6"	clarinet	`cla`
"/m/01xqw"	cello	`cel`

Table 3.2: Audioset IDs of interest

Before downloading the YouTube audio, we first checked if any of the positive labels we were interested in was contained within the positive_labels column. In Table 3.2 we show all the IDs we were interested in, along with some we were trying to avoid (pizzicato and electric piano). Some categories, such as "guitar", were general, and contained sub-categories, such as electric, bass, acoustic, steel guitar etc. That called for additional checks so that no unwanted categories get included. Checkpointing to a csv file was done every few thousand iterations, so that no progress is lost in case of exceptions. Although there were sometimes multiple `voi` class sources possible, we made sure to label `voi` as positive only once.

Finally, although all the files were said to have a duration of 10 seconds, we found that not to be the case. Consequently, we removed the files which were either shorter or longer than 10 seconds by more than 0.5 seconds. For all the other files, we padded them with zeros or truncated them. This removed around 0.1 % of the files. A bit larger number of files were padded, but the padding was usually much lower than 0.5 seconds. The new mean and standard deviation values were calculated for performing normalization on Audioset.

3.3 Normalization

Normalizing input data is a common technique in deep learning, usually done to help with the convergence of the gradient descent algorithm. There are a couple of different ways normalization could be done – we decided on normalizing audio input to have a mean of 0 and standard deviation of 1. In order to do so, we calculated the mean and standard deviation of audio files (loaded as Numpy arrays) on the train set, and then used those values when normalizing the input values according to the formula:

$$\text{New value} = \frac{\text{value} - \text{mean}}{\text{standard deviation}}$$

Only audio files were normalized, meaning that spectrograms or MFCC features generated from those audio files were not additionally normalized, with the exception of spectrograms passed to the Audio Spectrogram Transformer.

3.4 Augmentations

Due to the relatively small number of examples in the IRMAS dataset, and in an attempt to improve the generalization capabilities of our models, we decided to implement multiple augmentation strategies. The augmentation strategies can be split into two groups: augmentations done on the raw audio, and augmentations done on spectrograms. In all cases, all possible augmentations for the given features were used, meaning that, for example, if we were using spectrograms as inputs to our models, we would first augment the original audio files using audio augmentations, then additionally augment the spectrograms generated from those audio files. On the other hand, if we were using raw audio or general audio features, then only augmentations on raw audio would be used.

3.4.1 Audio augmentations

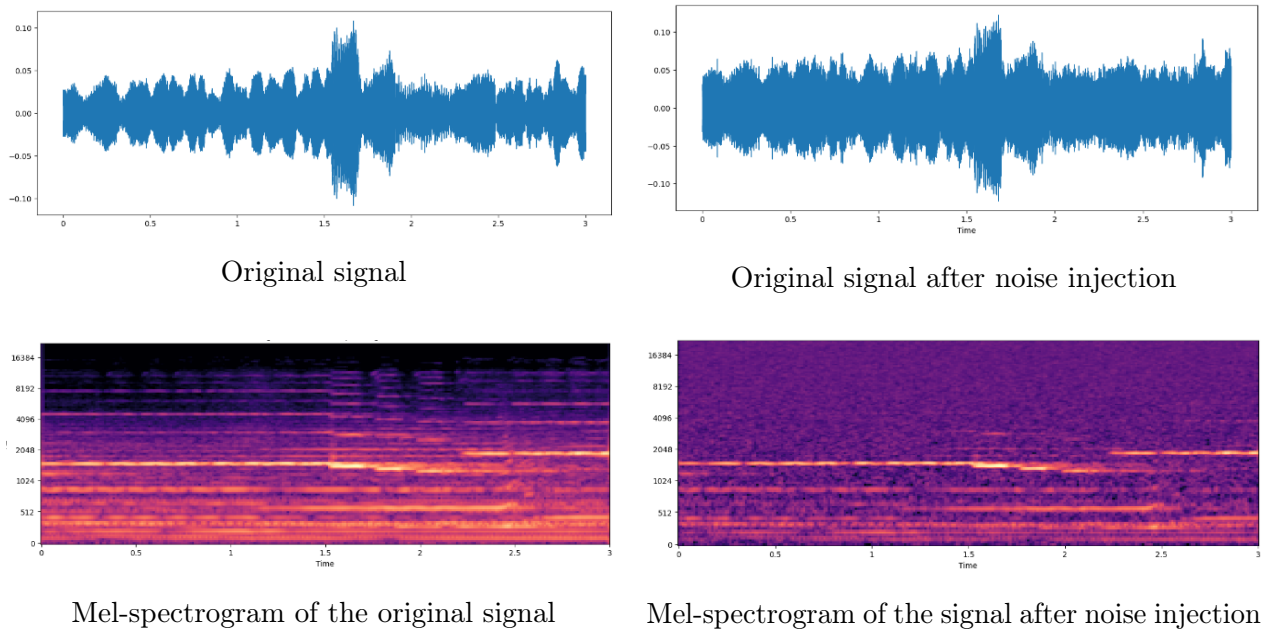
Noise injection

This is a common type of audio augmentation, often used in different types of audio-related machine learning. It consists of generating a noise sample from a distribution (i.e. Gaussian distribution), multiplying it by a small number, and adding it to the original audio. Simply, we can write:

$$\text{Augmented audio} = \text{Original audio} + \alpha \times \text{Noise}$$

Where α represents a small number used to make sure the added noise does not overpower the original signal.

In our experiments, we used noise generated from a Gaussian distribution **with the mean and standard deviation equal to the mean and standard deviation of the training dataset**. An example of applying the augmentation can be seen in Figure 3.1



Original signal | Original signal after noise injection

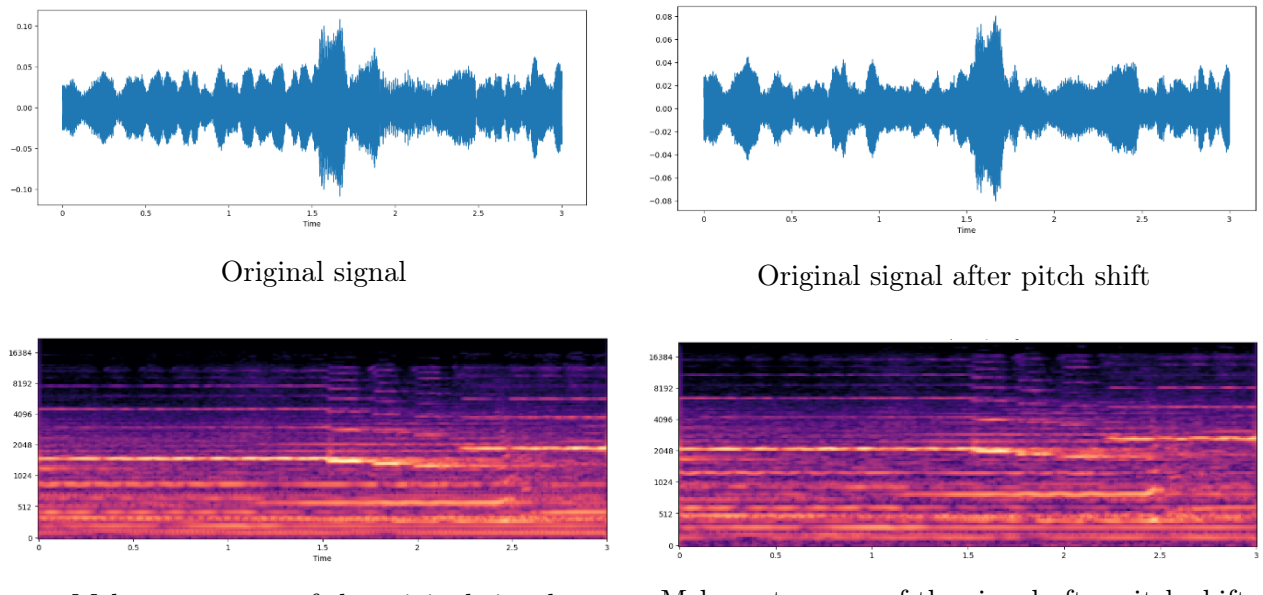
Mel-spectrogram of the original signal | Mel-spectrogram of the signal after noise injection

Figure 3.1: Example of the effect of noise injection, alpha=0.1

Although the signal intensity plot does not show much difference between the original and augmented signal, the difference is obvious when looking at the mel-spectrogram representations of the signal. It also clarifies that using a too-large alpha value can significantly degrade the original signal, and depending on the representation used, potentially harm training. That is why we decided to choose a more conservative alpha value of 0.005 in our experiments.

Pitch shift

The second type of audio augmentation we used is pitch shift. This type of audio augmentation consists of shifting the pitch of the input signal by a random amount. In our experiments, we shifted the signal by a random number of semitones generated uniformly in range [-6,6]. It is important not to shift the pitch of the original signal too much, as it could end up outside the normal pitch range for that instrument. The effects of pitch shift augmentation can be seen in Figure 3.2.



Original signal | Original signal after pitch shift

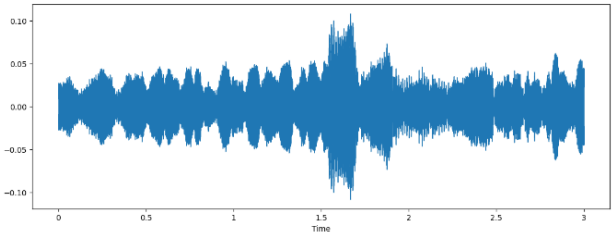
Mel-spectrogram of the original signal | Mel-spectrogram of the signal after pitch shift

Figure 3.2: Example of the effect of pitch shift up by 6 semitones

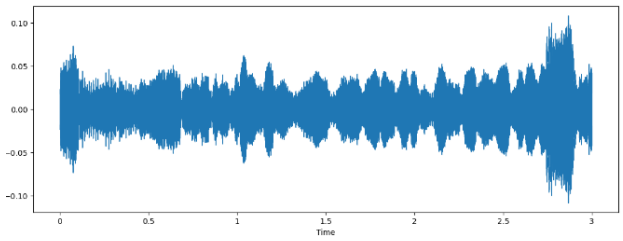
We can easily see the effects of the pitch shift in spectrograms in Figure 3.2. We can see the yellow lines representing the presence of certain frequencies clearly shift up in the augmented spectrogram, compared to the original one.

Time shift

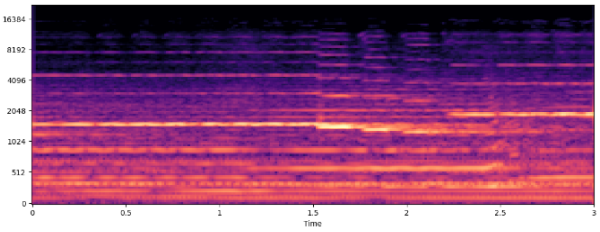
Finally, the third audio augmentation used is time shift. It consists of moving the entire signal by a certain amount forward or backward, so it starts and ends on a different part of the signal. The duration of the signal is not changed. Let's say we time shift the original signal by 1 second forward. That would mean that the new signal now starts at the previous signal's one-second mark, and its last second is what used to be the first second of the original signal. The effect of this augmentation is best seen through visualizations in Figure 3.3.



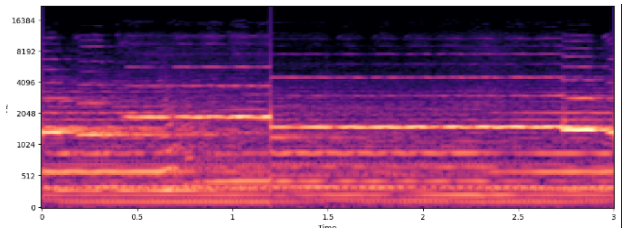
Original signal



Original signal after time shift



Mel-spectrogram of the original signal



Mel-spectrogram of the signal after time shift

Figure 3.3: Example of the effect of time shift up 40% of original length forwards

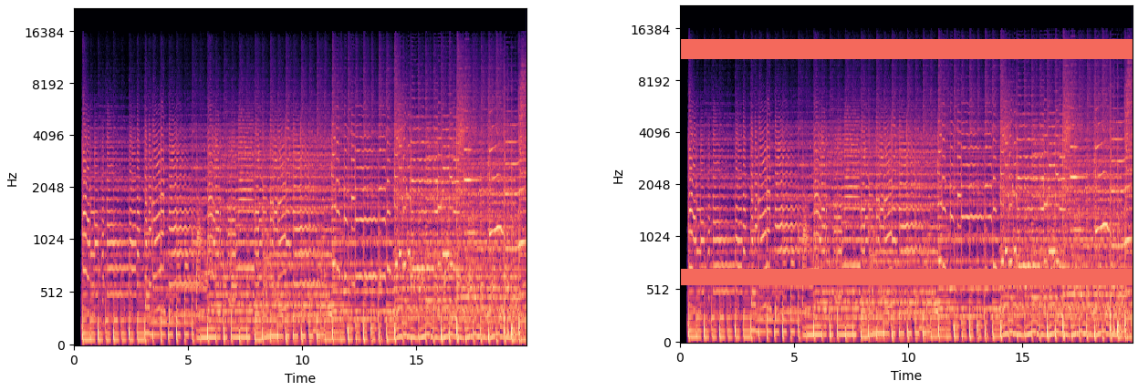
In our experiments, we performed time shifting by first randomly choosing a percentage in range [-0.4, 0.4], and then performing time shifting by shifting the original signal by the percentage of its original length forwards (if the chosen number was positive), or backward (if the chosen number was negative).

3.4.2 Spectrogram augmentations

The two spectrogram augmentations used in our experiments, time and frequency masking, were first introduced in [17]. The authors justify these augmentations by saying "These features (mel-spectrograms) should be robust to deformations in the time direction, partial loss of frequency information, and partial loss of small segments of speech." The essential idea of both techniques is the same - mask a part of the input spectrogram, forcing the model not to rely too much on any specific feature or part of the spectrogram, instead encouraging it to learn from more features, thus increasing its generalization.

Frequency masking

The first of these augmentations is frequency masking. As its name suggests, it consists of masking a block of consecutive mel frequency channels in the original mel spectrogram. The mask itself simply consists of the mean values of the entire spectrogram; this is important so as to not introduce any "foreign" information into the spectrogram. More formally, frequency masking is applied so that f consecutive mel frequency channels $[f_0, f_0 + f)$ are masked, f being chosen uniformly in range [0, F], F being the frequency mask parameter, while f_0 is chosen in range $[0, v - f)$, where v is the number of mel frequency channels. The effects of frequency masking can be seen in Figure 3.4



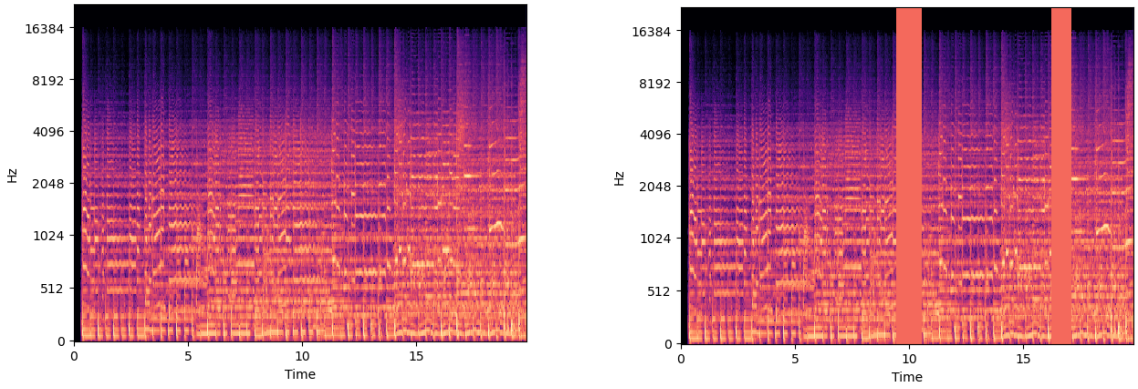
(a) Original spectrogram **(b)** Spectrogram after frequency masking

Figure 3.4: Example of the effect of frequency masking augmentation using two frequency masks

During our experiments, we consistently created two frequency masks, with each mask occupying a portion (P) of the total frequencies. The value of P was uniformly selected from [0,10]. As two masks were used, the percentage of total frequencies masked in any given spectrogram could be at most 20% and on average 10%.

Time masking

The second type of spectrogram augmentation is time masking. It is, in essence, equivalent to frequency masking, except that it is done in the time domain of the spectrogram. Formally, time masking is applied so that t consecutive time steps $[t_0, t_0 + t)$ are masked, where t is uniformly chosen in range $[0, T]$, T being the time mask parameter, and t_0 chosen uniformly from $[0, \tau - t)$. τ is the number of time steps in the mel spectrogram. The effects of time masking can be seen in Figure 3.5.



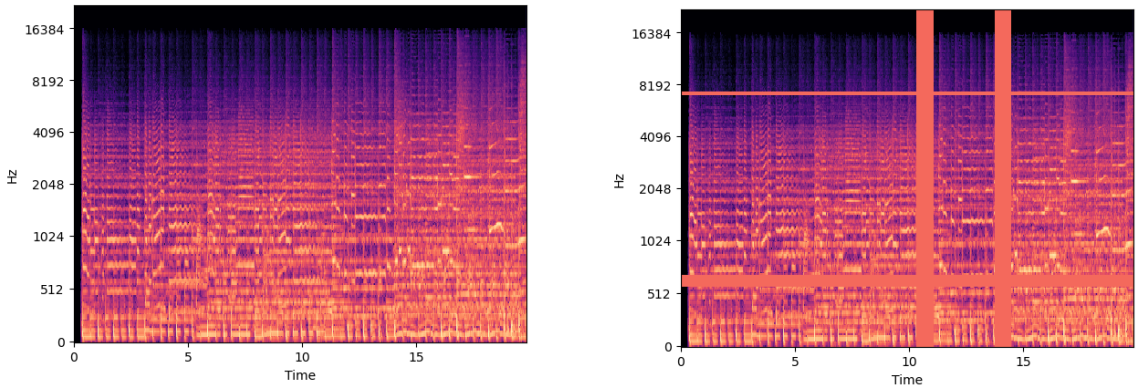
(a) Original spectrogram **(b)** Spectrogram after time masking

Figure 3.5: Example of the effect of time masking augmentation using two time masks

Same as in frequency masking, we always generated two time masks, each occupying P of the total time steps, P being chosen uniformly in range [0,10].

Time + frequency masking

Finally, we can see the combined effects of time and frequency masking in Figure 3.6.



(a) Original spectrogram **(b)** Spectrogram after time and frequency masking

Figure 3.6: Example of the combined effect of time and frequency masking augmentation using two time and frequency masks

3.5 Dynamic sampling

The specific nature of the IRMAS dataset is, as has already been stated, that all the examples from the train set contain only a single label, while the examples in the test set may contain multiple labels per example. To mitigate this problem, instead of combining multiple train samples once and creating a single new set of training data with multiple labels, we decided to implement a different approach we named **dynamic sampling**. In short, new training examples would be generated dynamically during training by randomly selecting and combining multiple examples with a single label into a new example containing multiple labels. In theory, this should greatly increase the variability of available data, as well as solve the issue of the training data having only a single label per example. In addition, it would save the space that static examples overlapping would occupy.

We implemented two types of dynamic sampling - true dynamic sampling, and base-sample-persistent dynamic sampling. We are going to shortly explain their differences and implementation, while the results we achieved will be presented in Chapter 5.

3.5.1 True dynamic sampling

The "true" in the name of this type of dynamic sampling comes from the fact that every original example from which the final combined example consists is randomly selected at the moment the dynamic sampling function is called, thus meaning there is no dependency between two dynamic sampling calls or dependency between samples created in one epoch and another. The pseudocode for this sampling method is shown below:

Algorithm 1: True dynamic sampling

```
function SampleTrueDynamic (minSampled, maxSampled, noClasses);
  Input  : Minimum and maximum number of sampled examples, the total number of classes
  Output : sampled audio, list of labels in combined audio
noSamples = Random(minSampled, maxSampled);
chosenLabels = RandomlySelectDistinctLabels(noClasses, noSamples);
combinedAudio = None;
for label in chosenLabels do
    // selects random audio out of all possible samples with the exception of
       samples with label label
    audio = SelectRandomAudioExcludingLabel(label) ;
    combinedAudio = CombineAudio(combinedAudio, audio) ;
end
return combinedAudio, chosenLabels ;
```

Of course, this pseudocode only shows the sampling of a single sample; when training a model using dynamic sampling this method would be called many times, each time resulting in a new dynamically sampled example, made by combining multiple examples with single labels into a new example with multiple labels. Of course, a careful reader will have already asked themselves the following question: if the samples are now dynamically generated, how many samples should I generate, if I want to maximally utilize the data I already have available? And that is an excellent question, as due to the stochastic nature of the sampling process we have no guarantee of which original samples get sampled, possibly resulting in some examples being chosen for sampling many times, while other examples are not being chosen a single time. In order to solve this, you can use the following formula:

$$N = \log\left(P, 1 - \frac{1}{n}\right) \times \frac{1}{n} \times \frac{2}{\text{minSampled} + \text{maxSampled}} \times n \tag{3.1}$$

Equation 3.1: Determining number of samples for dynamic sampling

The N in the formula denotes the number of dynamic samples you have to generate to make sure that, on average, only P % of the total number of original examples do not get sampled within one epoch. n is the total number of original examples, while minSampled and maxSampled represent the minimal and maximal number of original examples used when constructing a new, dynamically generated sample. The smaller the minSampled and maxSampled are, the larger N has to be to maintain the same P. Table 3.3 illustrates the effect minSampled and maxSampled have on N, assuming n is equal to the number of samples in the IRMAS training set, 6705, and P value of 0.1%, which we used in our experiments:

minSampled	maxSampled	N
1	2	30875
1	3	23156
1	4	18525
1	5	15437

Table 3.3: Effect of the minimal and maximal number of examples sampled using dynamic sampling on N

We can see that, if we want to dynamically sample and merge only two of the original samples at a time, we need to generate more than 30,000 dynamic samples, or almost 6x the size of the original dataset. On the other hand, if we dynamically sample and merge up to five original samples at a time, then a number of 15,437 suffices, which is rather more manageable. That is why in our experiments

we usually used 4 or 5 for the value of maxSampled, while, of course, keeping minSampled at 1, as to sample certain examples as is.

One important advantage of this sampling method, in contrast, to sequentially iterating over examples in the training dataset, is that it follows a two-step process. Firstly, we uniformly choose the labels of the examples we intend to merge. Secondly, we uniformly select an example from all the available examples with each chosen label. Consequently, this approach ensures that, on average, every label (class) is sampled an equal percentage of times, thereby naturally balancing the distribution of classes within the dataset and increasing the macro performance metrics.

3.5.2 Base-sample-persistent dynamic sampling

The second type of dynamic sampling we considered is base-sample-persistent dynamic sampling. It is similar to true dynamic sampling, except for the following difference: the sampling method receives an initial sample (audio) and then builds upon it by selecting additional samples and merging them with the base sample. In practice, you would iterate over every sample **in the original training set**, assigning that sample the role of the base-sample, and, using base-sample-persistent dynamic sampling, merge it with N other randomly chosen samples. This way, the base samples are consistent through every epoch; the only difference is the samples they are merged with.

The main idea behind this approach is to deal with very high variability in data which occurs when using true dynamic sampling, possibly leading to unstable training and the inability of the network to learn the distribution of samples. Furthermore, this sampling method **does not require** a large number of samples to work as does true dynamic sampling; as the base sample remains the same in every epoch, we have a guarantee that every sample from the original training data is going to get chosen at least once per epoch and usually more. The pseudocode of the algorithm is shown below:

Algorithm 2: Base-sample-persistent dynamic sampling

```
function SampleBasePersistent (baseSample, minSampled, maxSampled, noClasses);
  Input  : Base sample, minimum and maximum number of sampled examples, the total
           number of classes
  Output : sampled audio, list of labels in combined audio
noSamples = Random(minSampled, maxSampled);
chosenLabels = RandomlySelectDistinctLabels(noClasses, noSamples);
for label in chosenLabels do
    // selects random audio out of all possible samples with the exception of
       samples with label label
    audio = SelectRandomAudioExcludingLabel(label) ;
    baseSample = CombineAudio(combinedAudio, baseSample) ;
end
return baseSample, chosenLabels ;
```

3.6 Data splits

We split the available datasets into training, validation, and testing datasets. The same splits were then used for training, validation, and testing of every model, to enable better comparison between models.

3.6.1 IRMAS

For IRMAS, we used the original train set in unchanged form as our train set. This dataset consisted only of examples with a single label. To generate validation and test sets, we split the provided evaluation dataset, which consisted of examples with one and more labels, into validation and test sets using a ratio of 70-30. After the split, the validation dataset consisted of 862 examples and the test dataset of 2010 examples. The training dataset remained unchanged at 6704 examples.

3.6.2 Audioset

We collected a total number of 20,884 samples from the original Audioset dataset. Those samples were then split into training, validation, and testing datasets using a ratio of 70-10-20, resulting in the training set of size 14,618 examples, a validation set of size 2088, and a testing set of size 4176.

4 Modeling

An overview of the models we've used in given in Table 4.1. We've trained some of the models from scratch and used some pre-trained ones as well. The models trained from scratch were not expected to perform better than the pre-trained once. Instead, they were used to show the effectiveness of specific features and approaches, e.g. 1d vs 2d convolution.

$$l(x, y) = L = \{l_1, \ldots, l_N\},\ l_n = y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n) \tag{4.1}$$

Equation 4.1: Binary cross-entropy loss

The loss function, which was the same for all the models, was the binary cross-entropy (BCE) loss, whose loss formulation is given in Eq. 4.1. N denotes the batch size. Mean reduction was applied over the batch size. We used BCE loss instead of the CE loss because we have a multi-label problem. That means that each example can be labeled with multiple labels. CE loss is suitable for multi-class problems, where we have multiple class, but only one label can be positive.

Model	Architecture	Pre-trained	Num. params.
Raw audio 1-D CNN	CNN	No	180 K
MFCC 2-D CNN	CNN	No	879 K
Audio features 1-D CNN	CNN	No	114 K
ResNet50	CNN	Yes	23.5 M
AST	Transformer	Yes	86.2 M

Table 4.1: Overview of the models used

Throughout all the experiments we used the Adam optimizer [18]. It is the state-of-the-art (SOTA) approach, so we choose it over the standard stochastic gradient descent (SGD). As for the scheduler, we did some early experiments with the scheduler which reduces the learning rate on plateau, e.g. when loss (or potentially other specified metric) has not decreased for k epochs. The problem was that we had to define the hyperparameter k, which again requires some trial and error. Instead, we switched over to the polynomial decay scheduler. For the polynomial decay scheduler, we had to define the starting and desired ending learning rate, along with the polynomial factor. The factor of 1 corresponds to linear decay. We choose the factor of 0.7. Furthermore, we decided to use the learning rate warmup of 0.05. Warmup is the training period in which the learning rate linearly increases from effectively 0 to the defined starting learning rate. It us useful so that the initial training steps do not *overshoot* with gradient updates. Instead, small steps towards the minimum are performed at the beginning, and larger steps after the initial movement towards the minimum had already started rolling. In Figure 4.1, we show the behaviour of our learning rate throughout the training. As for the regularization, we used L2 regularization, which is implemented as weight decay in the PyTorch's version of the Adam optimizer[9]. We used different weight decay values throughout the experiments, which will be later discussed in Section 5.6.

[9] https://pytorch.org/docs/stable/generated/torch.optim.Adam.html

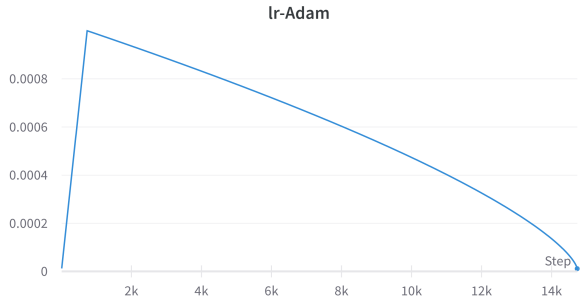


Figure 4.1: Learning rate behaviour throughout the training.

4.1 Raw audio 1-D CNN

Among the simplest convolutional approaches is the 1-dimensional convolution on the raw audio waveform. The input was a 1 second window with sample rate 44.1 kHz, resulting in the input shape of (batch size, 1, 44100). Due to the disproportional sizes, the architecture cannot be shown in an image. Instead, we show it in Table 4.2. The idea was to use the first kernel size corresponding to 10 ms of the audio as we believed that was a reasonable time frame to be able to extract some useful information. We omit mentioning ReLU and BatchNorm1d between each Conv1d and MaxPool1d layer. We also omit mentioning ReLU and dropout between the linear layers. No pre-trained layers were used. The hyperparameters are the same as in Table 4.3.

Layer	Shape in	Kernel
Conv1d	(B, 1, 44100)	(1, 440), stride = 2
MaxPool1d (4×4)	(B, 32, 21831)	-
Conv1d	(B, 32, 5457)	(1, 6), stride = 1
MaxPool1d (4×4)	(B, 32, 5452)	-
Conv1d	(B, 32, 1363)	(1, 3), stride = 1
MaxPool1d (4×4)	(B, 64, 1361)	-
Conv1d	(B, 64, 340)	(1, 3), stride = 1
MaxPool1d (4×4)	(B, 64, 338)	-
Conv1d	(B, 64, 84)	(1, 3), stride = 1
MaxPool1d (4×4)	(B, 128, 82)	-
Conv1d	(B, 128, 20)	(1, 3), stride = 1
MaxPool1d (4×4)	(B, 256, 18)	-
AvgPool1d	(B, 256, 4)	-
flatten	(B, 256, 1)	-
linear_1	(B, 256)	-
linear_2	(B, 64)	-
output (no weights)	(B, 11)	-

Table 4.2: Architecture of a 1-D CNN on raw audio

4.2 MFCC 2-D CNN

MFCC is a 2-dimensional feature, which makes it suitable for 2-d convolutional layers. We decided on extracting 40 cepstral coefficients in order to make more room for our convolutions and in belief that it would potentially bring more valuable information, as explained in Section 3.1.1. Due to lack of time, we did not experiment with 13 coefficients, although it would be an interesting comparison.

We leave that for future work. We additionally resize the width of the image to 256 in order to make it universal and capable of handling different input sizes. The only input length we have tried was the 1 second window at the sample rate of 44.1 kHz. The architecture is shown in Figure 4.2 and hyperparameters in Table 4.3. No pre-trained weights were used. The input had one channel and we apply four convolutional blocks which contain (Conv2d, ReLU, BatchNorm2d, MaxPool2d). Afterwards, average 2-dimensional pooling is applied, tensor is flattened and fed into two linear layers with a 20 % dropout and ReLU non-linearity in between.

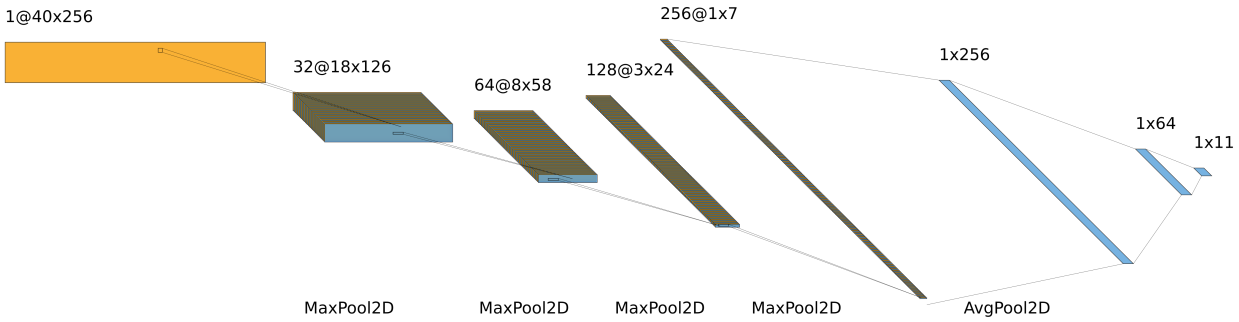


Figure 4.2: 2-dimensional CNN architecture on MFCC features

Hyperparameter	Value
Initial learning rate	1e-3
Final learning rate	1e-5
Learning rate scheduler	Polynomial decay, power=0.7, warmup=0.05
Number of epochs	35
Batch size	16
L2-regularization factor	5e-4

Table 4.3: Hyperparameters of our 1-D raw audio, 2-D MFCC, and 1-D audio features CNN models

4.3 Audio features 1-D CNN

For the convolutional model on the extracted audio features, we take a very experimental approach. After discussing the statistical significance of the features in Section 2.5, we omit tonnetz feature due to its insignificance. Additionally, we omit spectrogram, as it was covered by other, more powerful models (see Sections 4.4, 4.5). After including the features, we are left with 40 of them. Now, since most of these features do not interact, it would be incorrect to just stack the features vertically. Instead, we decided to feed each feature as a separate channel into the 1-dimensional convolution. Feeding each feature in a separate channel allows the model to learn individual patterns and correlations for each feature. However, it does not allow feature interactions. This is something we do want to avoid between, for example, zero-crossing rate and chroma features. However, this is something we would like to have between chroma features themselves. So, this approach was done out of curiosity to see if it would be more beneficial for the model to avoid learning unnecessary correlations between uncorrelated features more than it would be learning the correlations between both correlated and uncorrelated features. Unfortunately, due to the lack of time, we did not have time to test the 2-dimensional convolution on all the audio features. Thus, we leave that for future work. However, we can compare the results with the previously discussed 2-d convolution of MFCC features, which we will do in Chapter 5.

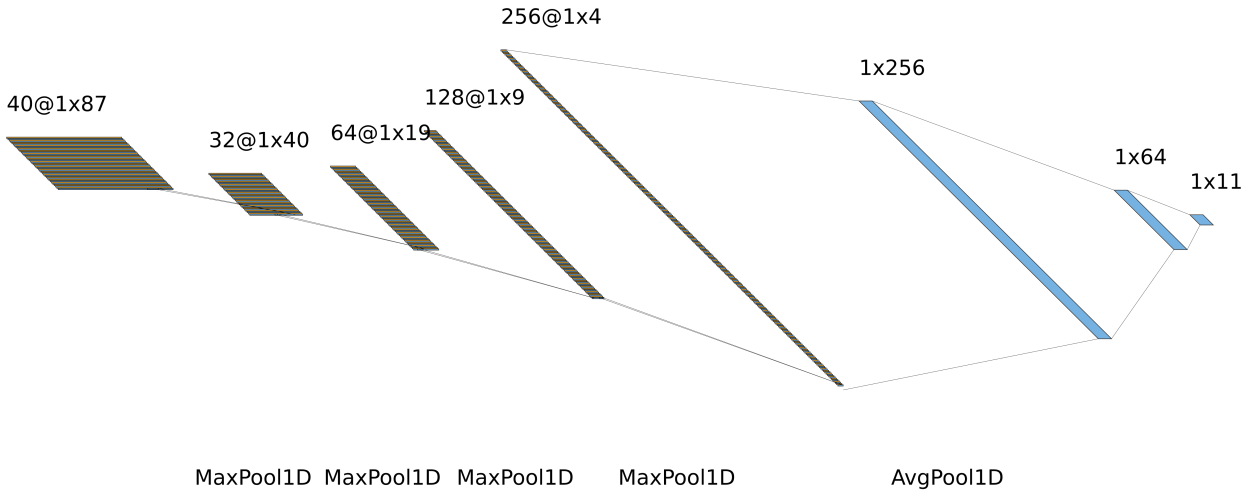


Figure 4.3: 1-dimensional CNN architecture on extracted audio features

The architecture of the model is shown in Figure 4.3 and the hyperparameters in Table 4.3. Again, the only window we experimented with was or size 1 second. Consequently, with n_fft, hop_length, frame_length, and sample_rate equal to 2048, 512, 2048, and 44100 respectively, we obtain an input of shape $H \times W$. Here, H is the number of features (40) and W is the example length derived when using the mentioned parameter values (87). The features are, however, fed into separate channels. Again, four (Conv1d, ReLU, BatchNorm1d, MaxPool1d) blocks are used with the average 1-dimensional pooling applied afterwards. Finally, the tensor is flattened and fed into two linear layers with a 20 % dropout and ReLU non-linearity in between. Logits are returned from the model's forward pass.

4.4 Spectrogram – ResNet50

Perhaps one of the most famous and widely used neural network architectures is ResNet, first introduced in [2]. In an attempt to mitigate the problems of vanishing and exploding gradients, which tend to appear more and more often as the depth of the network increases, authors of ResNet introduce a concept of a **residual unit**. Residual units enable the network to

(a) utilize skip connections, allowing the input data to skip a couple of layers, which in turn helps stop the exploding/vanishing gradient problems and reduce overfitting, by acting as a sort of dropout

(b) learn **residual mappings**, which allow the layer of a network to form an identity function that maps to an activation earlier in the network when a specific layer's activation converges to zero in the current layer. This also helps with the degradation of performance in the deeper layers, as the network can simply "revert" to an earlier layer's output

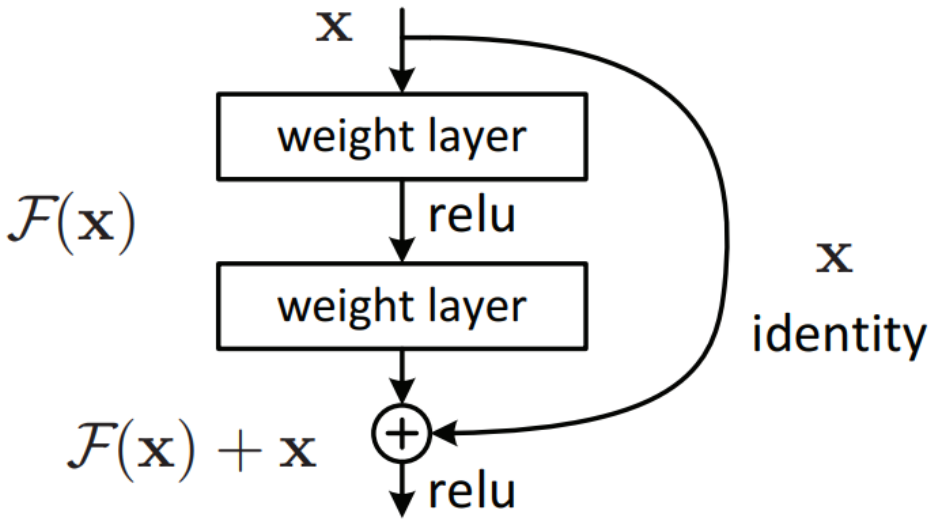


Figure 4.4: Visualisation of a residual unit, one of the key building blocks of residual networks [2]

Due to its unique architecture and the use of residual blocks, residual networks support a large number of layers - it is not uncommon for a residual network to have 50, 101, or even 152 layers. The number of layers used in residual networks is usually denoted by a number next to their name; for example, a residual network with 101 layers would be denoted as ResNet-101.

Of course, training such a deep network from scratch would require a large amount of data in order for the network to work well. So, a usual approach is to pre-train the network on a general image-related task (if the network is used for computer vision, of course, as ours was), such as classification on the large ImageNet dataset. The weights from that pre-trained model are then used as a starting point, in addition to replacing the final fully-connected layer with a new one, depending on the specific nature of the problem. This approach is known as **transfer learning** and is widely used.

We choose ResNet as the main convolutional network for spectrograms as it has proven itself to be a powerful and versatile network well suited to a wide array of applications. Furthermore, it has a large capacity, allowing it to better learn the distribution of our data, especially when using augmentation methods such as dynamic sampling, which add a high degree of variability.

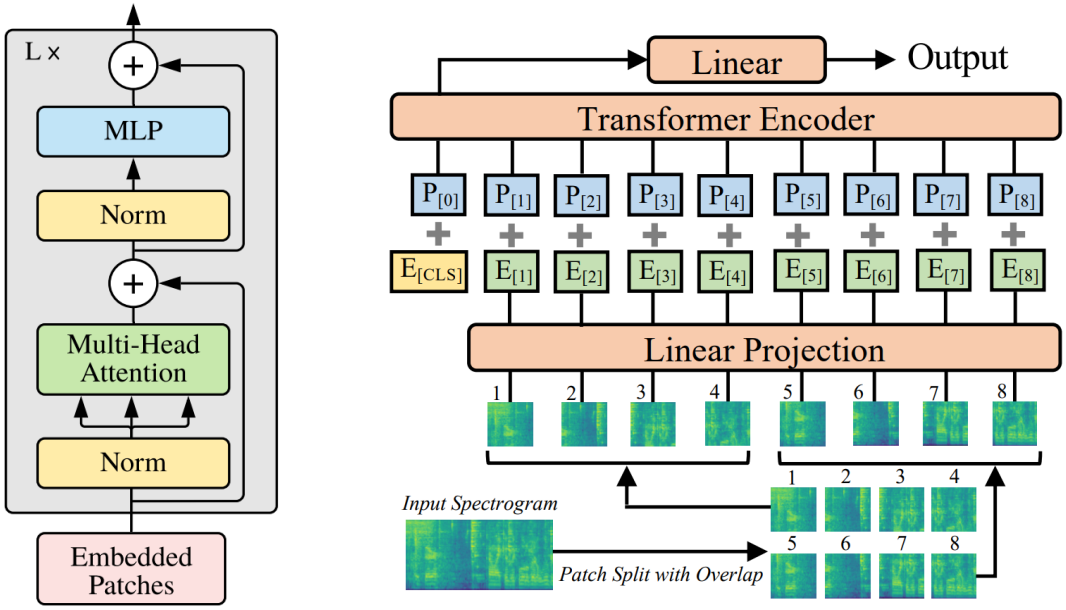
Our version of the ResNet architecture was ResNet-50, pre-trained on ImageNet-1K. We replace the final fully-connected layer with our own and add a dropout layer with p=0.2 between the FC layer and the output of the base ResNet-50 model. Hyperparameters used to train all the ResNet-50 models were, unless explicitly stated otherwise, the ones in Table 4.4.

Hyperparameter	Value
Initial learning rate	2e-4
Final learning rate	1e-6
Learning rate scheduler	Polynomial decay, factor=0.7, warmup=0.05
Number of epochs	35
Batch size	16
L2-regularization factor	1e-5

Table 4.4: Standard hyperparameters of our ResNet-50 model

4.5 Audio Spectrogram Transformer

The only model with no convolutional layers is the Audio Spectrogram Transformer (AST) [19]. There are multiple layers to decouple in the definition and background of AST. First, the backbone of the whole architecture is the Transformer [20]. It was originally proposed for the Natural Language Processing (NLP) problems. However, due to its incredible parallelism, model depth, and attention mechanism, it had achieved great success in models such as BERT [21], GPT-3 [22], XLNet [23], T5 [24] etc. Consequently, it was adjusted to be able to handle vision inputs in the similar way it handles text – in parallel. That is how the Vision Transformer (ViT) [25] was born. Finally, since spectrograms are the most often feature used in audio classifications, ViT was further trained on the complete Audioset dataset [14](Section 2.4). That is the simplified story of how AST came to life. In the next few paragraphs, we will go over the important details of the (Vision) Transformer.

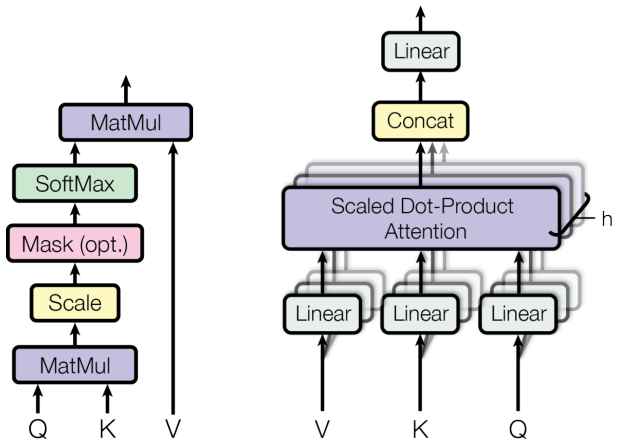


(a) Transformer encoder [25] **(b)** AST architecture [19]

Figure 4.5: Audio Spectrogram Transformer [19, 25]

The originally proposed Transformer is made of the encoder and decoder part. The encoder's task is to extract useful information from the input text (image), while the decoder's task is generating text (image) from the input and previously extracted information. As we require only the discriminative model, we discard any discussion on the decoder and focus solely on the encoder. It is shown in Figure 4.5(a). The input, which we will explain later, is passed onto the sequence of L layers, whose architectures are shown in grey. In out case, L was equal to 12. After normalization all the inputs (text tokens or image patches) are in parallel passed onto the Multi-Head Attention (MHA) sub-layer. *Multi-head* indicates multiple attention heads (layers), but let's focus on a single one for now. First, each input token (patch), is used to create a Query (Q) vector, a Key (K) vector, and a Value (V) vector. These vectors are created by multiplying the embedding by Query, Key, and Value matrices respectively. These matrices are trained end-to-end during the process. The resulting Q, K and V vectors are passed onto the scaled dot-product attention (Figure 4.6(a)) to calculate how much does each token (patch) *attend* to the others, i.e. how connected or important they are to each other in some abstract way which the model learns itself. It is important to mention that this is the simplified explanation, since the vectors are passed as part of the matrices in the original implementation. After calculating the scaled dot-product attention of the matrices for each head, the outputs are concatenated and multiplied with yet another weight matrix (Figure 4.6(b)). What follows are the residual connection and another layer normalization. Transformer architecture is well equipped with residual connections due to its depth, which allows the gradients to be passed all the way to the first

layers during gradient backpropagation. To continue, the output is passed through two feed-forward layers, another residual connection is applied and the final outputs are passed onto the next layer for the same treatment. Each layer receives the output of the previous layer, with the exception of the first layer, which receives the input, which we will now discuss.



(a) Scaled dot-product attention [25]

(b) Multi-Head Attention [19]

Figure 4.6: Transformer's attention mechanism

The biggest contribution of ViT is, in our opinion, the way they adjust the image input to be appropriate for the Transformer architecture, just as text tokens are. As discussed in [25], the standard Transformer receives as input a 1D sequence of token embeddings. To handle images, they reshape the image $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$ into a sequence of flattened 2D patches $\mathbf{x}_p \in \mathbb{R}^{N \times (P^2 \cdot C)}$, where ($H$, W) is the resolution of the original image, C is the number of channels, (P, P) is the resolution of each image patch, and $N = HW/P^2$ is the resulting number of patches, which also serves as the effective input sequence length for the Transformer. The Transformer uses constant latent vector size D through all of its layers. Thus they flatten the patches and map to D dimensions with a trainable linear projection (see Figure 4.5). The output of this projection is referred to as patch embeddings. For the base-sized models (and the model we had used) the size of the patch embedding is 768.

There is a slight difference in creating the input for AST. According to [19], the input audio waveform of t seconds is first converted into a sequence of 128-dimensional log Mel filterbank (fbank) features are computed with a 25ms Hamming window every 10ms. This results in a $128 \times 100t$ spectrogram as input to the AST. The spectrogram is then split into a sequence of N 16×16 patches with an overlap of 6 in both time and frequency dimension, where $N = 12 \lceil (100t - 16)/10 \rceil$ is the number of patches and the effective input sequence length for the Transformer. The patch is again flattened and linearly projected to a patch embedding of size 768. In order to fix these previously discussed numbers, AST requires users to use the sample rate of 16 kHz. Thus, we down-sampled our audio files from 44.1 kHz to 16 kHz.

Since the patch embedding is not in temporal order and the architecture itself has no mechanism of capturing the input order, they add a trainable 768-dimensional positional embedding in order to capture the information about the spatial structure of spectrograms. Positional embeddings are denoted with a blue color and letter `P` in Figure 4.5. The most important embedding, however, is the `[CLS]` embedding which is appended at the beginning of the sequence and passed later onto the single linear layer to obtain the output. As its name suggests, this embedding is used for classification purposes. The Transformer Encoder's output of the `[CLS]` embedding serves as the audio spectrogram representation [19]. In our use case, the 768-dimensional `[CLS]` embedding was linearly projected by a 768×11 (num_classes) matrix to obtain `logit` outputs for each class. We

used Huggingface[10] implementation of an AST.

According to Huggingface's specification, a maximum spectrogram length that can be fed into AST is 1024. With the sample rate of 16 kHz, this is equal to about 10 seconds of audio. Longer files get truncated, while the shorter ones are padded. We tried two approaches. First, we used AST as is, with pre-trained position embeddings and 10 second audio files. This has downsides of (a) longer training time due to longer (default) input and (b) much of padding and truncating is applied. Second, we tried fixing the input length to 1 second by using multiple windows on a single file. This has a downside of training completely new positional embeddings, due to the new width of input spectrograms. In total $1214 \times 768 = 932,352$ positional embeddings were discarded and $110 \times 768 = 84,480$ new ones were randomly (with normal distribution) initialized and trained. Just as a reference, the discarded positional embeddings contain more parameters than any of the raw audio 1-D CNN, MFCC 2-D CNN, and audio features 1-D CNN models! However, the benefit of such approach is a much faster training time and no padding/truncating. Hyperparameters used to train all the AST models were, unless explicitly stated otherwise, the ones in Table 4.5. We increased number of gradient accumulation steps according to the maximum batch size that could fit into the GPU memory. For the 10 second, the maximum batch size was 1, so we increase gradient accumulation steps to 16. Separate optimizer and schedulers were used for the (a) classifier and (b) all the other parameters. The reason for this is that the classifier is not pre-trained. Instead, it is randomly initialized using the normal distribution. Thus, we do not want to preserve any knowledge from before by using a small learning rate, like we do for the rest of the AST parameters.

Hyperparameter	Value
Initial learning rate (base)	2e-5
Final learning rate (base)	2e-6
Initial learning rate (classifier)	1e-3
Final learning rate (classifier)	1e-4
Learning rate scheduler	Polynomial decay, factor=0.7, warmup=0.05
Number of epochs	35
Batch size	16
L2-regularization factor (base)	1e-4
L2-regularization factor (classifier)	1e-3

Table 4.5: Standard hyperparameters of our AST model

4.6 Window size

Audio files are, as we already mentioned multiple times, unique in many ways when compared to other forms of data. One of the main ways they are unique is their variable size, which can often present a problem for neural networks. A good example is the IRMAS dataset – all the examples from the training set are 3s in length, while the data used for testing can vary anywhere between 5 and 20s. So, it is important that our model is good at working with files of different lengths, often different from what the model had been trained on. A good way of solving this is by splitting the full-length audio into windows of the same size, analyzing each section and predicting instruments it contains, and then aggregating outputs of all the windows into a single output, which is then equal to the prediction for the entire audio clip. By doing this, our network can be efficiently trained to work with fixed-size inputs (for example, 1s), and can later work with inputs of any size, as all inputs are split into windows equal in length to the window the network had been trained on. This approach is inspired by the work of [6], who also empirically prove that neural networks trained

[10] https://huggingface.co/docs/transformers/main/en/model_doc/audio-spectrogram-transformer

on the task of predominant instrument recognition are more efficient when trained and evaluated using shorter window sizes (1s) when compared to larger window sizes (3s). They claim that using a shorter window size helps obtain local instrument information. We confirm this finding in our own results, as shown later in Section 5.2.

IRMAS dataset is especially suited to this approach, as it guarantees that the instrument audio clip is labeled with is playing the entire length of the clip. This means that when training a model, original examples of lengths N can be split into M windows, each of lengths N/M, and each carrying the label of the original clip. In addition to allowing the network to train on shorter clips, this also acts as an augmentation strategy, increasing the number of examples used during training.

4.7 Aggregation functions

The question remains, once a prediction is made for every window of the original audio clip, how to aggregate all the predictions into a final prediction?

The authors of [6] provide two aggregation strategies. The first, one, they call S1, is simply taking the average of the sigmoid outputs **class-wise**, and then thresholding them without normalization. The goal of this method is to "capture the existence of each instrument with its mean probability such that it might return the result without any detected instrument" [6]. The second strategy, called S2, is done as follows: all sigmoid outputs are summed class-wise over all the window-based predictions for the audio clip. The values are then normalized by dividing with the maximum value amongst classes, so the normalized values fall into the [0,1] range. The normalized values are then thresholded, and all the classes with values over the threshold T, are taken as positive. The logic behind this method is based on the assumption that humans perceive the "predominant" instrument in a scaled sense; the strongest instrument is always detected, and the detection of the other instruments is judged relative to their strength compared to the most dominant instrument. In our case of predominant instrument classification, all instruments with predicted values above the threshold (after aggregation) are considered predominant. The authors found that, in general, the S2 aggregation strategy with a threshold value of 0.5 provides the best results, so this was the strategy we also used in our experiments.

5 Evaluation

In this section, we will go over evaluating our experiments. After building the models and running them, it is important to evaluate and compare them properly. Additionally, we will describe some additional steps we took in order to best utilize our models and their outputs to achieve better results.

5.1 Metrics

To start with, we define the metrics used for the evaluation of our models. As stated by the competition organizer, **hamming score** is used for the final evaluation, so we include it in our evaluation as well. For the multi-label classification problem, Hamming score, macro accuracy, and micro accuracy have the same metric values. However, Hamming score does not account for label imbalance. As we believe that performing well on all of the labels, not just the high-resource ones, is very important, we include **macro F1-score** in our evaluation. Additionally, when we want to discuss the precision and recall of some of our models separately, we discuss the **macro precision** and **macro recall**. Additionally, we include **exact match accuracy**, which accounts for the number of examples whose predictions were exactly matched divided by the total number of examples.

When choosing the best model, we look into the performance on the validation set. Since we kept the maximum of one checkpoint during the entire training, we needed to define the validation metric which will be used for deciding whether or not a checkpoint should be saved. We choose the validation loss as the metric. More specifically, we chose the validation loss calculated after performing the S2 aggregation over the validation examples. The reason behind this decision is that the same aggregation technique will be used for testing the model, so we wanted to actually save the model that is later most likely to perform the best on test data. We did not choose other metrics, such as hamming score and macro F1-score, because they do not give any insight into how confident is the model in its predictions. A checkpoint with the lowest validation loss should be the most general and should not suffer from overfitting.

5.2 Window sizes

We start by discussing the effect of different window sizes (Section 4.6) during training and evaluation. We conduct this experiment using the ResNet model. Three different models were trained using respective window sizes of 1, 2, and 3 seconds on training, validation, and test data. The S2 aggregation function was used during evaluation, as introduced in Section 4.7. The results are shown in Figure 5.1. We observe that using the window size of 1 second performs slightly better than the window size of 2 or 3 seconds. Thus, we use the window size of 1 second and the aggregation function S2 throughout most of our experiments conducted on the IRMAS dataset later.

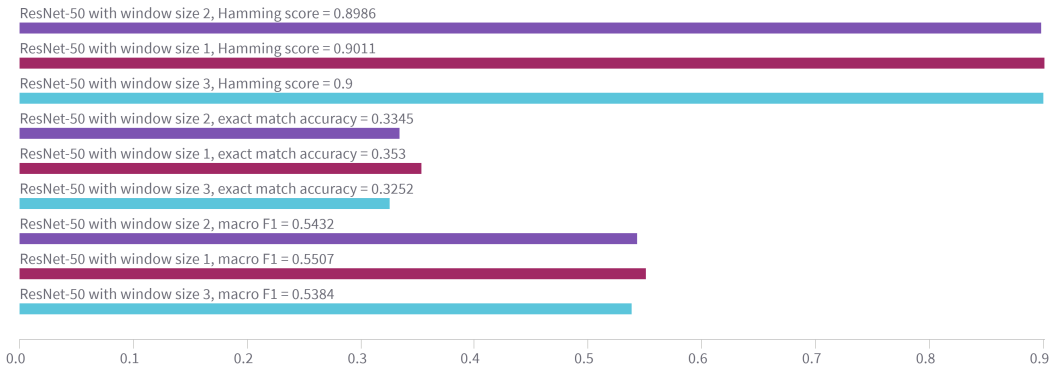
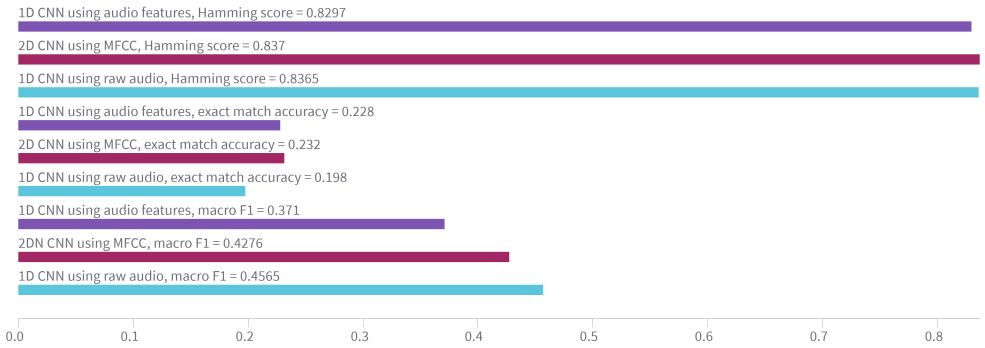


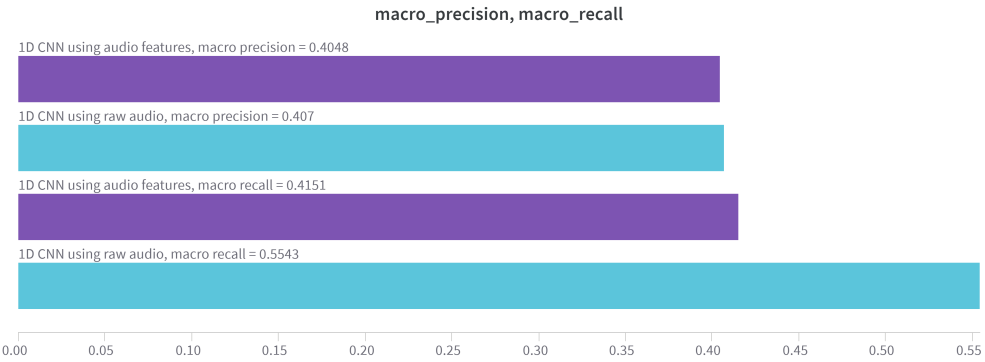
Figure 5.1: ResNet performance on IRMAS test set using different window sizes

5.3 Non-pretrained CNN models

After finding the optimal window size for the IRMAS dataset, we evaluate our non-pretrained CNN models. All the models were trained exclusively on IRMAS, using the previously found optimal window size of 1s. The performance comparison is shown in Figure 5.2(a). The window size was 1 and the aggregation function was S2. To start with, the performance is significantly worse than ResNet. That is expected since ResNet is a pre-trained model with 27×, 131×, and 206× more parameters than MFCC 2-D CNN, raw audio 1-D CNN, and audio features 1-D CNN respectively. Next, comparing the performance of the non-pretrained CNN models themselves, it varies by different degrees for different metrics. Although the hamming score is very similar for all three models, with 1-D CNN audio features model being slightly worse, exact match accuracy performance and especially macro F1 performances vary. After noticing such a discrepancy, we additionally decided to plot the macro precision and macro recall of the two models on Figure 5.2(b). The plots show that the CNN raw audio model has a much higher recall than the CNN audio features model, with their precision values almost the same. This suggests that performing a convolution directly on the raw audio allows us to catch more of the total positive labels. However, when claiming that some label in the example is positive, these two models have a similar hit rate. The fact that this happens implies that the CNN model on raw audio causes more positive labels in total compared to the CNN model on audio features, with a similar hit rate when claiming that some instrument (or voice) is present in the example. This suggests the convolution on the raw audio managed to learn the data better than the convolution on the audio features passed into separate channels.



(a) Hamming score, exact match accuracy, and macro F1 score comparison on IRMAS test set.



(b) Macro precision and recall comparison of 1-D CNN audio features and 1-D CNN raw audio models.

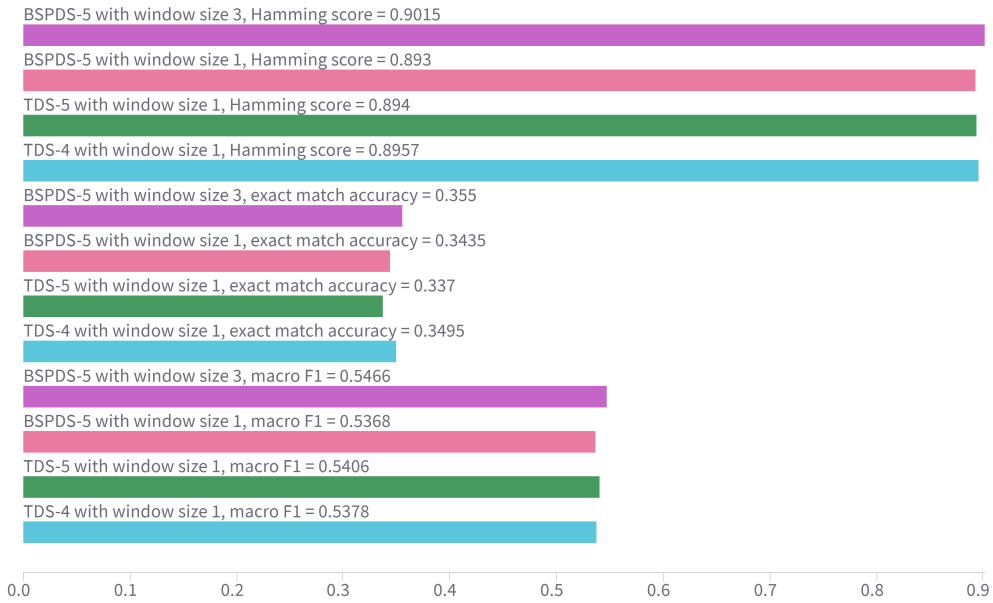
Figure 5.2: Performance comparison of the non-pretrained CNN models using the window size 1s

Regarding the better exact match accuracy performance of the audio features CNN model, it is probably due to the model learning some spurious patterns in the data. This possibly led to the model performing well on specific audio instances (e.g. predicting all the labels correctly), but being unable to generalize well. This shows the importance of using and inspecting different performance metrics before making any conclusions.

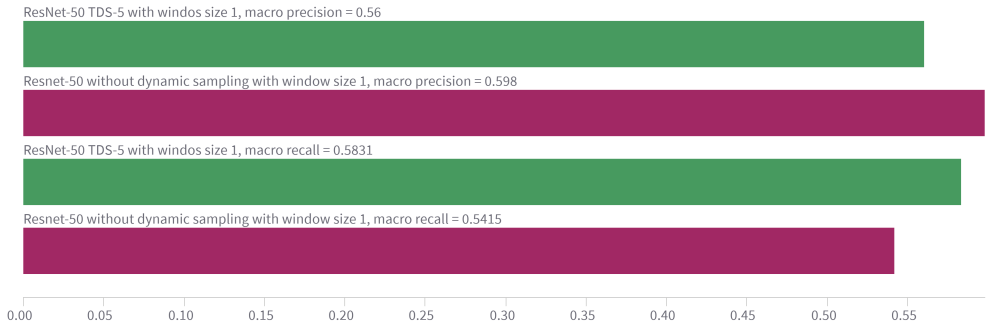
5.4 Dynamic Sampling

Two dynamic sampling methods were introduced in Section 3.5. For the sake of comparison of the dynamic sampling methods (with the non-dynamic sampling model), we go back to the ResNet. The performance comparison is shown in Figure 5.3(a). As previously mentioned, we sample and overlap anywhere from 1 to 5 examples at a time. We have experimented with sampling the maximum of 4 and the maximum of 5 examples as the time, which is denoted as -4 and -5 on the graph labels. *true-dynamic-5* denotes that 5 samples were dynamically sampled and overlapped. *base-sample-persistent-dynamic-5* indicates that, including the current sample, a maximum of 5 samples were included, i.e. maximum of 4 additional. We also experiment with window sizes of 1 and 3 seconds, which is denoted on the graph as *window-1* and *window-3*. Two different dynamic sampling techniques are denoted as *true-dynamic* and *base-sample-persistent-dynamic*. None of the 4 examined methods achieves significantly better results than the others. Since dynamic sampling generally causes more positive labels within the sample, we were interested in how that affects precision and recall. Figure 5.3(b) shows that the model trained using dynamic sampling has higher recall and lower precision. This intuitively makes sense. Dynamic sampling caused the model to see more positive labels within the samples during training, which consequently makes it label more of them as positive during evaluation. This increases the recall, but decreases the precision, leaving the

F1-score roughly the same. This is useful if one needs to build models to which either precision or recall is of greater importance than the other of two.



(a) IRMAS test set comparison of different dynamic sampling methods using the ResNet model. BSPDS is short for base sample persistent dynamic sampling, TDS for true dynamic sampling. The number next to the acronym designates the maximum number of sampled files.



(b) Macro precision and recall comparison of the dynamic and non-dynamic method.

Figure 5.3: IRMAS test set comparison of different dynamic sampling methods using the ResNet model.

5.5 Audioset

We trained our models on the Audioset models as well. For the sake of comparison, we show 4 combinations of (train, test) datasets – (Audioset, IRMAS), (Audioset, Audioset), (IRMAS, Audioset), (IRMAS, IRMAS) respectively. All the combinations used ResNet model. In Figure 5.4, we compared their performances. The models listed here in the text from left to right are listed in the plot from higher to lower. The models trained and evaluated on the same dataset (orange, green) work much better than models which were trained on one and evaluated on the other dataset (purple, grey). Audioset, in general, has higher metrics compared to IRMAS, which suggest that it is easier and possibly cleaner. The model trained on Audioset and tested on IRMAS (purple) is significantly worse than the model both trained and tested on IRMAS (green). The same goes for vice versa (grey and orange). We were hoping that introducing a new dataset with more balanced and numerous labels would bring some performance points. However, that was not the case. Neither

of the models (trained on either Audioset or IRMAS) managed to generalize well on the sound from other datasets.

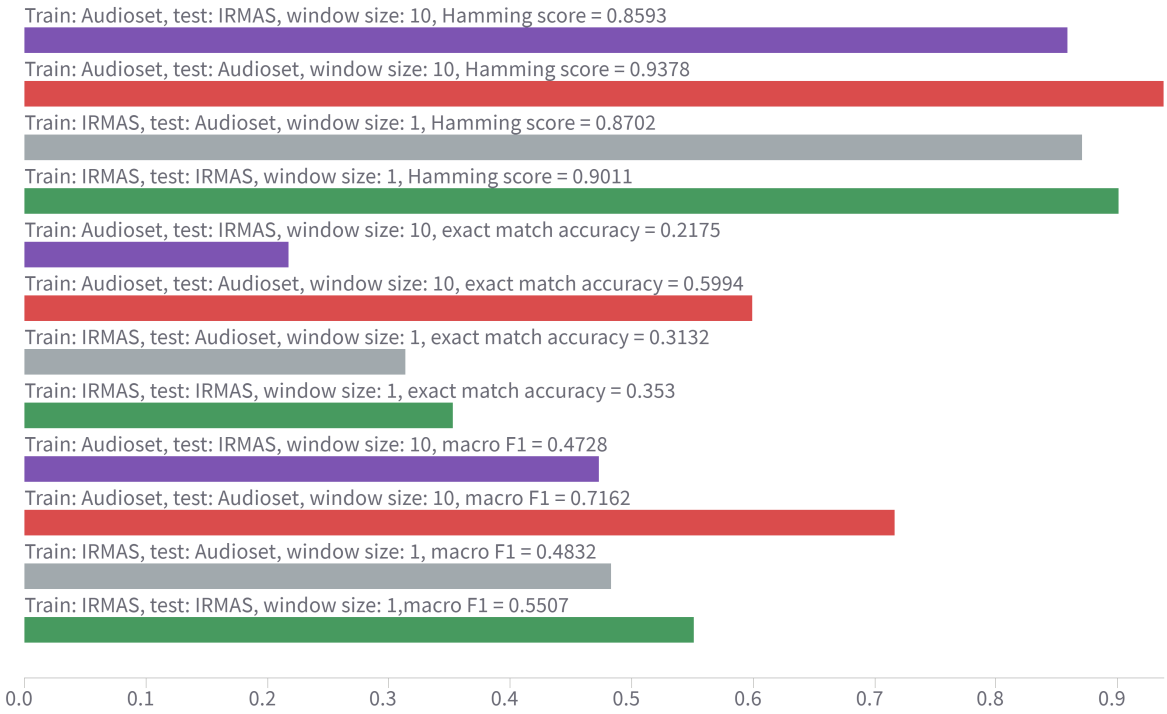
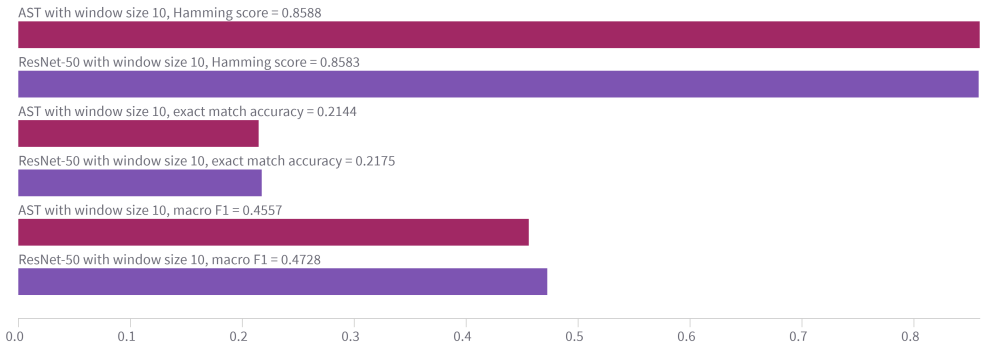


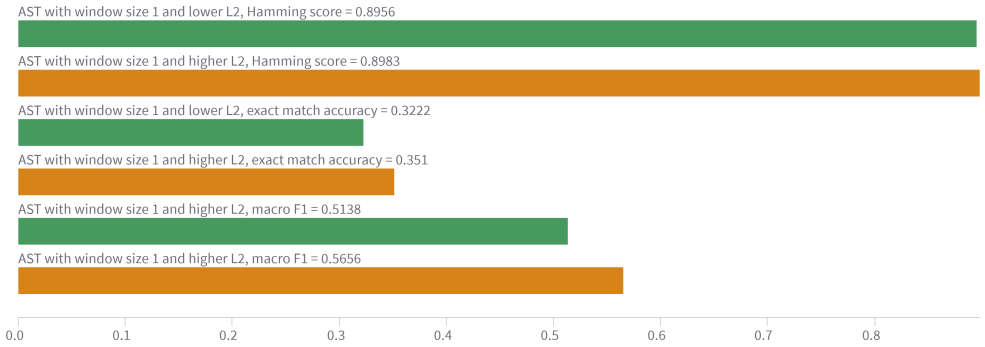
Figure 5.4: Comparison of the ResNet model(s) trained and evaluated on Audioset/IRMAS

5.6 AST

AST was expected to perform better than other models due to its powerful architecture and the fact that it was pre-trained on the complete Audioset. The comparison of the AST and ResNet models both trained on Audioset and tested on IRMAS is shown in Figure 5.5(a). Unlike our expectations, the models perform very similarly, with ResNet even performing slightly better. After further inspection of the AST training and validation metrics, we decided to increase the L2 regularization by increasing the weight decay in Adam from 1e-5 and 1e-5 to 1e-3 and 1e-4 for the optimizer of the classifier and all the other parameters respectively. The performance of IRMAS training and validation loss of the same AST model, with the only difference in L2 regularization, is shown in Figure 5.6(a, b). We can see that the training loss of the AST model with less regularization is lower than the one with more regularization. However, validation loss of AST with more regularization reaches a lower peak much sooner than the model with less regularization. What's more, that peak is much lower than the one of the models with lower regularization, especially if you take into account that smoothing was applied on the graph. The performance difference on the IRMAS set is shown in Figure 5.5(b). Again, the model with more regularization performs significantly better. This leads to the conclusion that, due to many parameters, AST model is too complex for the problem and overfits to the training data. Consequently, we choose the model with higher regularization.



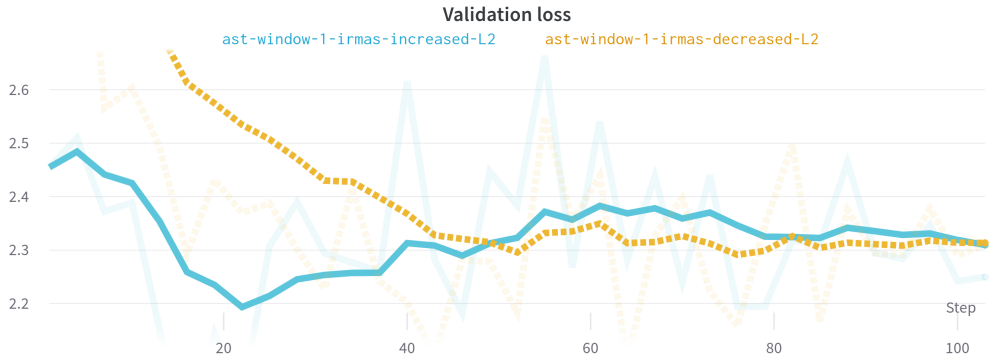
(a) Comparison of the RestNet and AST model trained on Audioset and tested on IRMAS.



(b) Comparison of the same AST models with different L2 regularization trained on IRMAS.

Figure 5.5: AST performance.

(a) Training loss.



(b) Validation loss with smoothing of 0.75.

Figure 5.6: AST L2 regularization comparison.

5.7 General

The overview of the computational demands and training times for all of the models is shown in Figure 5.7. All the models use a batch size of 16 and the window size of 1. Any other (hyper)parameters that would affect the training times of the models are also the same. Due to the reduced spectrogram width when using window size 1 for AST, it is actually the fastest-training model, although it has by far the most parameters. This is mostly due to the fact that Transformers are data-parallel and, due to their success, training them on GPUs became more and more efficient. On the other hand, 1-D CNN on audio features is the longest and least efficient of all the training models. When we add to this its performance, we come to the conclusion that our experimentation of passing each feature as a separate channel into 1-D convolution was not a successful idea. However, one first needs to fail trying many times in order to achieve something great.

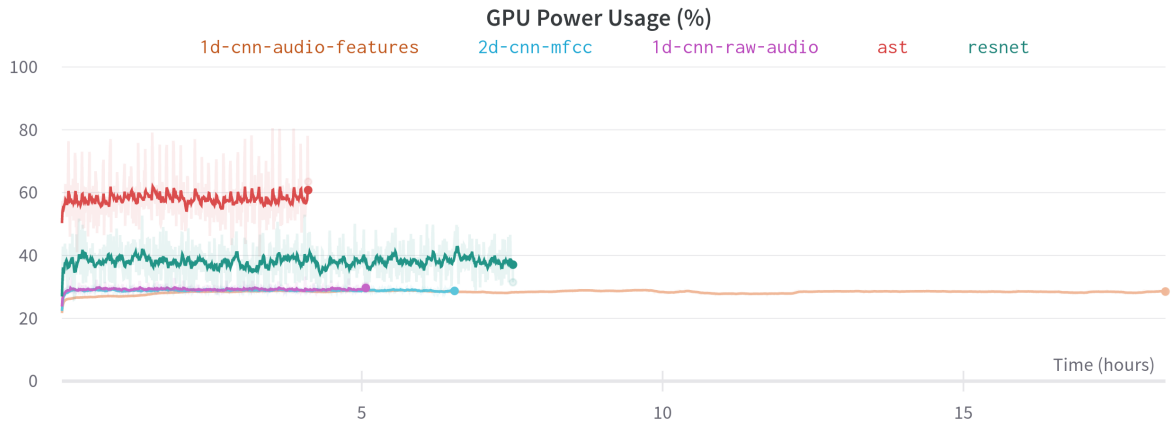


Figure 5.7: Comparison of the computational demands and training times of all the models using window size 1s on IRMAS.

5.8 Final model

Choosing a final model proved to be an interesting task, to say the least. There were three final contenders, all trained on IRMAS: the surprisingly well-performing ResNet-50 trained on static data with window size 1, the ResNet-50 model trained using base-sample-persistent dynamic sampling with window size 3, and the AST trained using window size 1 and stronger regularization.

In the end, we decided to choose the ResNet-50 model trained with dynamic sampling as our final model; simply as it was, technically, the best-performing model on 2 out of 3 metrics, and due to the smaller size and overhead compared to the AST.

However, we believe the true value of our work does not lie in the exact model we decided to choose, but in the various new approaches we introduced, explained, and meticulously evaluated and compared. We believe our work can serve as a good starting point for future exploration and research of new ideas regarding instrument detection and audio analysis in general. One idea we are particularly intrigued by, which we did not have enough time to properly implement and evaluate, is an ensemble of dynamically and statically trained models - we believe this combination of models could lead to an improvement in the results, as the ensemble model could benefit from the increased recall of the dynamic model, while at the same time not loosing too much precision in the process due to the influence of the statically trained model. Of course, there are many more excellent ideas waiting to be discovered in the exciting task of instrument recognition and audio analysis.

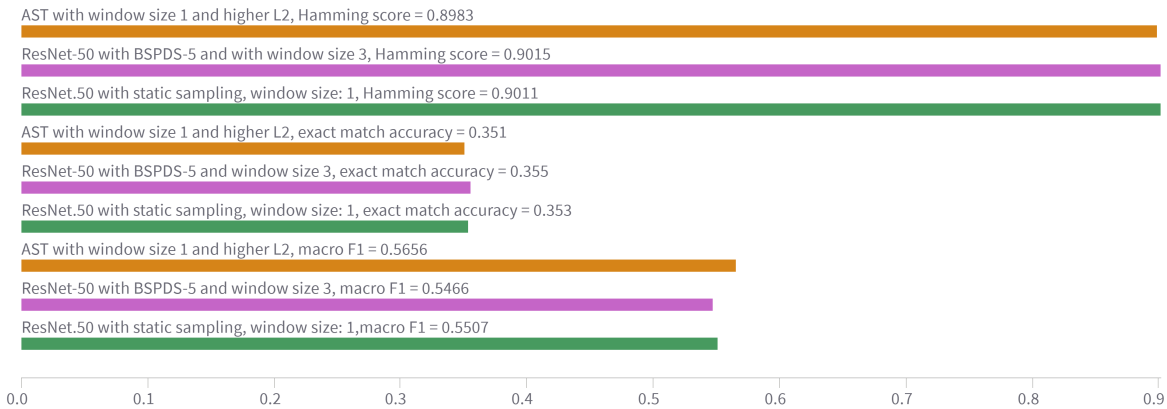


Figure 5.8: Comparison of the best-performing models taken into consideration for the final model

Bibliography

[1] Qingkai Kong, Timmy Siauw, and Alexandre Bayen. *Python programming and numerical methods: A guide for engineers and scientists.* Academic Press, 2020.

[2] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, Los Alamitos, CA, USA, jun 2016. IEEE Computer Society.

[3] K. Kashino and H. Murase. Music recognition using note transition context. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*, volume 6, pages 3593–3596 vol.6, 1998.

[4] Karthikeya Racharla, Vineet Kumar, Chaudhari Bhushan Jayant, Ankit Khairkar, and Paturu Harish. Predominant musical instrument classification based on spectral features. In *2020 7th International Conference on Signal Processing and Integrated Networks (SPIN)*. IEEE, feb 2020.

[5] Lekshmi C.R and Rajeev Rajan. Transformer-based ensemble method for multiple predominant instruments recognition in polyphonic music. *EURASIP Journal on Audio, Speech, and Music Processing*, 2022, 05 2022.

[6] Yoonchang Han, Jaehun Kim, and Kyogu Lee. Deep convolutional neural networks for predominant instrument recognition in polyphonic music. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 25(1):208–221, 2017.

[7] Peter Knees and Markus Schedl. *Music similarity and retrieval: an introduction to audio-and web-based strategies*, volume 9. Springer, 2016.

[8] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[9] Fabien Gouyon, François Pachet, Olivier Delerue, et al. On the use of zero-crossing rate for an application of classification of percussive sounds. In *Proceedings of the COST G-6 conference on Digital Audio Effects (DAFX-00), Verona, Italy*, volume 5, page 16, 2000.

[10] Dan-Ning Jiang, Lie Lu, Hong-Jiang Zhang, Jian-Hua Tao, and Lian-Hong Cai. Music type classification by spectral contrast feature. In *Proceedings. IEEE International Conference on Multimedia and Expo*, volume 1, pages 113–116. IEEE, 2002.

[11] Brian McFee, Colin Raffel, Dawen Liang, Matt McVicar, Eric Battenberg, and Oriol Nieto. librosa: Audio and music signal analysis in python. 2015.

[12] Christopher Harte, Mark Sandler, and Martin Gasser. Detecting harmonic change in musical audio. In *Proceedings of the 1st ACM workshop on Audio and music computing multimedia*, pages 21–26, 2006.

[13] Juan J Bosch, Jordi Janer, Ferdinand Fuhrmann, and Perfecto Herrera. A comparison of sound segregation techniques for predominant instrument recognition in musical audio signals. In *ISMIR*, pages 559–564, 2012.

[14] Jort F. Gemmeke, Daniel P. W. Ellis, Dylan Freedman, Aren Jansen, Wade Lawrence, R. Channing Moore, Manoj Plakal, and Marvin Ritter. Audio set: An ontology and human-labeled dataset for audio events. In *Proc. IEEE ICASSP 2017*, New Orleans, LA, 2017.

[15] Ronald E Walpole, Raymond H Myers, Sharon L Myers, and Keying Ye. *Probability and statistics for engineers and scientists*, volume 5. Macmillan New York, 1993.

[16] Aleksandr Diment, Padmanabhan Rajan, Toni Heittola, and Tuomas Virtanen. Modified group delay feature for musical instrument recognition. 2013.

[17] Daniel Park, William Chan, Yu Zhang, Chung-Cheng Chiu, Barret Zoph, Ekin Cubuk, and Quoc Le. Specaugment: A simple data augmentation method for automatic speech recognition. pages 2613–2617, 09 2019.

[18] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[19] Yuan Gong, Yu-An Chung, and James Glass. Ast: Audio spectrogram transformer. *arXiv preprint arXiv:2104.01778*, 2021.

[20] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina N Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. 2018.

[22] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[23] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems*, 32, 2019.

[24] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

[25] Alexander Kolesnikov, Alexey Dosovitskiy, Dirk Weissenborn, Georg Heigold, Jakob Uszkoreit, Lucas Beyer, Matthias Minderer, Mostafa Dehghani, Neil Houlsby, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. 2021.