UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DEPARTMENT OF INFORMATION ENGINEERING

MASTER'S DEGREE IN COMPUTER ENGINEERING

IMPLEMENTATION OF AN AI-BASED MODEL FOR DETECTING SPEED VARIATIONS BY MEANS OF SPECTROGRAM IMAGES FOR NEW MAGNETIC TAPES PRESERVATION STRATEGIES

Supervisor: Prof. Sergio Canazza Targon

co-Supervisors: dr. Alessandro Russo, dr. Matteo Spanio

Graduand: Lorenzo Lunardon

ACADEMIC YEAR 2023-2024

Abstract

Old audio storing methods such as magnetic tapes have been obsolete for decades, now more than ever with the advent of digital formats. The material stored on those tapes, though, is still precious today as it was when it was recorded, and it is at risk of being lost to time, as the materials degrade. The obvious solution to this problem is to digitize these documents, which is a tedious and error-prone process for any technician.

This study focuses on preventing one of those errors, which is reproducing the tape using a playback speed different from the recording speed, leading to a very inaccurate signal and a spoilage of the heritage associated to the original document.

Instead of analyzing the audio signal directly in the time domain, we opted to explore the frequency domain, computing the spectrograms of the tapes and analyzing them using convolutional neural networks. The approach gave promising results, and the research showed that this is a valid way, worth exploring further.

Acknowledgements

First and foremost, I would like to thank my family for allowing me to take on this strenuous academic path and being by my side for all bumps in the road. I don't know if I would have reached the ending of it without their unrelenting support and belief in me, which I myself lacked many times.

I also want to express my gratitude to all the friends who have made this whole journey much more pleasurable, from the ones I've met in my hometown as a kid and have been with me all along, to the ones I've met in Erasmus or some random hostel in the middle of nowhere. I am thankful for the laughs we had together and wish for many more to come.

Last but not least, I would like to thank professor Sergio Canazza and Matteo and Alessandro at CSC for helping me write this thesis and letting me contribute to their amazing work.

Again, thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The rapid growth of the world wide web in the past decades has completely changed the way we listen to music. Almost a hundred years ago the latest invention in audio recording was the german Magnetophon, a huge reel-to-reel tape recorder that used magnetic tapes to store soundwaves. Then cassette tapes and vinyls became the norm, and the compact, more portable design was far more successful, making musical records appear in every household.

In the 90s, the advent of digital media opened a whole new world of possibilities, with the mediums becoming smaller and smaller while their capacities only got bigger.

Today, we don't even have to own a copy of the record we want to listen to. As streaming services have become the standard, a good internet connection is sufficient to get access to pretty much every composition known to man. The thought that one day we won't be able to hear our favourite record never crosses our mind, as digital media doesn't fear the test of time. It hasn't always been this way, though.

Active and Passive Preservation

In fact, older formats like open reel tapes are vulnerable to the degradation of the medium over time [1], making it challenging to preserve the content inside them. This act of preserving the audio memory can be done in two ways [2]: passively, which consists in storing the documents in the most optimal environment possible, slowing the degradation as long as we can. The second option is active preservation, which involves migrating data onto new media.

Passive preservation only helps in delaying the inevitable, so the only long-term solution is turning all the old archives that risk getting wiped into digital archives. This would have the added benefit of making the access to the documents easier to the general public, as it takes away the need for format-specific hardware. As we're going to see, though, the data transfer from a magnetic tape to a digital format is subject to various types of errors, namely electronic, procedural and operative.

1.1 Centro di Sonologia Computazionale

The task of preserving the cultural heritage associated with endangered audio documents has been one of the main objectives at Centro di Sonologia Computazionale (CSC), the Sound and Music Computing Laboratory at the Department of Information Engineering of the University of Padova.

For over two decades, the laboratory has been developing a framework for both active preservation of historical sound material, in particular stored on magnetic tapes, and enabling better access to it. Their achievements in the field sparked very important collaborations with prominent European collections and archives such as the Paul-Sacher-Stiftung in Basel, the Fondazione Arena di Verona, the Luigi Nono Arhive in Venice, and the Historical Archive of the Teatro Regio di Parma. [3]

Methodology at CSC

The methodology for actively preserving the tapes involves, as said previously, the digitisation of the material, but it's not limited to that. Additional information such as noise signals that characterize the recording system, signs of damage and alterations on the carrier, notes on the container and other metadata must also be preserved.

The process followed at CSC is composed of these steps:

1. Digitise the analog magnetic tapes into digital audio files, using lossless formats and sufficient sample rates and bit resolutions[1].
2. Record the playback head of the tape recorder, obtaining a high-resolution video.
3. Listen back to the recording and note down any anomaly or irregularity in the audio signal. Some irregularities are caused by damage on the tape, while others may have been caused by a wrong configuration of the tape recorder, such as setting a wrong equalization curve or an incorrect playback speed.
4. Watch the footage, locate and classify the irregularities found at the previous step.
5. Collect all the metadata related to the tape, like photos of the damage on the tape or annotations on the casing. An example is in figure 1.1.

The complete package of data and metadata is called a *preservation copy*.

The steps regarding finding and classifying the irregularities in the recording and the footage have always been done by hand by technicians, but the latest advances in the fields of artificial intelligence suggest that the process could be automated, in order to aid professionals in the tedious task of detecting errors, thus saving a lot of time.

[1] Generally, 96 kHz for sample rate and 24-bit resolution are sufficient

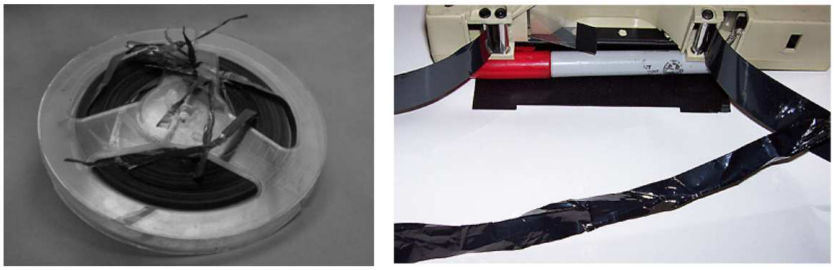


Figure 1.1: Damaged and corrupted magnetic tapes, taken from [4]

Previous works at the University of Padova [4][5] have experimented applying machine learning computer vision techniques to analyze the video footage and detect physical irregularities like splices, signs and various corruptions on the tape's surface, with promising results. The audio part, though, is still mostly unexplored. This thesis will address one of the numerous problems that can occur in the digital copy of a magnetic tape. Before diving in, though, it's better to define where this work falls in the grander scheme of things.

1.2 MPAI

Artificial intelligence has been a hot topic for the past two decades, and its reach is only getting bigger by the day. It has entered our daily life with interactive chatbots and voice assistants, improved computer vision applications in all kinds of fields, from medical diagnosis to self-driving vehicles, and it also has been adopted to predict the trends of financial markets.

All this variety in applications has made it so that every software project adopts its own methodology, creating a multitude of frameworks that don't share any interoperability between each other. If there was a common standard that governed the structure of all these projects, it would improve the development process in terms of time, money and user experience.

This lack of orchestration in the development of AI-based software products is the main concern for the MPAI - *Moving Picture, Audio and Data Coding by Artificial Intelligence* - project. Its goal is to create a set of standards for applications and services that use AI at their core. Many areas are involved: autonomous vehicles, AI for health data, context-based audio enhancement and more.

AI Workflows and modules

The MPAI standards have a flexible and modular structure, and are based on frameworks denoted AIF (AI-Framework) which are built upon building blocks called AIM (AI-Modules), each performing a specific task. The AIF orchestrates these modules in order to perform more complex operations, and its input-output pipeline is defined by the MPAI project, like can be seen in figure 1.2. It's noteworthy that the

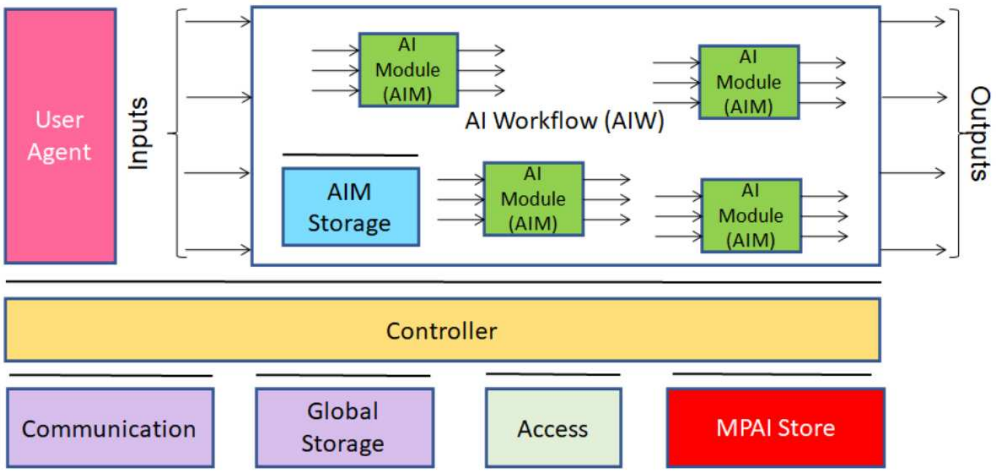


Figure 1.2: MPAI AI Framework

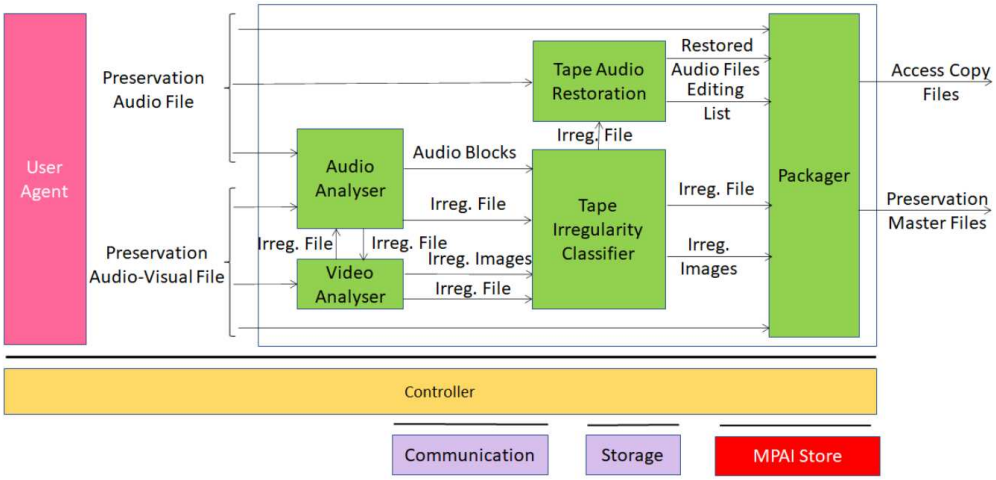


Figure 1.3: ARP Framework

standard only what goes in and out of the pipeline, but what happens in between with the modules is chosen by the developer.

As said before, MPAI standards cover several application and use cases. The active preservation of magnetic tapes falls under the Context-Based Audio Enhancement (MPAI-CAE) use case, more specifically the Audio Recording Preservation (ARP). Figure 1.3 describes the pipeline of the ARP framework, where all the modules can be seen.

The framework takes in input the digitized audio of the magnetic tape and the video recording of the playback head of the recorder during the reproduction. After processing, it produces in output preservation master files (a copy of the input) and access copy files, which contain the final restored version of the audio.

This thesis will describe an implementation of the Audio Analyser module using artificial neural networks on spectrogram images extracted from the digitized tapes. Among all the problems that can arise during the digitization process, this work will aid in detecting irregularities between the original recording speed and the playback speed used during reproduction. The next chapter will showcase the problem further, then it will discuss a possible solution.

Chapter 2

Problem and Solution

Magnetic tapes can be recorded over different tape speeds: generally, the higher the speed, the better the reproduction quality, but the more quantity of tape it requires to record the same duration of audio. Hence, slower speeds help to conserve tape, and are primarily used when audio quality is not the primary concern. However, this variety comes with some possible technical problems, as explained next.

2.1 Irregularities in Playback and Recording Speed

During the digitization process, the tape recorder must be configured correctly, because if the document gets reproduced at a speed which is different than the recording speed, the digitized audio gets sped up or down and the quality plummets, rendering conservation efforts virtually useless. Since the digitization is done for hours and hours of audio material, it is not unusual to make this sort of mistake, so we want to have a way of preventing it automatically via software.

Intuitively, if a digital sample gets sped up, then all the audio signal's frequencies will get transposed up in the frequency spectrum. More precisely, doubling the reproduction speed will result in a transposition up one octave, quadrupling up two octaves, and so on. Similarly for the inverse direction, slowing down a sample will result in a transposition down the register.

This information suggests that the way to tell if a magnetic tape is being reproduced incorrectly is looking into the frequency domain. But how is it possible to discern between a violin playing a melodic line in the third octave, and another violin playing the same melodic line in the second octave, and then being doubled in speed?

Obviously, a person could easily which is which, as the sped up version sounds very unnatural and grating to the ear. Automating this task with software, though, is not as straightforward.

2.2 A Possible Solution

Since the goal is to be able to classify between regular and irregular samples, the first thing that comes to mind is **machine learning**, in particular **classifiers**. A lot of variables are at play: what kind of classifier to use, what features to extract, what data to use for the training and testing, just to name a few.

A possible path to follow would be to extract numerical features from a Fourier analysis of the audio signal and then create a dataset (labeled or unlabeled) which is used to train a model, either with supervised learning techniques (decision trees, logistic regression, etc) or unsupervised techniques like clustering. A list of possibilities regarding what features to use can be found in Rodà et al.[6], in the frequency domain analysis section.

It is also possible to represent signals in visual ways, for example using spectrograms, to which we can apply computer vision machine learning techniques [7, 8], using the images' pixels as features. In recent years deep learning has been used extensively in computer vision, most notably through the use of **convolutional neural networks**. These networks automatically extract features from images using a series of filters, and use those features to make predictions.

The goal of this thesis is to explore a possible approach: after digitizing the tape, convert the audio to a spectrogram image and analyze it using convolutional neural networks, which detect the sections in which the playback speed is incorrect, if there are any. The timestamps of the irregularities are then written on a file and processed accordingly. The computation of the spectrograms and the structure of the networks are going to be presented next.

2.2.1 Spectrograms

A spectrogram can be defined as an intensity plot of the discrete **Short-Time Fourier Transform** (STFT) magnitude. The discrete STFT is simply a sequence of DFTs of windowed data segments, where the windows are usually allowed to slightly overlap in time.

$$STFT(x[n])(m,\omega) = X(m,\omega) = \sum_{n=-\infty}^{\infty} x[n]w[n-m]e^{-i\omega n}$$

where x is the original signal to be transformed and w is a window function centered around zero. In this case, m is discrete while the frequency ω is continuous, but in practical application the STFT is computed using the fast-fourier-transform algorithm, so both variables are discrete and quantized.

It is a very important representation of audio because human hearing is based on a kind of real-time spectrogram encoded by the cochlea of the inner ear [9] [10]. The spectrogram has been used extensively in the field of computer music as a guide during the development of sound synthesis algorithms.

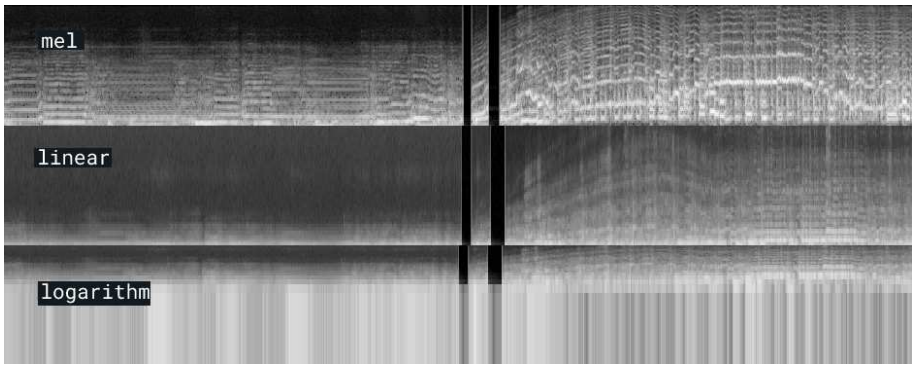


Figure 2.1: Spectrograms of the same sample with three different scales

In practice, the y-axis of the plot represents the frequency, the x-axis the time and the z-axis (the color) represents the intensity of the corresponding frequency at that timestamp. The y-axis can use a variety of different scales, and for this thesis the possible options considered are:

- **Linear Scale:** Every frequency interval occupies the same amount of space on the y-axis. This results in the higher octaves occupying more space on the plot.
- **Logarithm scale:** The frequencies are usually measured in decibels, where every octave occupies the same amount of space on the y-axis.
- **Mel scale:** A different kind of logarithm scale is used on the y-axis:

$$mel(f) = \begin{cases} f & \text{if } f \leq 1 \text{ kHZ} \\ 2595 log_{10}(1 + \frac{f}{700}) & \text{if } f > 1 \text{ kHz} \end{cases}$$

  For reference, on the mel scale 1000Hz match 1000mel.

The difference in the result using the three different scales can be seen at figure 2.1, where the frequencies were capped at a maximum of 20 kHz for simplicity and the intensity is colored in grayscale. The difference in the space given to the lower octaves is especially great in the linear and logarithm spectrograms. The question to be answered is: what scale is more appropriate to detect irregularities? This will be answered in the later chapters, in section 4.4.

When choosing what technologies to use for computing the spectrograms, there are numerous options available, from simpler command-line interface tools to more complex GUI programs like digital audio workstations. Since the goal is to create a dataset of thousands of images, the task has to be automated, which is easier to do with a CLI program.

For computing the linear and logarithm spectrograms the choice fell on **SoX** [11], a great command-line tool written in C that, among several other functions, allows

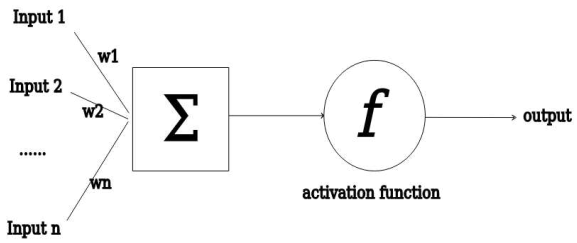


Figure 2.2: Model for one neuron

to compute spectrograms from audio files [1]. SoX unfortunately doesn't support mel spectrograms, for which the Python module **Librosa** [12] was chosen. [2]

2.2.2 Convolutional Neural Networks

Convolutional Networks, or CNNs, are a specialized type of artificial neural network used for processing data which has a known, grid-like topology, like images. These models have been tremendously successful in practical applications in different fields, from face recognition to medical image analysis.

The structure of a CNN is the usual neural network structure: there's an input layer, composed by the pixels of the input image, then one or more hidden layers, and an output layer at the end. Each layer is composed of neurons, each one of them having some inputs, an output and an activation function. Neurons are connected with each other with weighted links, each link having a weight w that measures how important the link is to that neuron, like represented in figure 2.2.

During the training phase the network updates these weights at every iteration. The most basic method to find the optimal weights is the gradient descent method, or its stochastic variant.

After having propagated the input through all the layers, the network produces an output, with which it makes a prediction on the input. This output is then fed to a loss function L, which quantifies how wrong the predicted output is with respect to the desired result[3]. Afterwards, the error is adjusted by modifying the weights w using backpropagation, moving in the opposite direction of the gradient of L using the chain rule. The gradient indicates the direction in which a function grows more rapidly, so moving in the opposite direction reduces the error.

The name "convolutional network" implies that it makes use of a mathematical operation called **convolution**, which is a specialized kind of linear operation. Concisely, *convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers* [13].

[1] SoX - Sound eXchange
[2] Librosa Docs
[3] A label in the classification task, or a real number in a regression task

Convolution

In one dimension and in in the countinuous domain, the convolution operation between two functions x and w is defined as follows [13]:

$$s(t) = \int x(a)w(t-a)da$$

and it is typically denoted with an asterisk

$$s(t) = (x * w)(t)$$

In convolutional neural networks terminology, the first argument, in our case x, is referred to as the **input**, while the second argument as the **kernel**. The output is sometimes called the **feature map**.

If we assume that the input and the kernel are two-dimensional and discrete, then the convolution takes the form:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

where I is the two-dimensional input (an image in our case) and K is a two-dimensional kernel. An example of 2-D convolution can be seen in figure 2.3.

The convolution process is inspired by the biological process for visual analysis in living organisms. The layer of neurons responsible for convolution divides the image in several juxtaposed fragments, that are analysed to extract patterns that are sent to the subsequent layers as feature maps.

The kernels (also called filters) are matrices, which are much smaller in dimensions than the input images, and they contain numerical values. The convolution between the input and the kernel produces some activation maps, regions in which some features were found. The numerical values in the kernels change at every iteration on the training set, because the network is learning how to recognise the features. Furthermore, in order to reduce the dimensionality of the convolution's output and to avoid overfitting, CNNs use another type of layer, the **pooling layer**.

Pooling

A pooling operation replaces the output of the network at a certain location with a summary statistic of the nearby outputs. For example, the **max pooling** operation calculates the maximum output within a rectangular neighborhood. There are other options, such as average pooling or a weighted average based on the distance from the central pixel. An example of max and average pooling can be seen in figure 2.4.

Regardless of the choice of function, pooling helps in making the representation approximately invariant to small translations of the input image. This essentially

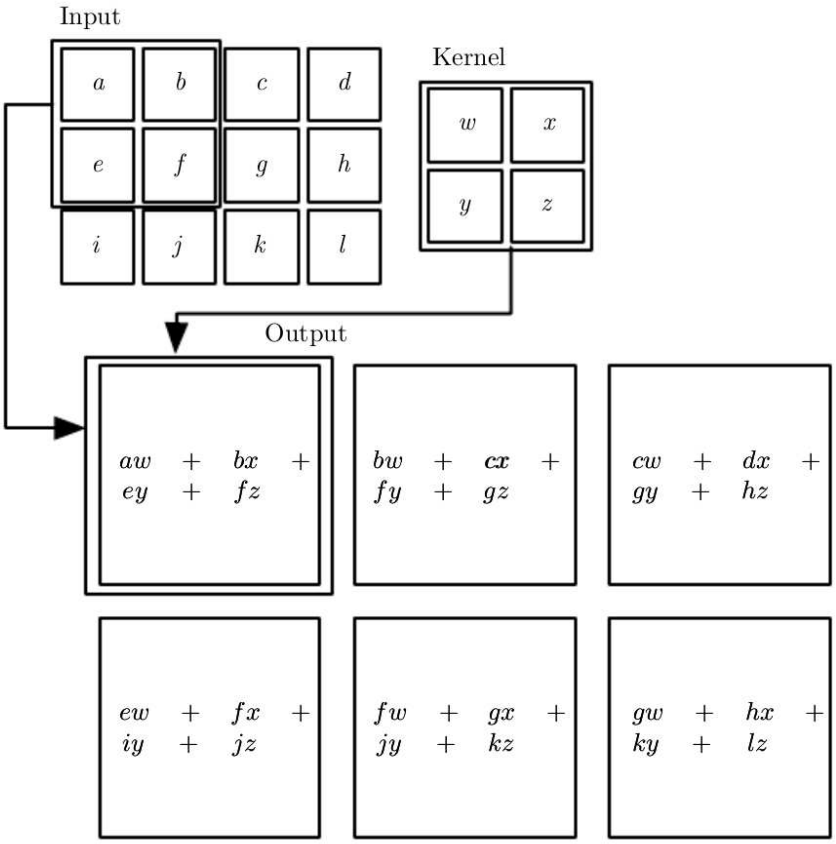


Figure 2.3: Example of 2-D convolution, taken from [13], page 334

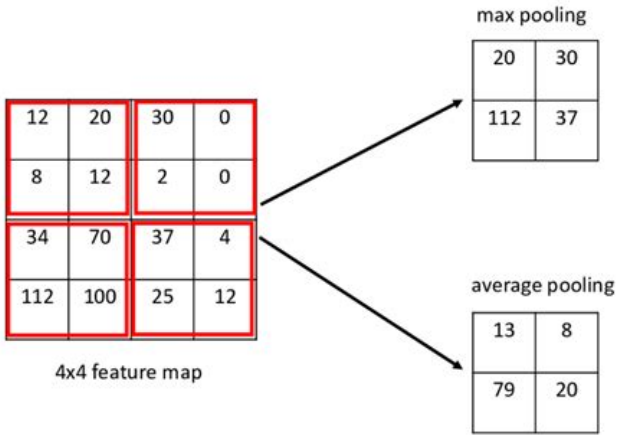


Figure 2.4: Example of max and average pooling, taken from [14]

means that if we translate the input by a small amount, the pooled outputs mostly don't change. This property helps if we care more about whether a certain feature is present in the image, and not the location of said feature in the image.

These layers are almost always put after a convolutional layer, in order to reduce the dimensionality of the input to pass to the next layers.

Activation Function

The most basic activation function which can be used in neural networks is the **sigmoid**:

$$f(v_k) = \frac{1}{1 + e^{v_k}}$$

where v_k is the input at the k-th neuron.

In deep networks with many hidden layers, though, this activation function becomes problematic. During the propagation phase, in fact, the weights get updated proportionally to the partial derivative of the function with respect to the current weight at every iteration. The derivative of the sigmoid function is typically less than 1, so the chain rule causes the multiplication of many terms which are less than 1, resulting in the degradation of the gradient in the layers far from the output. The sigmoid has another problem: is tends to saturate towards the extremes (0 and 1), which further contributes in the degradation of the gradient.

To avoid the problem another activation function is often used, called **rectified linear unit** usually referred to as ReLu:

$$f(v_k) = max(0, v_k)$$

Such function saturates only when the input is less than 0, and it is also more computationally efficient than the sigmoid.

Still, the sigmoid activation function is useful for binary classification when used in the last layer, as the output can be converted to the label. Similarly, for classification with more than two classes another function can be used, called **softmax**, which is used to predict the probabilities associated with a multinoulli distribution. It is defined as:

$$f(\mathbf{v})_k = \frac{exp(v_k)}{\sum_{j=1}^{n} exp(v_j)}$$

Both the sigmoid and the softmax functions return a set of values between 0 and 1, one for each class, representing the probability that the input belongs to the corresponding class. To produce the predicted label, just pick the highest one like this:

$$\text{Predicted Label} = argmax(\mathbf{v})$$

where $\mathbf{v}$ is the output of one of the two activation functions.

Loss Functions

There are several choices when it comes to loss functions, but for this work the focus will be on **cross-entropy**, which is widely used when optimizing classification models.

In a supervised training task, each labeled sample has a known class label with probability 1 and probability 0 for all other labels. A neural network takes in input a sample and outputs the probability of an example belonging to each class label. Cross entropy can then be used to calculate the difference between the two probability distributions.

In classification, there are the expected and predicted probabily distributions:

- Expected Probability: the known probability of each class label for an example in the dataset, call it P
- Predicted Probability: the probability of each class label predicted by the model, call it Q

So, we can estimate the cross-entropy for a single prediction as follows:

$$H(P, Q) = -\sum_{x \in X} P(x) log(Q(x))$$

where each $x \in X$ is a label that could be assigned to the sample, and $P(x)$ will be 1 for the known label and 0 for all other labels. When there are only two possible classes, it's called **binary cross-entropy**, while with more than two classes **categorical cross-entropy**.

All in all, an example of a structure of a convolutional neural network can be found in figure 2.5, where the convolution and pooling layers are followed by dense layers right before the output layer, as it's common practice.

Optimizers

It was previously stated that during the backpropagation phase, the algorithm moves in the opposite direction than that of the gradient (for minimization problems). At every step the movement is done with a certain step called *learning rate*, which is a hyperparameter. A learning rate which is too small results in a very slow training process that can often get stuck. On the other hand, taking too big of a step can make the algorithm zig-zag around the optimal solution without ever reaching it.

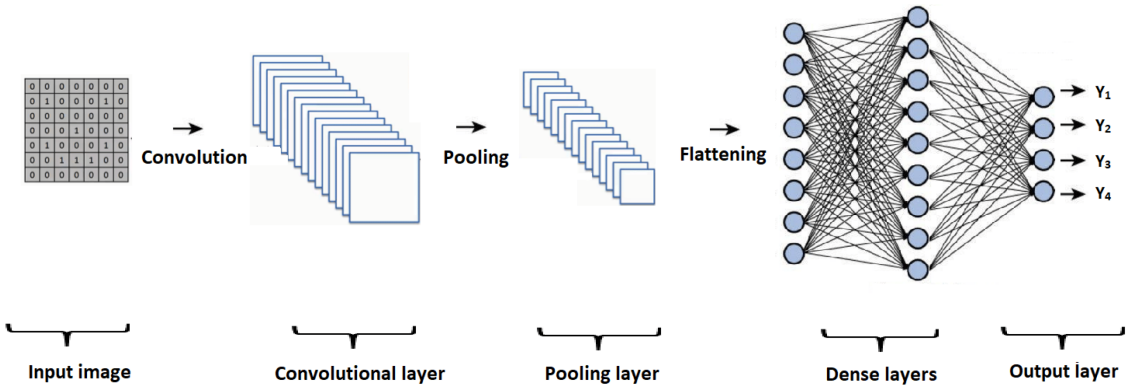


Figure 2.5: Example of structure of a CNN

There are several extensions of the stochastic gradient descent method that take care of this problem by adopting an adaptive learning rate strategy, using momentum and gradient forgetting factors. One of these implementations is called **adam**, invented in 2014, which will be used in the training phase of the classifiers.

This optimizer doesn't guarantee the convergence to the optimal point for all convex objective functions, but it's widely used for its strong performance in practice.

Regularization

Neural networks trained on relatively small datasets can overfit on the training data. There are many adjustments that can be made to combat this, in a process called *regularization*. This work will use **dropout** as the main method for decreasing overfitting.

It consists in ignoring, or "dropping out" a certain number of layer outputs with a certain probability. This is done on all or some of the hidden layers of the network, and it has the effect of making the training process noisy, forcing nodes within a layer to probabilistically take on more or less responsibility for the inputs.

Dropout simulates a sparse activation from a given layer, which interestingly, in turn, encourages the network to actually learn a sparse representation [15] as a side-effect.

Technologies Used

The chosen programming language for this work is **Python**, as it is by far the most common in the machine learning and data analysis worlds, and it's pretty much become the industry standard. Its ease of use, clear syntax and huge number of pre-written modules make it really easy to develop projects quickly and effectively. Its data visualization libraries are also ubiquitous in data science.

For building and training neural networks, the two most prominent Python modules are **PyTorch**, originally developed at Meta, and **TensorFlow**, developed at Google.

The choice fell on TensorFlow, because of the ease of use given by the open-source library **Keras** which provides a high-level, user-friendly Python interface for quickly building and testing machine learning models.

Keras allows users to build complex neural networks modularly, just by adding the chosen layers, loss, activation functions and optimizers as pre-built blocks, and it also allows to generate and split datasets from a folder. At the end of the training, the other modules that are going to be used are **matplotlib** and **scikit-learn**, which will be dedicated to the visualization of the results and to measure the performance of the models.

Approach: Two Classifiers

The approach is pretty simple: take several audio samples of digitised magnetic tapes, with some samples being reproduced at the same speed that they've been recorded with, and the others being reproduced incorrectly. Label the first "correct" and the others "wrong". Now compute the spectrograms for all of them, and extract thousands of fixed-size windows from the spectrograms in order to make a binary dataset.

Then, the subfolder labeled "wrong" can be divided further to make a multi-class dataset, with every class being the speed-up or slow-down factor of the reproduction speed with respect to the recording speed. For example, a tape recorded at 7.5ips and reproduced at 15ips will be labeled "double", while if it was reproduced at 3.75ips it would be labeled "half".

The most used recording speeds for magnetic tapes are 3.75ips, 7.5ips and 15ips, so these will be the ones used for the thesis. All in all, there are four possible classes:

Recording Speed	Playback Speed	Label
3.75ips	7.5ips	double
7.5ips	15ips	double
7.5ips	3.75ips	half
15ips	7.5ips	half
3.75ips	15ips	quadruple
15ips	3.75ips	quarter

Table 2.1: Labels for multi-class classifier

Now there are two datasets, which can be used to train two classifiers: a binary one, which guesses if a spectrogram image is irregular, and a multi-class one, which quantifies the type of irregularity of an irregular image.

After having trained the models, we can use them to analyze new audio samples in a similar way: take in input an audio sample, compute its spectrum and divide it into fixed-size segments, then finally input the segments into the classifiers and extract the timestamp of found irregularities.

Chapter 3

Dataset Creation

The processing, preparation and cleaning of data is arguably the most important part of any machine learning application, as well as the most tedious and lengthy. Usually datasets are huge tables filled with numerical and categorical values that need to be filtered and corrected, but since we're working with CNNs it's only images that will be dealt with. The main goal is having a set of images of the same size, all properly labeled and ready to be fed to a model.

This chapter describes the nature of the data handled by the software, as well as the manipulation that was done on it in order to generate an adequate dataset for training the classifiers. The whole codebase was uploaded on GitHub, and the repository can be accessed at this link.

3.1 Audio Dataset

The data on which the models were trained was made in the laboratory specifically for this study: it consists of 300 audio samples of magnetic tapes being reproduced using a reel-to-reel tape recorder and then digitised in WAV format, adding up to around 90 minutes of material.

The samples include a wide variety of audio content, spanning from spoken word, classical and opera music to more eclectic genres such as noise and electronic music, using completely different instrumentation. This was done in order to expose the models to a broad variety of possibilities, and make their generalization capabilities better.

Each sample is divided in two phases: firstly, the tape is reproduced at the same rate at which it was recorded, resulting in a "regular" phase. A switch then occurs, and the tape gets reproduced at a speed different from the original one, resulting either in a sped-up or slowed-down version of the audio sample. This change introduces a ton of audio artifacts, which ideally the binary classifier will pick up on and label as "irregular".

The possible recording-playback rates considered in this study are 3.75ips [1], 7.5ips and 15ips. These three are by no means the only possibilities, but they are the most common ones used in recordings. All the six possible combinations of these speeds were included in the original dataset, which is summarised in table 3.1.

It is notable that labels "half" and "double" are assigned quite more samples than the other two, making the dataset slightly unbalanced. It will be more clear after the training results if this skew causes problems.

Recording Speed	Playback Speed	Label	Samples	Time
3.75ips	7.5ips	Double	50	882s
7.5ips	15ips	Double	50	787s
7.5ips	3.75ips	Half	50	766s
15ips	7.5ips	Half	50	786s
3.75ips	15ips	Quadruple	50	1,105s
15ips	3.75ips	Quarter	50	1,117s
		Total	300	5,445s

Table 3.1: Summary of the audio dataset before preprocessing

3.2 Spectrogram Dataset

Convolutional Neural Networks work on fixed-size images: in this case, the images are excerpts of the spectrograms computed from the audio samples. So, in order to feed the CNNs the training data, the original dataset needs to be processed accordingly. The pipeline is summarised in the steps below, which can be recreated following the instructions written on the GitHub repository's `README.md` file.

- **Channel Separation:** The samples were digitised in stereo format, and while the two channels sound the same to the ear, they look slightly different in the frequency domain. Thus, in order to enrich the dataset, the channels were separated, creating a new collection of 600 samples.
- **Spectral Extraction:** Compute the spectrogram of each sample: each image has fixed height and width proportional to the audio duration[2]. An example can be seen at figure 3.1. While the difference between the two channels is not that notable, it can still be seen right after the black band, where in the right channel the glissando is slightly more prominent. Another example can be seen in the close-up at figure 3.2, taken from another sample halving in playback speed.

  In the repository, the script that performs this step is `src/wav2spec.py`. The images are saved in 8-bit grayscale format, so to minimize the storage space

[1] ips stands for "inches per second", which is the speed at which the tape rotates during the recording-playback process

[2] In this case the height is fixed to 128 pixels, while the width has a resolution of 256pixels/s

needed while carrying the same amount of information.

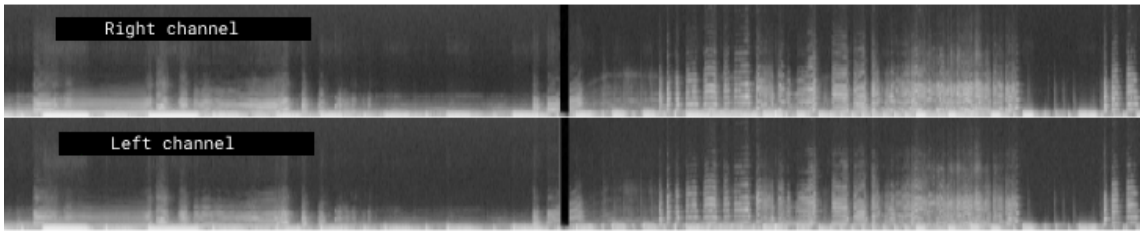


Figure 3.1: Spectrograms of the same sample doubling in playback speed, using a linear scale on the y-axis. The black band corresponding to the speed change is clearly visible

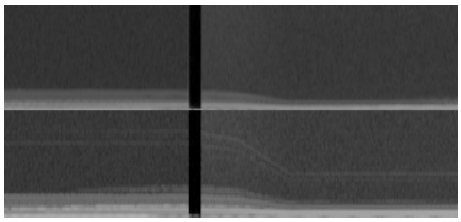


Figure 3.2: Close-up of glissando region, different from channel to channel

- **Spectrogram Division:** Each image has to be divided in two parts, but the timestamp at which the switch occurs is not known a-priori. Some image manipulations can be used to locate the black band in the spectrograms and use its position to accomplish the goal. This task is assigned to script `src/divide.py` in the repository. The script uses the OpenCV module to apply a thresholding function and a morphological filter to isolate the region of interest and store the relative x-coordinate. It then uses the coordinate to cut the images in two and save the two halves separately.
- **Spectrogram Segmentation:** Extract multiple fixed-width segments from each half: segments coming from the left of the band will be labeled "c" for correct, while the others will be labeled "w" for wrong. This step is done by script `src/segment.py`.

  In this work segments were taken with a step of 128 pixels, starting with an offset of 128 pixels to the right of the black band in order to skip the "glissando" region, which in normal cases doesn't occur and could lead to confusion during the training of the classifiers.

  At this point of the process, it's still not clear how wide every segment should be: theoretically, if the segments are thin then a spectrogram can be analyzed more precisely, so is would seem the preferable choice. But since smaller images carry less information, this could cause poor classification performances. After trying out widths of 64px, 128px and 256px, the choice fell on the latter, as it resulted in better performance. The reasoning goes more in depth in the next chapter, where the training results are discussed.

These steps have been recreated for all three possible scales, thus creating three different datasets. It has to be noted that the speed change doesn't always occur near the middle of the sample, but rather it usually tends to happen in the latter half of it, making the dataset unbalanced, as seen in table 3.2.

Label	Samples
correct	12,222
wrong	6,429
Total	18,651

Table 3.2: Summary of the binary dataset for the first classifier

The subfolder labeled "w" can then be subdivided to make another dataset, which will be used to train the second classifier. Depending on the filename, samples are assigned four possible labels, which represent the speed change factor. The second dataset is summarised in table 3.3

Recording Speed	Playback Speed	Label	Samples
3.75ips	7.5ips	double	956
7.5ips	15ips	double	1,094
7.5ips	3.75ips	half	677
15ips	7.5ips	half	1,295
3.75ips	15ips	quadruple	1,041
15ips	3.75ips	quarter	1366
		Total	6,429

Table 3.3: Summary of the dataset for the second classifier

Considerations

Again, the dataset is not completely balanced, as the labels "double" and "half" are assigned more samples than the two other classes. A possible solution would be applying some light *random oversampling* [16] on the "quadruple" and "quarter" classes in order to level out the label counts, which has proven to a robust strategy.

It's worth noting that no *data augmentation* was done on the initial data, which is surely a possibility to explore in the future if the performance is not satisfactory and if overfitting problems arise. Some options would be to digitise the same tapes again but with slightly different configurations during the reproduction phase, for example adjusting the gain of introducing some kind of noise signal.

The spectrograms images were kept as they were after being computed, and no further preprocessing functions were applied to them using OpenCV or other software. It's possible that some transformations like increasing the contrast or doing some thresholding could highlight some additional information inside the spectrum and

help the classifiers learn more optimally. However, for the sake of simplicity and the already high number of variables at play, the decision was to avoid doing any more processing.

Chapter 4

Model Training

In the previous chapter it was mentioned how it wasn't clear what shape the input should have. This will be the first thing touched upon in this next part, along with the quality measures used to choose the right size.

After settling for one size will come the training of all six classifiers (two for each of the three scales) and then the comparison between the quality metrics.

The training process on thousands of images is computationally heavy, so I used Google Colab to make use of powerful nVidia T4 GPUs which are available for free. The training process can be replicated with the notebook `model_fit_colab.ipynb`, which is suited for running in Google Colab[1].

4.1 Network Configuration

The configuration for the CNNs was chosen through trial and error, as there is no specific rule to follow to increase accuracy and avoid overfitting. The networks for both classifiers have pretty much the same structure, apart from the last layer which is responsible for outputting the predicted label:

- Three **convolutional layers** with kernel size 7x7, **relu** activation function and padding set to "same" The first layer has 8 neurons, the second 16 and the third 32.
- Each convolutional layer is followed by a **max pooling** layer with kernel size 5x5.
- One **global average pooling** layer.
- One **dense** layer of size 32 with **relu** activation function.
- One **dropout** layer with probability 0.3.

[1]It is suggested to save the archived datasets into Google Drive, to not have to manually upload them to Colab each time an experiment has to be run.

- One final **dense** layer:
  - Binary classifier: 2 neurons, **sigmoid** activation function.
  - Multi-class classifier: 4 neurons, **softmax** activation function.
- Optimizer: **adam** for both classifiers.
- Loss Function:
  - Binary classifier: **Binary Crossentropy**.
  - Multi-class classifier: **Categorical Crossentropy**.

Overall, the binary classifier has 32,930 total parameters and the multi-class one has 32,996.

All the models were trained for 30 epochs using a batch size of 20 images. The training set accounted for 70% of the dataset, the validation set for 20% and the testing set for the remaining 10%.

4.2 Quality Measures

Accuracy could be used as a measure of a classifier's performance, but sometimes this metric can be misleading, especially since the dataset is not balanced. The chosen metrics are then **precision**, **recall** and **f1-score**. The formulas to compute these sligthly vary in the binary and multi-class case.

In the binary case the confusion matrix looks like in table 4.1, while with more than two classes like in table 4.2. In the tables the labels are numerical, but they can easily be translated to our case.

		Predicted	
		0	1
Actual	0	True Negative	False Positive
	1	False Negative	True Positive

Table 4.1: Example of confusion matrix with two classes

		Predicted			
		0	1	2	3
Actual	0	TN	TN	FP	TN
	1	TN	TN	FP	TN
	2	FN	FN	**TP**	FN
	3	TN	TN	FP	TN

Table 4.2: Example of confusion matrix with four classes, in this case to compute metrics for class 2

- **Precision:** we denote as precision the fraction of positive detections reported by the model that were correct. In formula,

$$P = \frac{TruePositive}{TruePositive + FalsePositive}$$

- **Recall:** the fraction of positive events that were rightfully detected

$$R = \frac{TruePositive}{TruePositive + FalseNegative}$$

- **F1-score:** can be interpreted as the harmonic mean of precision and recall, where the best possible score is 1 and the worst is 0.

$$F1 = \frac{2PR}{P + R}$$

These metrics can be computed for every class, but it's also possible to compute the averages for all classes to summarise the model's quality with one global metric. These are called **macro averages** for the unweighted mean and **weighted averages** for the weighted mean.

4.3 Comparing Input Sizes

The height of the spectrograms was arbitrarily fixed at 128 pixels, and three possibilities were considered for the widths: 64, 128 and 256 pixels. The next part compares the metrics achieved at the end of the training processes, using the linear scale on the y-axis. The function `sklearn.metrics.classification_report` was used to calculate all the metrics.

Label	Precision	Recall	F1
c	0.77	0.95	0.85
w	0.84	0.51	0.63
macro avg	0.81	0.73	0.74
weight avg	0.80	0.79	0.77
accuracy	0.79		

Table 4.3: Metrics of the binary classifier with 64 pixels width, linear scale

Label	Precision	Recall	F1
double	0.90	0.38	0.54
half	0.78	0.04	0.07
quadruple	0.75	0.69	0.72
quarter	0.29	0.98	0.45
macro avg	0.68	0.52	0.44
weight avg	0.71	0.45	0.40
accuracy	0.45		

Table 4.4: Metrics of the multi-class classifier with 64 pixels width, linear scale

Tables 4.3 and 4.4 contain the results for the width of 64 pixels. The first table is the summary for the binary model, while the second table is the summary for the multi-class model. As can be seen, the models perform badly, especially the multi-class one, which achieved a weighted recall of only 0.45 and a weighted f1

score of just 0.40, making it safe to say that this width is not viable. In addition to the quality metrics, the confusion matrix that the model achieved on the test set at figure 4.1 is clearly a mess, since almost all "half" labels were predicted incorrectly. In fact, the accuracy on the test set came at just 44.9% for the multi-class classifier, and 78.6% for the binary one, which is not a satisfying result.

Label	Precision	Recall	F1
c	0.93	0.97	0.95
w	0.95	0.87	0.91
macro avg	0.94	0.92	0.93
weight avg	0.94	0.94	0.94
accuracy	0.94		

Table 4.5: Metrics of the binary classifier with 128 pixels width, linear scale

Label	Precision	Recall	F1
double	0.96	0.95	0.96
half	0.98	0.98	0.98
quadruple	0.92	0.95	0.94
quarter	0.99	0.98	0.99
macro avg	0.96	0.97	0.97
weight avg	0.97	0.97	0.97
accuracy	0.97		

Table 4.6: Metrics of the multi-class classifier with 128 pixels width, linear scale

The results for the width of 128 pixels are in tables 4.5 and 4.6. Doubling the input width led to a great performance increase. The binary model has every metric over 0.9 except for the recall of the "wrong" class, which is 0.87. The multi-class model performed even better, with every average being over 0.97. This is most certainly a viable option, having a balanced combination of good performance and small size for the images, reducing training time and computation loads. The confusion matrix at figure 4.2 also looks much better, with the majority of the elements lying on the main diagonal, as it ideally should be. The test accuracy went from 44.9% to 96.8%, a night and day difference.

Label	Precision	Recall	F1
c	0.98	0.96	0.97
w	0.92	0.95	0.94
macro avg	0.95	0.96	0.95
weight avg	0.96	0.96	0.96
accuracy	0.96		

Table 4.7: Metrics of the binary classifier with 256 pixels width, linear scale

Label	Precision	Recall	F1
double	1.00	0.98	0.99
half	0.97	0.97	0.97
quadruple	1.00	1.00	1.00
quarter	0.96	0.98	0.97
macro avg	0.98	0.98	0.98
weight avg	0.98	0.98	0.98
accuracy	0.98		

Table 4.8: Metrics of the multi-class classifier with 256 pixels width, linear scale

In tables 4.7 and 4.8 the results for the width of 256 pixels can be seen. Doubling the input size once more gave use a few percentage points in every metric, and the weak recall for the "wrong" class went from 0.87 to 0.94, which is much better. In addition, the accuracy on the test set was 95.9%. The confusion matrix at figure 4.3 is very similar to the previous case as well. Ultimately the latter 256px input width was chosen in order to maximize accuracy, even though the 128px option would have been a good choice as well.

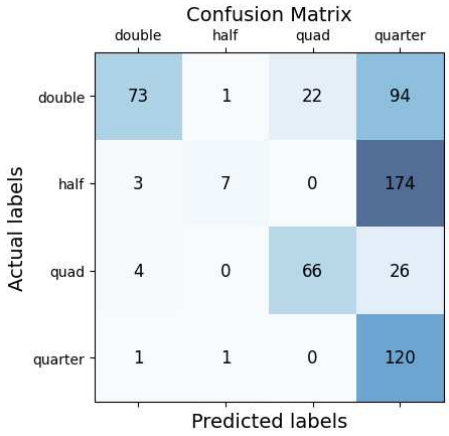


Figure 4.1: 64 pixels width

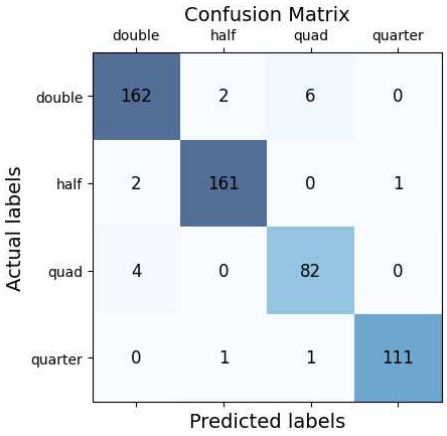


Figure 4.2: 128 pixels width

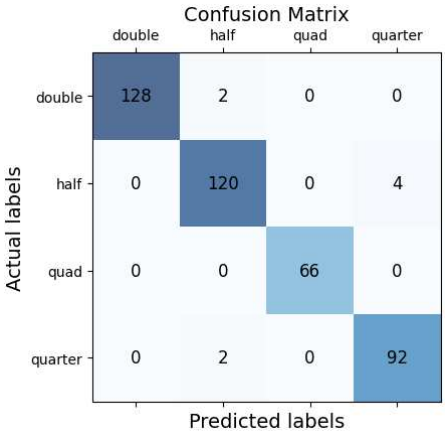


Figure 4.3: 256 pixels width

Confusion matrices with different input sizes, linear scale

4.4 Comparing Scales

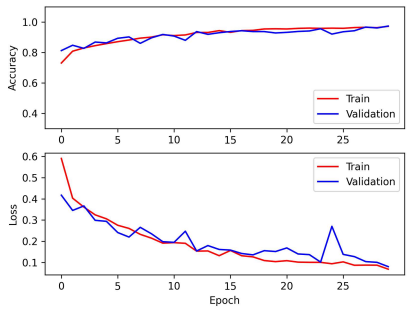
Having now fixed the input size, in this section the classifiers will be trained using the three different scales and the same quality metrics as before will be used to compare the performances.

In addition, some plots will show how the loss and the accuracy on validation and training sets vary during the training process, to check for overfitting. At first glance, if the two curves go hand-in-hand, then there shouldn't be any.

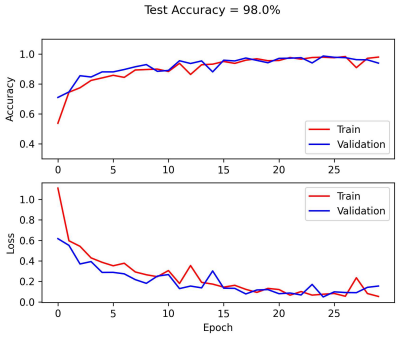
Linear Scale

The training process of the binary model using the linear scale on the y-axis is visualized in figure 4.4. The top half plots the accuracy as the epochs go on, while the bottom one plots the loss.

The accuracy side looks really good, as the training and validation curves go hand-



(a) Training of **binary** classifier, linear scale



(b) Training of **multi-class** classifier, linear scale

Figure 4.4: Plot of training phase of linear scale classifiers

in-hand. The loss for the binary classifier is slightly worse, and there seemed to be a bit of overfitting in some epochs, but it subdues at the end.

The quality metrics for these two models were previously computed, and are contained in tables 4.7 and 4.8. Up next, they will be compared to the logarithm and mel ones to see how they stack up against each other.

Logarithm Scale

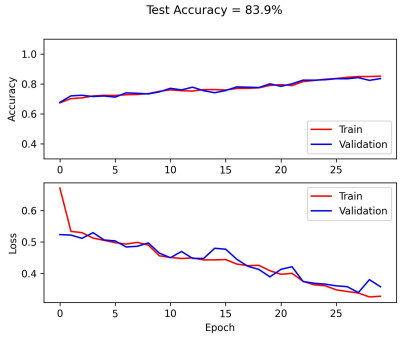
The plots of the training phase using the logarithm scale are shown in figure 4.5. The accuracy doesn't get as high as the linear case, both in the training and validation curves. There seems to be close to no overfitting. The most striking metric is the lower accuracy on the test set for the binary classifier, which came out at 83.9% against the 95.9% of the previous model. The multi-class model, on the other hand, performed better on the test set, with an accuracy of 98.5% against the 98% of the linear case.

Regarding the other quality metrics, they can be seen at tables 4.9 and 4.10. Again, for the binary model the precision, recall and f1 score are significantly lower for the logarithm scale. In particular, the recall for the "wrong" class is went from 0.95 in the binary-linear to 0.73 in the binary-logarithm. All the averages have a difference of at least 0.12 as well.

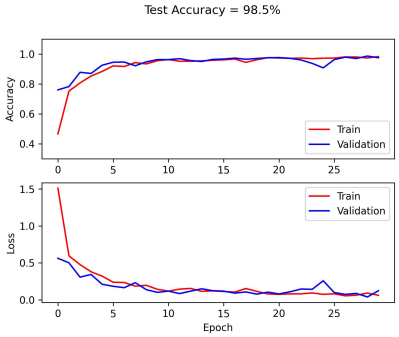
Interestingly, the multi-class model performed almost exactly the same as the linear case, with all the averages being equal.

Mel Scale

The plots of the training phase for the mel scale are contained in figure 4.6. The curves look almost identical to the linear case, and again there is no apparent overfitting. The accuracy on the test set is the highest yet for the binary classifier, at 97.4%, and for the multi-class one it's comparable to the other two cases. The quality metrics for the binary classifier are also the best among the three, with the



(a) Training of **binary** classifier, logarithm scale



(b) Training of **multi-class** classifier, logarithm scale

Figure 4.5: Plot of training phase of logarithm scale classifiers

Label	Precision	Recall	F1
c	0.87	0.89	0.88
w	0.76	0.73	0.75
macro avg	0.82	0.81	0.81
weight avg	0.84	0.84	0.84
accuracy	0.84		

Table 4.9: Metrics of the binary classifier with 256 pixels width, logarithm scale

Label	Precision	Recall	F1
double	0.97	0.98	0.98
half	1.00	0.98	0.99
quadruple	0.97	0.97	0.97
quarter	0.98	0.99	0.98
macro avg	0.98	0.98	0.98
weight avg	0.98	0.98	0.98
accuracy	0.98		

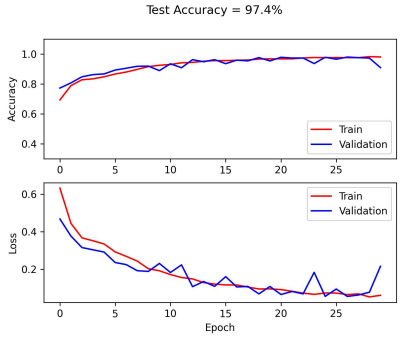
Table 4.10: Metrics of the multi-class classifier with 256 pixels width, logarithm scale

averages gaining two percentage points over the linear case, and of course beating the logarithm by a large margin. It is especially noteworthy how the precision for the "wrong" class is 0.5 higher than the linear scale. The metrics for the multi-class are also the best, although only by a tiny margin, as the averages are at most 0.01 higher than the other two.

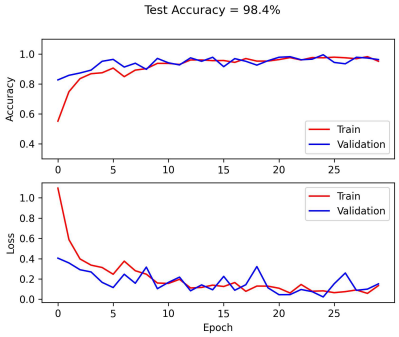
4.4.1 Assessment

With this knowledge, the conclusion is that the linear and mel scale are both valid options with respect to binary classification, while the logarithm scale seems more unreliable. The big difference in the performance of the binary classifier could suggest that the higher end of the frequency spectrum carries more information about the irregularities, since the logarithm scale gives more space to the lower octaves than the linear one. For the multi-class classification of the speed change factor, all the scales seem viable, even though the mel one lands on top by a couple percentage points in the average metrics.

It must be noted that these are results extracted on the test set, but sometimes machine learning models perform really well on samples which are similar to the training data but then fail to generalize well on new content.



(a) Training of **binary** classifier, mel scale



(b) Training of **multi-class** classifier, mel scale

Figure 4.6: Plot of training phase of mel scale classifiers

Label	Precision	Recall	F1
c	0.98	0.99	0.98
w	0.97	0.95	0.96
macro avg	0.97	0.97	0.97
weight avg	0.97	0.97	0.97
accuracy	0.97		

Table 4.11: Metrics of the binary classifier with 256 pixels width, mel scale

Label	Precision	Recall	F1
double	0.98	0.99	0.99
half	1.00	0.96	0.98
quadruple	0.98	1.00	0.99
quarter	0.97	1.00	0.98
macro avg	0.98	0.99	0.98
weight avg	0.99	0.98	0.98
accuracy	0.98		

Table 4.12: Metrics of the multi-class classifier with 256 pixels width, mel scale

To see if there are relevant differences between the scales and if the models have sufficient generalization, numerous tests will be ran on completely different data. A new dataset of completely different samples will be appositely generated to try out the models' capabilities. This will be shown in the next chapter.

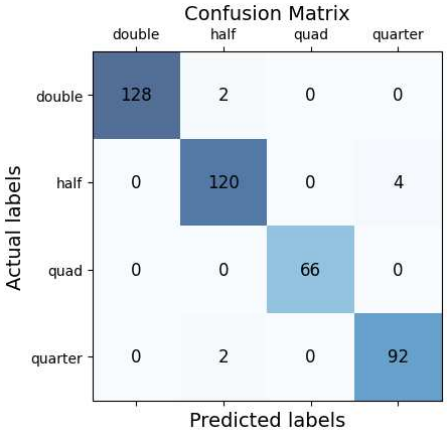


Figure 4.7: Linear scale

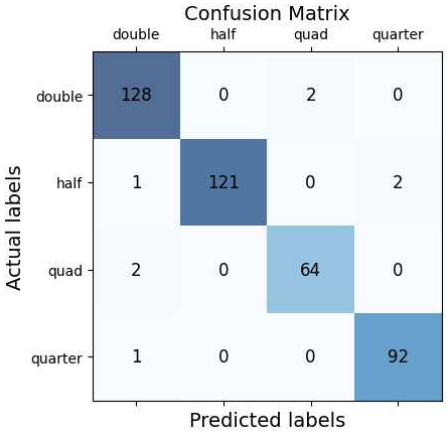


Figure 4.8: Logarithm scale

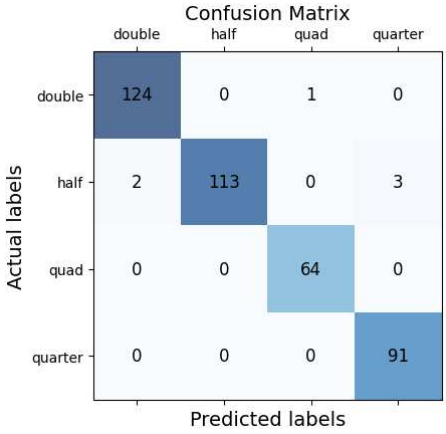


Figure 4.9: Mel scale

Confusion matrices with different scales, 256 pixels width

Chapter 5

Testing

This chapter is dedicated to measuring how well the classifiers react to new information. In the previous chapter the metrics were attained on the test split that was extracted from the initial dataset, but unfortunately the samples were pretty similar among each other, so they cannot be taken for granted.

The metrics used will be the same as before: accuracy, precision, recall and F1 score, in order to have a fair comparison with the results obtained in the last chapter. The question to answer is if the new results are deemed satisfactory and whether these models can be used in the final version of the software. There will also be some considerations on how the results could be improved.

5.1 Test Dataset

Now that the models have been trained and that the Mel scale was chosen to be the most appropriate to use in terms of metrics, the decision was to generate some new samples of digitised tapes in order to see if the models perform well.

The new dataset consists of 100 audio samples that contain multiple speed changes throughout a single file. This material will also be used to test the final version of the software, so it is useful to see how the program will respond to different speed changes. An example of a spectrogram is in figure 5.1.

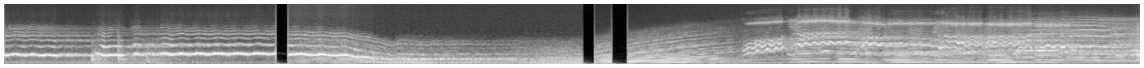


Figure 5.1: Spectrogram of a tape recorded at 7.5ips and reproduced at the correct speed, then 3.75ips and finally 15ips

In order to compare the performance obtained in the last chapter, the same metrics need to be used, and for that labeled dataset is needed. Like for the previous experiments, the samples were processed using the python scripts: divide the channels, compute the spectrograms using the Mel scale, divide them and segment them. The binary test set is summarised in table 5.1.

Label	Samples
correct	2,847
wrong	4,905
Total	7,752

Table 5.1: Summary of the binary test dataset

The subfolder of irregular images can then be subdivided to make the multi-class test set, which is summarised in table 5.2.

Label	Samples
double	2,105
half	1,540
quadruple	759
quarter	501
Total	4,905

Table 5.2: Summary of the multi-class test dataset

5.2 Performance On Test Dataset

The experiment can be recreated with the notebook `model_test.ipynb` in the repository, and the `README.md` file also contains a Google Drive link to download the test dataset. Table 5.3 contains the results for the binary classifier.

Label	Precision	Recall	F1
c	0.70	0.93	0.80
w	0.95	0.77	0.85
macro avg	0.82	0.85	0.82
weight avg	0.86	0.86	0.83
accuracy	0.83		

Table 5.3: Metrics of the binary classifier on the test dataset

Label	Precision	Recall	F1
double	0.82	0.73	0.77
half	0.94	0.63	0.76
quadruple	0.55	1.00	0.71
quarter	0.67	0.83	0.74
macro avg	0.75	0.80	0.74
weight avg	0.80	0.75	0.75
accuracy	0.75		

Table 5.4: Metrics of the multi-class classifier on the test dataset

As can be seen, there is a noteworthy performance drop from the previous experiments. Nonetheless, the scores are still good: the averages are all above 0.82 (for reference, previously they were all greater than 0.97). Strangely, for the "correct" class the recall is much higher than the precision, while for the "wrong" class it's exactly the opposite. The F1 score, on the other hand, is more stable.

For the multi-class experiment, the results are in table 5.4. The difference is even more noticeable, with a 20% drop in almost all metrics. The same pattern that was present in the binary case emerges, where precision and recall are inversely

proportional. The F1 score is, again, more stable and averaging 0.75, which is much lower than the previous 0.98 but still at an acceptable level.

Figure 5.2 contains the confusion matrix for the multi-class model: it's not surprising that the classifier has a hard time discerning between the two speed-ups (double and quadruple) and the two speed-downs (half and quarter), since their effect on the frequency spectrum is very similar. On the other hand, it's very strange that the model often classifies samples labeled "half" (second row) as "double" and sometimes as "quadruple". This phenomenon doesn't happen nearly as often in any other class.

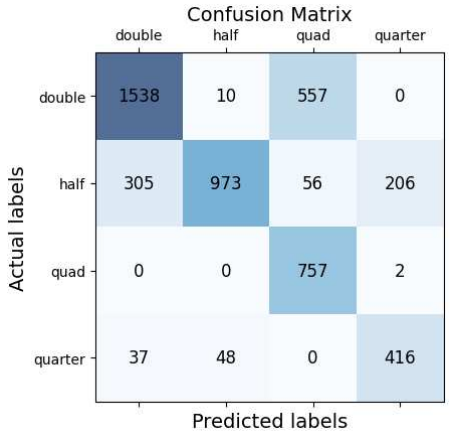


Figure 5.2: Confusion matrix computed by the multi-class classifier on the test dataset

This confusion between classes could have a number of different reasons: for sure the training set wasn't very balanced, and that can possibly cause bias for some classes over others.

It could also be an overfitting problem, and in this case a different network configuration maybe would work better. Adding some dropout layers and a batch normalization layer is surely a possibility to explore in the future. Reducing the number of trainable parameters through decreasing the amount of fully-connected layers and their respective neuron count is also a viable option.

The mistakes made by the classifiers may also be caused by some "glissandi" regions which made it into the dataset, even though they're very few. Most of them were skipped during the creation of the spectrograms, but the results could still be a bit skewed. However, if that's the case, in real magnetic tapes case studies these glissandi are not present, so this problem wouldn't persist.

Finally, it is also possible that, since different genres of music appear wildly different in the frequency domain, the models perform well on some genres and badly on others. The difference in the spectrograms can be appreciated in figure 5.3, where the upper spectrogram was computed from a spoken word sample, the middle one from an opera performance and the lower one is taken from a drum track.

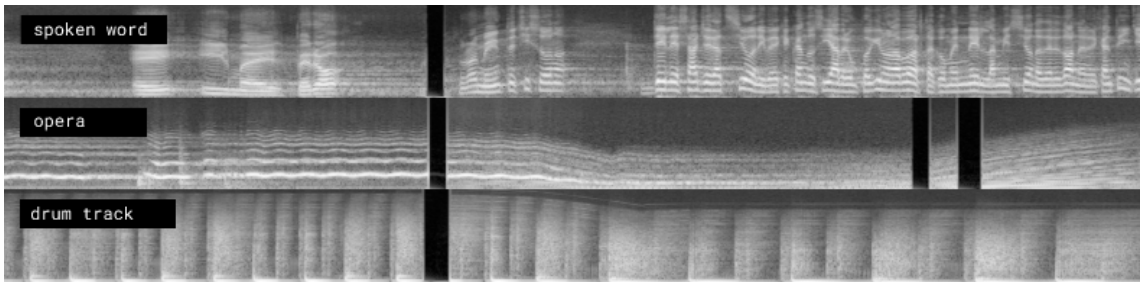


Figure 5.3: Mel spectrograms of different genres, where you can see the broad variety in frequency responses

After doing some tests on the samples separately, it became apparent that the multi-class model works better with opera music than with spoken word. In fact, in the speech samples the class "double" gets very often mislabeled as "quadruple", which doesn't happen as much in the other genres. This could be fixed by feeding the models more of the "problematic" material, although it is not certain to work.

Some Considerations

It's undeniable that there was a significant performance drop in the transition from the initial samples to this new dataset. This is in fact very common, with some machine learning models performing perfectly on the test-validation sets but not learning anything on data never seen before.

The results should not invalidate the approach at all, in fact they are very promising: considering the training samples used were just 300 and did not cover every genre, and added in the fact that the structure of the networks is very simple, it's fair to predict that with proper adjustments the future research holds really good results.

It's also worth noting that as of right now, the models make predictions on the segments as if they are isolated from each other, without any kind of memory about past predictions. When in the final version of the software the analyzed spectrogram is scanned from left to right, there's the possibility of externally influencing the sigmoid and softmax activation functions in the last layer in order to take account of the past predictions. An alternative would be to store all the predictions made and analyze the resulting array in post, identifying the irregular regions with a bird's eye view. This would mitigate the errors made by the classifiers without needing to improve the performance metrics so much.

The next step is to build a piece of software that uses these models to implement the strategy described in chapter 2. The program will take as input an audio file of a digitised tape and output all the time intervals which are deemed irregular, if there are any. This will be described in the next chapter.

Chapter 6

Implementation

This chapter will describe the approach to the implementation of the audio analyzer using the classifiers obtained previously. The complete code can be seen in script `src/main.py` on the GitHub repository. Here the description will be done mostly using pseudocode.

The objective is to create a CLI program with these parameters:

- Two required input arguments: audio sample stored in WAV format and the scale to use when computing the spectrogram.
- Two optional arguments: resolution to use for the x-axis of the spectrogram and the step used when classifying irregularities.
- Output: text file containing timestamps and nature of all found irregularities.

The pseudocode is detailed below:

```
# src/main.py

def analyze_speed(sample, scale, bin_model, multi_model, res, step):
    create output folder

    divide audio channels

    for each channel:
        create text file where to write the logs

        bufferize channel, each buffer at most 60 seconds

        for each buf_index, buf:
            duration = get_duration(buf)
            buf_start_time = buf_index * 60

            # compute spectrogram using wav2spec
            spectrogram = wav2spec(buf) # numpy array
            height, width = spectrogram.shape[:2]
```

```
20
21              if width < 256:
22                  # audio too short to be analyzed
23                  break
24
25              # scan the whole spectrogram and divide it in windows
26              window_width = 256
27
28              for i in range(0, width//step):
29                  if i*step + window_width < width:
30                      window = spectrogram[0 : height,
31                                i*step : (i*step)+window_width]
32
33                      # classify the window
34                      binary_label = argmax(
35                          bin_model.predict(window))
36
37                      if binary_label == 1:
38                          # irregularity detected, get timestamp
39                          time = buf_start_time + (duration*i*step /
    width)
40
41                          # classify the speedup factor
42                          speedup_label = argmax(
43                              four_classes_model.predict(window))
44
45                          write irregularity on output logs
46
47
48 if __name__ == "__main__":
49      parse  arguments
50
51      if not valid .wav file
52          exit program
53
54      if not valid scale
55          exit program
56
57      analyze_speed(
58          sample, scale, bin_model, multi_model, res, step)
```

The operation of dividing the audio in buffers is done in order to avoid filling up too much memory. In fact, the digitized tapes are often dozens of minutes in length and even systems with large amounts of RAM struggle upon loading them directly. Moreover, computing a single spectrogram from a half-hour long sample using a 256px/s resolution will result in an image almost half a million pixels wide, which is obviously not ideal.

6.1 Future Developments

At lines 42-43 the prediction with the multi-class model is obtained. As of now the program treats every image as isolated, without being influenced by previous choices. This is not ideal, especially if the performance metrics are not amazing: intuitively, if the past thirty segments were labeled "double", then the next irregular segment will probably be of the same label as well.

For this task, instead of simply taking the argmax, the softmax function could be slightly modified to take in input some of the previously obtained predictions, so when the model is especially undecisive between two or more labels, the decision can be a more educated guess. For example, a possibility is to store in an array the past 10 predictions and increase some of the softmax outputs accordingly by a small ϵ. The natural question that arises is: how much should this ϵ be? Of course there's no way of answering without a lot of trial and error.

The same could be said about about the binary predictions, but the accuracy scores for the first classifier are considerably better, so it seems unnecessary to influence the sigmoid output function.

Another thing to note is the format of the log file: as of now it's just a plain text file, but in future developments the best thing to do would be to save the irregularities in JSON format in order to better comply with the MPAI standard and be fully integrated in the framework.

Chapter 7

Conclusion

The objective of the thesis was to explore the potential of artificial intelligence, more specifically convolutional neural networks, in aiding the digitization of old magnetic tape archives. It's safe to say that the approach used shows great potential, not only in preventing recording/playback speed irregularities but also in other audio problems that arise during the tape reproduction.

For example, another important parameter to configure on the reel-to-reel recorder is equalization. In fact, during recording, a pre-emphasis curve is applied to the original signal to maximize signal-to-noise ratio, and this must be compensated during the recording [17]. Discrepancies between the curve used during recording and the one used in playback could also be detected by CNNs, but there needs to be trials in the future to confirm it.

Additionally, sometimes the same tape gets used by multiple people in opposite directions, so the technicians have to manually correct the rotation of the tape so that the audio doesn't get inverted. The use of neural networks for solving this problem is being researched at CSC as of now.

Nevertheless, the current strategy certainly has much room for improvement, on multiple fronts:

- The dataset has to be enlarged, because 300 samples cannot cover sufficient musical variety, whether it's about genre, harmony and melody, effects, etc...
- A data augmentation process to enrich the existing dataset is also a step in the right direction, mainly through the introduction of noise or manipulating the gain parameter.
- For this work a simple ad-hoc network structure was used, mainly to keep things simple, as well as reducing training times for the six classifiers. This led to good results, but there are several predefined networks such as GoogleNet [18] that are more complex and therefore could detect patterns in the spectrograms that the current model doesn't.

- Other hyperparameters of the network like the optimizer, number of epochs and batch size could influence the models' generalization capabilities: a possibility is the technique known as early stopping, in which the training process stops if a certain number of iterations didn't lead to improvements in training/validation loss.

It also has to be noted that different reel-to-reel tape recorders have slight differences in frequency responses, so the spectrograms obtained from the samples used in this thesis may look slightly different from the spectrograms obtained with another machine. This means that maybe the models might not recognize patterns in images coming from other sources. A straightforward solution, again, is to include as many samples as possible in the training set, even samples reproduced with different recorders.

Speaking purely from a software development perspective, the code could be refactored for better clarity and performance, as well as a better integration in the MPAI standard. The part that needs most improvement is grouping together irregularities in time intervals instead of writing each found irregularity individually.

Appendixes

Appendix A

Code

A.1 Methods used for dividing spectrograms in half

```
# src/vision.py

import cv2 as cv
import numpy as np
import os
import utils
import librosa
import skimage.io


def mel_spectrogram_image(y, sr, out, hop_length, n_mels,
    dimensions, save):
    # use log-melspectrogram
    mels = librosa.feature.melspectrogram(
        y=y, sr=sr, n_mels=n_mels, n_fft=hop_length*4, hop_length=
    hop_length, fmax=20000)
    mels = np.log(mels + 1e-9)  # add small number to avoid log(0)

    # min-max scale to fit inside 8-bit range
    img = scale_minmax(mels, 0, 255).astype(np.uint8)
    img = np.flip(img, axis=0)  # put low frequencies at the bottom
    in image
    img = cv.resize(img, dimensions, cv.INTER_LINEAR)

    # save as PNG and return numpy array
    if save == True:
        skimage.io.imsave(out, img)

    return img


"""
```

```
30      extract the spectrogram of an audio sample loaded with librosa.
    load
31  """
32
33
34  def scale_minmax(X, min=0.0, max=1.0):
35      X_std = (X - X.min()) / (X.max() - X.min())
36      X_scaled = X_std * (max - min) + min
37
38      return X_scaled
39
40
41  def highlight_split(img, low_thresh=3, high_thresh=255, h_size=3,
    v_size=20):
42      # threshold the original image and extract vertical lines using
        vertical morphological operator
43      ret, thresh1 = cv.threshold(img, low_thresh, high_thresh, cv.
    THRESH_BINARY)
44      verticalStructure = cv.getStructuringElement(
45          cv.MORPH_RECT, (h_size, v_size))
46      ret_img = cv.dilate(thresh1, verticalStructure)
47
48      return ret_img
49
50
51  """
52      takes in input a spectrogram and highlights the split regions
53      returns an image of the same dimensions as the original, with
    the ROI highlighted in black
54  """
55
56
57  def find_splits(img):
58      height, width = img.shape[:2]
59      list_splits = []
60      i = 0
61
62      for j in range(0, height-1):
63          # start from the last seen black column if already
    encountered
64          if len(list_splits) > 1:
65              i = list_splits[-1][-1]
66
67          while (i < width):
68              # found a black pixel: append x coordinate at the end
    of the ROI
69              if not (img[j][i].any()):
70                  start_roi = i
71
72                  while (not (img[j][i]).any() and i < width):
73                      i += 1
74
75                  end_roi = i
76
```

```
                list_splits.append([start_roi, end_roi])
                break

            i += 1

    return list_splits


"""
    takes in input a thresholded image (pixels are either 0 or 255)
    returns a list of lists, with every list being an interval of x coordinates that make a black band
"""


def divide_half(img, filename, middle, left_path, right_path):
    height, width = img.shape[:2]

    left_roi = img[0:height, 0:middle[0]]
    right_roi = img[0:height, middle[1]:width-1]

    left_filename = left_path + "c_" + filename
    right_filename = right_path + "w_" + filename

    cv.imwrite(left_filename, left_roi)
    cv.imwrite(right_filename, right_roi)

    return [left_roi, right_roi]


"""
    takes in input an image and the middle region and saves the two halfs in separate files
    middle = list of two x coordinates, which mark the start and the end of the middle region
"""


def segment(img, step, window_width, multiple, offset):
    height, width = img.shape[:2]
    ret_list = []

    if multiple == False:
        window = img[0:height, offset:offset+window_width]
        ret_list.append(window)
    else:
        for i in range(0, width//step):
            if offset + i*step + window_width < width:
                window = img[0:height, offset + i *
                             step:offset+(i*step)+window_width]
                ret_list.append(window)

    return ret_list

```

```
128
129 """
130     takes in input an image (numpy array) and divides it into
    segments
131     returns a list that contains all the segments
132
133     multiple: boolean value. if set to false, only save the first
    segment of the image
134     offset: how many pixels from the left to use as starting point
135 """
136
137
138 def compute_segments(in_paths, out_paths, step, window_width,
    multiple, offset):
139     if len(in_paths) == 0:
140         return
141
142     if len(in_paths) != len(out_paths):
143         print(
144             f"Length of input list is {len(in_paths)} while length
    of output list is {len(out_paths)}")
145         return
146
147     for i, in_path in enumerate(in_paths):
148         for (root, dirs, files) in os.walk(in_path, topdown=True):
149             for filename in files:
150                 if filename.endswith('.png'):
151                     img = cv.imread(os.path.join(in_path + filename
    ))
152
153                     seg_list = segment(
154                         img, step, window_width, multiple, offset)
155                     for j, seg in enumerate(seg_list):
156                         out_name = out_paths[i] + \
157                             filename[0:-4] + "_" + str(j) + ".png"
158                         seg = cv.cvtColor(seg, cv.COLOR_RGB2GRAY)
159                         cv.imwrite(out_name, seg)
160
161
162 """
163     in_paths: list of input folders
164     out_paths: list of output folders
165     multiple: boolean value. if set to false, only save the first
    segment of the image
166     offset: how many pixels from the left to use as starting point
    (used to skip most of the glissando region)
167 """
```

A.2 Script used for computing the spectrograms

```
# src/wav2spec.py

import os
import sys
import numpy as np
import argparse
import subprocess
import librosa

import utils
import vision


def wav2spec(sample, scale, out_path, width_res=256):
    if not out_path.endswith("/"):
        out_path = out_path + "/"
    if not os.path.isdir(out_path):
        utils.create_folder(out_path)

    # create spectrogram's filename
    out_name = sample.split("/")
    out_name = out_path + out_name[-1].split(".")[0] + '.png'

    duration = round(librosa.get_duration(filename=sample), 2)
    if duration < 1:
        print(f"file {sample} is too short. Skipping to next one.."
    )
        return

    # mel-spec
    if scale == "mel":
        y, sr = librosa.load(sample, sr=None)

        vision.mel_spectrogram_image(
            y, sr, out_name, hop_length=512, n_mels=128, dimensions
    =(int(duration*width_res), 128), save=True)

    else:
        width = str(duration*width_res)
        height = str(128)

        # lin-spec
        if scale == "lin":
            subprocess.call([
                'sox',
                sample,
                '-n', 'spectrogram',
                '-y', height,
                '-x', width,
                '-r',
                '-m',
                '-R', '0:20k',
```

```
51                  '-o', out_name,
52              ])
53
54          # log-spec
55          elif scale == "log":
56              subprocess.call([
57                  'sox',
58                  sample,
59                  '-n', 'spectrogram',
60                  '-y', height,
61                  '-x', width,
62                  '-r',
63                  '-m',
64                  '-L',
65                  '-R', '0:20k',
66                  '-o', out_name,
67              ])
68
69      return out_name
70
71
72  if __name__ == "__main__":
73      parser = argparse.ArgumentParser(
74          description="Convert .wav file(s) to .png spectrograms")
75
76      # required argument: audio sample
77      parser.add_argument('-i', '--input', type=str,
78                          help="""Path to the audio sample(s) to
    analyze. It can be either
79                          a folder or a single file""")
80
81      # required argument: output folder
82      parser.add_argument('-o', '--output', type=str,
83                          help="""Path in which to store the output.
    If it doesn't exist, it
84                          will be created""")
85
86      # required argument: scale
87      parser.add_argument('-s', '--scale', type=str,
88                          help="""What scale to use on the y-axis of
    the spectrogram.
89                          possible options are 'log', 'mel' or 'lin'
    """)
90
91      # optional arguments: resolution of x-axis (pixels per second)
92      parser.add_argument('-w', '--width', nargs='?', type=int,
    default=256,
93                          help="What resolution to use on the x-axis,
     in pixels/s. Default is 256.")
94
95      args = parser.parse_args()
96
97      in_path = args.input
98      out_path = args.output
```

```
99      scale = args.scale
100     width_res = args.width
101
102     # check validity of scale parameter
103     if scale not in ["log", "mel", "lin"]:
104         print(
105             f"{scale} is not a valid scale option, see wav2spec.py
    -h for help. Exiting program")
106         sys.exit(1)
107
108     if in_path.endswith(".wav"):
109         wav2spec(sample=in_path, scale=scale,
110                  out_path=out_path, width_res=width_res)
111
112     elif os.path.isdir(in_path):
113         sample_list = utils.collect_audio_files(in_path)
114
115         if len(sample_list) == 0:
116             print(f"{in_path} doesn't contain any .wav samples.
    Exiting program.")
117             sys.exit(1)
118
119         for sample in sample_list:
120             wav2spec(sample=sample, scale=scale,
121                      out_path=out_path, width_res=width_res)
122
123     else:
124         print(f"{in_path} is neither a .wav file, nor a folder.")
125         sys.exit(1)
```

A.3 Main audio analyzer script

```
# src/main.py

import os
import sys
import numpy as np
import argparse

import utils
import vision
from wav2spec import wav2spec

import cv2 as cv
from scipy.io import wavfile
import librosa


def analyze_speed(sample, scale, binary_model_path, four_classes_model_path, width_res, step):
    from keras.models import load_model

    # load the keras classifiers
    binary_model = load_model(binary_model_path)
    four_classes_model = load_model(four_classes_model_path)

    # speedup corresponding to the label output by the second classifier
    speedup_dict = {
        0: "double",
        1: "half",
        2: "quadruple",
        3: "quarter"
    }

    # create output folder
    out_path = os.path.dirname(os.path.realpath('__file__')) + "/output/"
    utils.create_folder(out_path)

    # read audio sample
    fs, data = wavfile.read(sample)

    # sample is mono
    if len(data.shape) == 1:
        data.shape = (data.shape[0], 1)

    # for each channel:
    for i in range(0, data.shape[1]):
        log_filename = out_path + "output-ch" + str(i) + ".txt"
        with open(log_filename, 'w') as log_file:
            log_file.write(f"Filename: {sample}\n")
            print(f"Created file {log_filename}")

```

```
50          # create folders where to save the separate audio and
    segmented spectrograms
51          channel_path = out_path + "ch" + str(i) + "/"
52          segments_path = out_path + "segments_ch" + str(i) + "/"
53          utils.create_folder(channel_path)
54          utils.create_folder(segments_path)
55
56          # save audio of channel
57          channel_filename = channel_path + "ch" + str(i) + ".wav"
58          wavfile.write(channel_filename, fs, data[:, i])
59
60          # load audio of channel and get the duration
61          y, sr = librosa.load(channel_filename, offset=0.0, duration
    =None)
62          duration = round(librosa.get_duration(filename=sample), 2)
63          with open(log_filename, 'a') as log_file:
64              log_file.write(f"Duration: {duration}s\n")
65
66          # save spectrogram and load it into numpy array
67          spec_name = wav2spec(channel_filename, scale, channel_path,
     width_res)
68          spectrum = cv.imread(spec_name, cv.IMREAD_GRAYSCALE)
69          height, width = spectrum.shape[:2]
70
71          if width < 256:
72              print("The audio file is too short to be analyzed")
73              break
74
75          # settings for the scanning of the spectrogram
76          offset = 0
77          window_width = 256
78
79          # scan the whole spectrogram and divide it in windows (
    segments)
80          for i in range(0, width//step):
81              if offset + i*step + window_width < width:
82                  window = spectrum[0:height, offset+i *
83                                    step:offset+(i*step)+window_width
    ]
84                  window = np.expand_dims(window, axis=0)
85
86                  # classify the window
87                  binary_label = np.argmax(
88                      binary_model.predict(window, verbose=0))
89
90                  if binary_label == 1:  # irregularity detected
91                      timestamp = round(duration * (i * step) / width
    , 2)
92
93                      # classify the speedup factor
94                      speedup_label = np.argmax(
95                          four_classes_model.predict(window, verbose
    =0))
96                      print(
```

```
97                          f"Segment {i}, time: {str(timestamp)}s,
    speedup: {speedup_dict[speedup_label]}")
98                      with open(log_filename, 'a') as log_file:
99                          log_file.write(
100                             f"Segment {i}, time: {str(timestamp)}s,
     speedup: {speedup_dict[speedup_label]}\n")
101
102         # save the windows for manual analysis
103         vision.compute_segments([channel_path], [segments_path],
    step=step,
104                                 window_width=window_width, multiple
    =True, offset=offset)
105
106
107 if __name__ == "__main__":
108     script_path = utils.get_script_path()
109     models_path = os.path.dirname(script_path) + "/models/"
110
111     # parse input arguments
112     parser = argparse.ArgumentParser(
113         description="""Analyze a digitised magnetic audio tape and
    detect discrepancies
114         between the recording speed and the playback speed.""")
115
116     # required arguments: audio sample
117     parser.add_argument('-i', '--input', type=str,
118                         help='Path to the WAV audio sample to
    analyze.')
119
120     # required arguments: scale for y-axis
121     parser.add_argument('-s', '--scale', type=str,
122                         help="""Which scale to use for the y-axis
    of the spectrogram. Also switches to the
123                         correct model to use for the classification
    . Valid choices are 'log', 'lin' and 'mel.""")
124
125     # optional arguments: resolution of x-axis (pixels per second)
126     parser.add_argument('-w', '--width', nargs='?', type=int,
    default=256,
127                         help="What resolution to use on the x-axis,
     in pixels/s. Default is 256.")
128
129     # optional arguments: step
130     parser.add_argument('-j', '--jump', nargs='?', type=int,
    default=64,
131                         help="What step to use for the scanning of
    the spectrum. Default is 64 pixels.")
132
133     # read the input parameters and check for correctness
134     args = parser.parse_args()
135     sample = args.input
136     scale = args.scale
137     width_res = args.width
138     step = args.jump
```

```
139
140     if not (sample.endswith(".wav")):
141         print("The input file is not a .wav file.")
142         sys.exit(1)
143
144     if not (scale in ['lin', 'log', 'mel']):
145         print("The scale chosen is not valid. See main.py -h for
    information.")
146
147     # define paths to models
148     binary_models = {
149         'lin': models_path + "model-binary-lin/",
150         'log': models_path + "model-binary-log/",
151         'mel': models_path + "model-binary-mel/"
152     }
153
154     four_classes_models = {
155         'lin': models_path + "model-4c-lin/",
156         'log': models_path + "model-4c-log/",
157         'mel': models_path + "model-4c-mel/"
158     }
159
160     analyze_speed(
161         sample, scale, binary_models[scale], four_classes_models[
    scale], width_res, step)
```

A.4 Jupyter Notebook for training the models

```
# notebooks/google_fit_colab.ipynb

# Import files from Google Drive
from google.colab import drive
drive.mount("/content/drive")
!tar -xzf drive/MyDrive/dataset-binary-complete-separatechannels.tar.gz

import os
import numpy as np
import shutil
import math
import random

def delete_folder(path):
    for filename in os.listdir(path):
        file_path = os.path.join(path, filename)
        try:
            if os.path.isfile(file_path) or os.path.islink(file_path):
                os.unlink(file_path)
            elif os.path.isdir(file_path):
                shutil.rmtree(file_path)
        except Exception as e:
            print('Failed to delete %s. Reason: %s' % (file_path, e))

def create_folder(path, overwrite=True):
    if not os.path.exists(path):
        os.makedirs(path)
        print(f"Created folder {path}")
    else:
        if overwrite == False:
            print(f"Couldn't substitute folder because overwrite is set to False")
        else:
            delete_folder(path)
            print(f"Substituted folder {path}")

"""
    inputs: in_paths  ---> list of input paths, where every path corresponds to a different label
            out_paths ---> names of training, validation and test folders, in this order
              ratios  ---> what fraction of the dataset goes into training, validation and testing
              seed    ---> seed used for randomization
"""
def create_dataset(in_paths, labels, out_paths, ratios, seed):
    if len(ratios) != 3 or len(out_paths) != 3:
        print("Output configuration is wrong")
        return
```

```
46
47      if np.sum(ratios) > 1.0:
48          print("Sum of ratios must be less than 1")
49          return
50
51      for out_path in out_paths:
52          create_folder(out_path)
53      for label_index, in_path in enumerate(in_paths):
54          # label = in_path.split('/')[-2]
55          label = labels[label_index]
56
57          for (root, dirs, files) in os.walk(in_path, topdown=True):
58              # number of elements in each split
59              n_train = math.floor(ratios[0] * len(files))
60              n_valid = math.floor(ratios[1] * len(files))
61              n_test = len(files) - (n_train + n_valid)
62
63              train_files = []
64              valid_files = []
65              test_files = []
66
67              # create list of random indexes and shuffle it
68              indexes = list(range(0, len(files)))
69              random.Random(seed).shuffle(indexes)
70
71              for j in range(0, len(files)):
72                  index = indexes[j]
73                  if j < n_train:
74                      train_files.append(files[index])
75                  elif n_train <= j < n_train + n_valid:
76                      valid_files.append(files[index])
77                  else:
78                      test_files.append(files[index])
79
80              for i, out_path in enumerate(out_paths):
81                  create_folder(out_path + label + "/")
82
83                  if i == 0:
84                      for filename in train_files:
85                          shutil.copyfile(in_path + filename,
    out_path + label +  "/" + filename)
86                  elif i == 1:
87                      for filename in valid_files:
88                          shutil.copyfile(in_path + filename,
    out_path + label +  "/" + filename)
89                  else:
90                      for filename in test_files:
91                          shutil.copyfile(in_path + filename,
    out_path + label +  "/" + filename)
92
93  # Create the binary dataset split
94  # For the one with four classes, see next cell
95
96  correct_segments = "/content/dataset/c/"
```

```
wrong_segments = "/content/dataset/w/"
labels = ['c', 'w']

in_paths = [correct_segments, wrong_segments]

train_path = "/content/training-set/"
validation_path = "/content/validation-set/"
test_path = "/content/test-set/"

create_dataset(in_paths=in_paths, labels=labels, out_paths=[
    train_path, validation_path, test_path], ratios=[0.7, 0.2, 0.1],
     seed=2023)

# Create the dataset with four classes

wrong_segments = "/content/dataset/w/"
new_dataset_path = "/content/dataset-4classes/"

labels = ['double', 'half', 'quadruple', 'quarter']

create_folder(new_dataset_path, overwrite=True)

in_paths = []

for label in labels:
    create_folder(new_dataset_path + label + "/")
    in_paths.append(new_dataset_path + label + "/")

# add spectrograms to subdirectories depending on the name of the
    file
for (root, dirs, files) in os.walk(wrong_segments, topdown=True):
    for filename in files:
        if "3,75ips_to15ips" in filename:
            shutil.copyfile(wrong_segments + filename,
    new_dataset_path + 'quadruple' +  "/" + filename)
        elif "15ips_to3,75ips" in filename:
            shutil.copyfile(wrong_segments + filename,
    new_dataset_path + 'quarter' +  "/" + filename)
        elif "3,75ips_to7,5ips" in filename:
            shutil.copyfile(wrong_segments + filename,
    new_dataset_path + 'double' +  "/" + filename)
        elif "7,5ips_to15ips" in filename:
            shutil.copyfile(wrong_segments + filename,
    new_dataset_path + 'double' +  "/" + filename)
        elif "15ips_to7,5ips" in filename:
            shutil.copyfile(wrong_segments + filename,
    new_dataset_path + 'half' +  "/" + filename)
        elif "7,5ips_to3,75ips" in filename:
            shutil.copyfile(wrong_segments + filename,
    new_dataset_path + 'half' +  "/" + filename)

train_path = "/content/training-set/"
validation_path = "/content/validation-set/"
test_path = "/content/test-set/"
```

```
142
143 create_dataset(in_paths=in_paths, labels=labels, out_paths=[
    train_path, validation_path, test_path], ratios=[0.7, 0.2, 0.1],
     seed=1998)
144
145 # Define model structure
146 # For the binary model select last layer with sigmoid activation.
    Otherwise, select the softmax one
147
148 from keras.layers import Dense, Conv2D, MaxPooling2D,
    GlobalAveragePooling2D, Dropout, AveragePooling2D
149 from keras import models
150
151 my_model = models.Sequential()
152 my_model.add(Conv2D(8, (7, 7), activation='relu', padding='same',
    input_shape=(128, 256, 1)))
153 my_model.add(MaxPooling2D((5, 5), padding='same'))
154
155 my_model.add(Conv2D(16, (7, 7), activation='relu', padding='same'))
156 my_model.add(MaxPooling2D((5, 5), padding='same'))
157
158 my_model.add(Conv2D(32, (7, 7), activation='relu', padding='same'))
159 my_model.add(MaxPooling2D((5, 5), padding='same'))
160
161 my_model.add(GlobalAveragePooling2D())
162
163 my_model.add(Dense(32, activation='relu'))
164 my_model.add(Dropout(0.2))
165
166 my_model.add(Dense(4, activation='softmax'))
167 # otherwise my_model.add(Dense(2, activation='sigmoid'))
168
169
170 # Compile the model
171 # For the binary model, select the binary_crossentropy loss
    function, otherwise the categorical_crossentropy
172
173 from tensorflow.python.keras.callbacks import EarlyStopping,
    ModelCheckpoint
174 es = EarlyStopping(monitor='val_loss', mode='min', verbose=1,
    patience=30)
175 mc = ModelCheckpoint('model/', monitor='val_accuracy', mode='max',
    verbose=1, save_best_only=True)
176 cb_list = [es, mc]
177
178 my_model.compile(optimizer='adam', loss='categorical_crossentropy',
     metrics=['accuracy'])
179 # otherwise my_model.compile(optimizer='adam', loss='
    binary_crossentropy', metrics=['accuracy'])
180
181 # Training
182
183 import cv2
184 import numpy as np
```

```
185 from tensorflow.keras.preprocessing.image import ImageDataGenerator
186
187 batch_size = 20
188 data_generator = ImageDataGenerator(preprocessing_function=None)
189 train_generator = data_generator.flow_from_directory('/content/
    training-set',
190                                                     target_size =
    (128, 256),
191                                                     batch_size=
    batch_size,
192                                                     color_mode='
    grayscale',
193                                                     class_mode='
    categorical')
194
195 validation_generator = data_generator.flow_from_directory('/content
    /validation-set',
196                                                     target_size =
    (128, 256),
197                                                     color_mode='
    grayscale',
198                                                     batch_size=
    batch_size,
199                                                     class_mode='
    categorical')
200
201 n_train = len(train_generator.filenames)
202 n_valid = len(validation_generator.filenames)
203
204 history = my_model.fit(
205     train_generator,
206     epochs=30,
207     steps_per_epoch = n_train//batch_size,
208     validation_data=validation_generator,
209     validation_steps=n_valid//batch_size,
210     callbacks=cb_list
211 )
212
213
214
215 # Testing
216 # The test ouputs the fraction of the test samples which have been
    correctly classified
217
218 from keras.models import load_model
219 import numpy as np
220
221 model = load_model('model')
222
223 # generate data for test set of images
224 test_generator = data_generator.flow_from_directory(
225         '/content/test-set/',
226         target_size=(128, 256),
227         color_mode='grayscale',
```

```
228         batch_size=1,
229         class_mode='categorical',
230         shuffle=False)
231
232 filenames=test_generator.filenames
233 n_test = len(filenames)
234
235 # obtain predicted activation values for the last dense layer
236 test_generator.reset()
237 pred=model.predict(test_generator, verbose=1, steps=n_test)
238
239 # determine the maximum activation value for each sample
240 predicted_class_indices=np.argmax(pred,axis=1)
241 actual_labels=test_generator.labels
242 class_labels = list(test_generator.class_indices.keys())
243
244 # determine the test set accuracy
245 match=[]
246 for i in range(0, n_test):
247     match.append(predicted_class_indices[i]==actual_labels[i])
248
249 acc = str(match.count(True) / n_test * 100)[0:4]
250 print(f"The model predicted accurately {acc}% of the samples")
251
252 # Plotting the results
253 import matplotlib.pyplot as plt
254
255 fig, axs = plt.subplots(2)
256 title = "Test Accuracy = " + acc + "%"
257 fig.suptitle(title)
258 axs[0].plot(history.history['accuracy'], color='r')
259 axs[0].plot(history.history['val_accuracy'], color='b')
260 axs[1].plot(history.history['loss'], color='r')
261 axs[1].plot(history.history['val_loss'], color='b')
262
263 axs[0].set_ylim([.3, 1.1])
264 axs[0].set_xticks(range(0, len(history.history['accuracy']), 5))
265 axs[1].set_xticks(range(0, len(history.history['accuracy']), 5))
266
267 axs[0].set_ylabel('Accuracy')
268 axs[1].set_ylabel('Loss')
269
270 axs[1].set_xlabel('Epoch')
271 axs[0].legend(['Train', 'Validation'], loc='upper left')
272 axs[1].legend(['Train', 'Validation'], loc='upper right')
273 plt.savefig("plot.png", dpi=300)
274
275 # For the model with four classes, plot the confusion matrix
276 from sklearn.metrics import confusion_matrix
277
278 conf_matrix = confusion_matrix(y_true=actual_labels, y_pred=
       predicted_class_indices)
279
280 fig, ax = plt.subplots(figsize=(4.5, 4.5))
```

```
ax.matshow(conf_matrix, cmap=plt.cm.Blues, alpha=0.7)

for i in range(conf_matrix.shape[0]):
    for j in range(conf_matrix.shape[1]):
        ax.text(x=j, y=i,s=conf_matrix[i, j], va='center', ha='
   center', size='large')

plt.xlabel('Predicted labels', fontsize=14)
plt.ylabel('Actual labels', fontsize=14)
ax.xaxis.set_ticklabels(['', 'double', 'half', 'quad', 'quarter'])
ax.yaxis.set_ticklabels(['', 'double', 'half', 'quad', 'quarter'])
plt.title('Confusion Matrix', fontsize=14)
plt.savefig("confusion_matrix.png", dpi=300)
plt.show()
```

A.5 Jupyter Notebook for testing the models

```
# notebooks/model_test.ipynb

# binary model
from keras.models import load_model
import numpy as np
from tensorflow.keras.preprocessing.image import ImageDataGenerator
data_generator = ImageDataGenerator(preprocessing_function=None)
from sklearn.metrics import classification_report

model = load_model("../models/model-binary-mel")

# generate data for test set of images
test_generator = data_generator.flow_from_directory(
        "/home/lorenzo/Pictures/test-dataset/binary-seg/",
        target_size=(128, 256),
        color_mode='grayscale',
        batch_size=1,
        class_mode='categorical',
        shuffle=False)

filenames=test_generator.filenames
n_test = len(filenames)

# obtain predicted activation values for the last dense layer
test_generator.reset()
pred=model.predict(test_generator, verbose=1, steps=n_test)

# determine the maximum activation value for each sample
predicted_class_indices=np.argmax(pred,axis=1)
actual_labels=test_generator.labels
class_labels = list(test_generator.class_indices.keys())

# determine the test set accuracy
match=[]
for i in range(0, n_test):
    match.append(predicted_class_indices[i]==actual_labels[i])

acc = str(match.count(True) / n_test * 100)[0:4]
print(f"The model predicted accurately {acc}% of the samples")
print(classification_report(actual_labels, predicted_class_indices)
   )



# multiclass model

model = load_model("../models/model-4c-mel")

# generate data for test set of images
test_generator = data_generator.flow_from_directory(
        "/home/lorenzo/Pictures/test-dataset/4c-seg/",
        target_size=(128, 256),
```

```
        color_mode='grayscale',
        batch_size=1,
        class_mode='categorical',
        shuffle=False)

filenames=test_generator.filenames
n_test = len(filenames)

# obtain predicted activation values for the last dense layer
test_generator.reset()
pred=model.predict(test_generator, verbose=1, steps=n_test)

# determine the maximum activation value for each sample
predicted_class_indices=np.argmax(pred,axis=1)
actual_labels=test_generator.labels
class_labels = list(test_generator.class_indices.keys())

# determine the test set accuracy
match=[]
for i in range(0, n_test):
    match.append(predicted_class_indices[i]==actual_labels[i])

acc = str(match.count(True) / n_test * 100)[0:4]
print(f"The model predicted accurately {acc}% of the samples")
print(classification_report(actual_labels, predicted_class_indices)
    )



# plot confusion matrix

from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt

conf_matrix = confusion_matrix(y_true=actual_labels, y_pred=
    predicted_class_indices)

fig, ax = plt.subplots(figsize=(4.5, 4.5))
ax.matshow(conf_matrix, cmap=plt.cm.Blues, alpha=0.7)

for i in range(conf_matrix.shape[0]):
    for j in range(conf_matrix.shape[1]):
        ax.text(x=j, y=i,s=conf_matrix[i, j], va='center', ha='
    center', size='large')

plt.xlabel('Predicted labels', fontsize=14)
plt.ylabel('Actual labels', fontsize=14)
ax.xaxis.set_ticklabels(['', 'double', 'half', 'quad', 'quarter'])
ax.yaxis.set_ticklabels(['', 'double', 'half', 'quad', 'quarter'])
plt.title('Confusion Matrix', fontsize=14)
plt.savefig("confusion_matrix.png", dpi=300)
plt.show()
```

Bibliography

[1] A. Russo, M. Spanio, and S. Canazza, "Enhancing preservation and restoration of open reel audio tapes through computer vision," in *Image Analysis and Processing - ICIAP 2023 Workshops* (G. L. Foresti, A. Fusiello, and E. Hancock, eds.), (Cham), pp. 297–308, Springer Nature Switzerland, 2024.

[2] F. Bressan and S. Canazza, "A systemic approach to the preservation of audio documents: Methodology and software tools," *Journal of Electrical and Computer Engineering*, vol. 2013, 01 2013.

[3] N. Pretto, C. Fantozzi, E. Micheloni, V. Burini, and S. Canazza, "Computing methodologies supporting the preservation of electroacoustic music from analog magnetic tape," *Computer Music Journal*, vol. 42, no. 4, pp. 59–74, 2019.

[4] Y. Grava, "Progettazione e sviluppo di un software per il riconoscimento e la classificazione di discontinuità nei nastri magnetici audio, basato su tecniche di computer vision e neural network," *Master's degree Thesis, University of Padova*, 2019.

[5] M. Pignotti, "Riconoscimento e classificazione di discontinuità in nastri magnetici audio mediante reti neurali e tecniche di visione computazionale: un software innovativo basato su un esteso insieme di addestramento," *Master's degree Thesis, University of Padova*, 2020.

[6] A. Roda, N. Orio, L. Mion, and G. De Poli, "From audio to content," 01 2012.

[7] S. Prince, *Computer Vision: Models Learning and Inference*. Cambridge University Press, 2012.

[8] R. Szeliski, *Computer vision algorithms and applications*. Springer, 2011.

[9] D. O'Shaughnessy, *Speech Communication: Human and Machine*. Addison-Wesley, 1987.

[10] J. O. Smith, *Mathematics of the Discrete Fourier Transform (DFT)*. Stanford Education, accessed January 2024. online book, 2007 edition.

[11] Chris Bagwell et al., "Sox, sound exchange."

[12] B. McFee, M. McVicar, and D. F. et al., "librosa 0.10.1," Aug. 2023.

[13] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[14] Y. Zhou, *Sentiment classifcation with deep neural networks*. PhD thesis, 07 2019.

[15] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014.

[16] C. X. Ling and C. Li, "Data mining for direct marketing: Problems and solutions," in *Knowledge Discovery and Data Mining*, 1998.

[17] M. Spanio, "A study on equalization curve detection in audio tape digitization process using artificial intelligence," *Bachelor's degree Thesis, Ca' Foscari University of Venice*, 2022.

[18] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," 2014.