Applied speech emotion recognition on a serverless Cloud architecture

Reconocimiento de emociones de la voz aplicado sobre una arquitectura Cloud serverless

Por
Robert Farzan Rodríguez

UNIVERSIDAD COMPLUTENSE
MADRID

Trabajo de fin de grado del Grado en Ingeniería Informática
FACULTAD DE INFORMÁTICA

Dirigido por
José Luis Vázquez Poletti

MADRID, 2021–2022

Acknowledgements

I deeply thank my mother for the trust, the unconditional love and support and the opportunity to study and thrive. Also to my grandpa, for planting the seed of curiosity, knowledge, religion, perseverance and struggle, for teaching me to always choose the narrow path instead of the easiest way. To all the Salesian family, for teaching me to tell right from wrong and to pursue the good and the wisdom.

Abstract

The purpose of this final degree thesis *Applied speech emotion recognition on a serverless Cloud architecture* is to do research into emotion recognition on human voice through several techniques including audio signal processing and *deep learning* technologies to classify a certain emotion detected on a piece of audio, as well as finding ways to deploy this functionality on Cloud (*serverless*). From there we can get a brief implementation of a streaming nearly real-time system in which an end user could record audio and retrieve responses of the emotions continuously.

The idea intends to be a "emotion tracking system" that couples the technologies mentioned above along with a simple end-user GUI app that anyone could use purposefully to track their own voices in different situations - during a call, a meeting etc. - and get a brief summary visualization of their emotions across time with just a quick glance.

This prototype seems to be one of the first software products of its kind, as there is a lot of literature on the Internet on Speech Emotion Recognition and tools for software engineers to facilitate this task but an easy final user product or solution for real-time SER appears to be non-existent.

As a short summary of the project road map and the technologies involved, the process is as follows: development of a CNN model on Tensorflow 2.0 (with Python) to get emotion labels as output from a short chunk of audio as input; deployment of a Python script that uses this previously mentioned CNN model to return the emotion predictions in AWS Lambda (the Amazon service for *serverless* Cloud); and finally the design of a Python app with GUI integrated to send requests to the Lambda service and retrieve the responses with emotion predictions to present them with beautiful visualizations.

Keywords

Serverless, CNN, Artificial Intelligence, Cloud, Python, Tensorflow, AWS, Speech Emotion Recognition, GUI.

Resumen

El propósito de este TFG *Reconocimiento de emociones de la voz aplicado sobre una arquitectura Clous serverless* es investigar el reconocimiento de emociones en la voz humana usando diversas técnicas, entre las que se incluye el procesamiento de señal y *deep learning* para clasificar una cierta emoción en una pieza de audio, así como encontrar maneras de desplegar esta funcionalidad en el Cloud (*serverless*). A partir de estos pasos se podrá obtener una implementación de un sistema en *streaming* en tiempo cuasi real, en el que un usuario pueda grabarse a sí mismo y recibir respuestas cronológicas sobre su estado de ánimo continuamente.

Esta idea trata de ser un "sistema monitor de emociones", que envuelva las tecnologías mencionadas arriba junto con una simple interfaz gráfica de usuario que cualquiera pueda usar para monitorizar intencionadamente su voz en diferentes situaciones - durante una llamada, una reunión etc. - y obtener una breve visualización de sus emociones a lo largo del tiempo en un simple vistazo.

Este prototipo apunta a ser una de las primeras soluciones software de este tipo, ya que a pesar de haber mucha literatura en Internet acerca de *Speech Emotion Recognition* y herramientas para desarrolladores en esta tarea, parece no haber productos o soluciones de SER en tiempo real para usuarios.

Como breve resumen de la hoja de ruta del proyecto y las tecnologías involucradas, el proceso es el siguiente: desarrollo de una red neuronal convolucional en TensorFlow 2.0 (con Python) para predecir emociones a partir de una pieza de audio como *input*; despliegue de un script de Python que use la red neuronal para devolver predicciones en AWS Lambda (el servicio de Amazon para *serverless*); y finalmente el diseño de una aplicación final para usuario en Python que incluya una interfaz gráfica que se conecte con los servicios de Lambda y devuelva respuestas con las predicciones y haga visualizaciones a partir de ellas.

Palabras clave

Serverless, CNN, Inteligencia Artificial, Cloud, Python, Tensorflow, AWS, Speech Emotion Recognition, Interfaz Gráfica de Usuario (GUI).

Sobre TEFLON X

Esta plantilla fue desarrollada para facilitar la creación de documentación profesional para Trabajos de Fin de Grado, Trabajos de Fin de Máster o Doctorados. La versión usada es la X

V:X OVERLEAF V2 WITH XELATEX, MARGIN 1IN, BIB

CONTACTO

AUTOR: DAVID PACIOS IZQUIERO

CORREO: `dpacios@ucm.es`

ASCII: `ascii@ucm.es`

DESPACHO 110 - FACULTAD DE INFORMÁTICA

Contents

Chapter 1

Introduction

This project intends to be a software solution to analyse audio data from recordings and retrieve the emotion associated from that data to present it on a final application. The development of this solution involves several disciplines such as AI, Cloud and Software Engineering coupled together.

1.1 Motivation

In current times there has been an increasing concern on mental health issues and the technology involvement in such matters. With the appearance of new devices such as home virtual assistants and other IoT devices, as well as smartphones, it is widely known that a vast amount of data - now focused on biometric data as voice or facial recognition - is being collected from us. Some of this data is just used to interact with the devices (e.g. a user giving a command to Alexa) and some other data can be used for further analysing with commercial or profiling purposes.

The motivation of this project lies on the novelty of using a distributed architecture such as *serverless* to deploy a *deep learning* model, and the advantages it could carry with the use of Cloud, such as modularity, isolation (the deployment of code on containers), scalability and compatibility.

Currently there is no available commercial software (at least public or well-known software) that uses voice data for the sake of measuring some person's mental state by analysing the emotions in it. However, there is a lot of literature on speech recognition techniques with *deep learning* methods such as CNNs that can help us to develop such solution. Just as a person can keep track of their burnt calories or a step counter in real-time, it would be interesting if they could also be able to measure their current emotional status in real-time.

1.2 Goals

The principal aim of this project is to provide a software prototype (desktop application) that can record a person's voice from an input channel and extract and show the emotions derived from that input. To do so, the project tasks can be fragmented into three parts and its subsequent subparts:

- **Speech Emotion Recognition model design**: develop a *deep learning* model with high accuracy to analyze audio data and extract emotions from it.
- **Deployment**: upload the model to a *serverless* Cloud service to be able to get access remotely and make calls to that service that will return the emotion value.
- **Final app**: develop a simple app with GUI that a final user can use intuitively to record its voice and visualize the emotions.

1.3 Project plan

The plan for this project is organized in three parts that correspond to the three modules that compose this project:

1. **Deep learning model** (01 September - 01 November)
   - Explore Speech Emotion Recognition state of the art and techniques used.
   - Search for data sources and public datasets and assess their suitability for the project.
   - Search common features for audio processing and how to preprocess it.
   - Build demos and prototypes using the features and models present in literature.
   - Fine-tune the model until a final optimal version is found.
2. **Cloud architecture** (01 November - 01 February)
   - Explore Cloud *serverless* providers and assess costs.
   - Develop a script that embeds the SER model to be used in the provider's main function.
   - Build the architecture that supports the function, that may include other Cloud provider's services.
3. **User application** (01 February - 01 April)
   - Develop a prototype script that connects to the Cloud architecture.
   - Develop a prototype script for recording voice from microphone and slicing it.
   - Join the prototypes into one single console script.
   - Build a GUI upon the console script using the MVC pattern to plot the emotions over a chronogram.

Chapter 2

State of the art

In this chapter some concepts that are discussed in the project are going to be explained and detailed in different sections. We are going to explore some already existing solutions as well as literature about each topic, including CNNs, Speech Emotion Recognition and Serverless computing.

2.1 Convolutional Neural Networks

Convolutional Neural Networks are one of the most famous pillars that form Computer Vision techniques. Computer vision is a field of artificial intelligence (AI) that allows computers and systems to extract relevant information from images, videos and other visual inputs and take actions or make recommendations based on the information[1].

CNNs belong to the specific area of *Deep Learning*, which is a subset of Machine Learning (ML) which is, in turn, a subset of artificial intelligence (AI) [25]. Deep Learning algorithms, as Machine Learning ones, need tons of data to figure out proper approximations to solve a specific problem, but the former are known to perform incredibly well with some tasks that ML could not do so, such as Computer Vision.

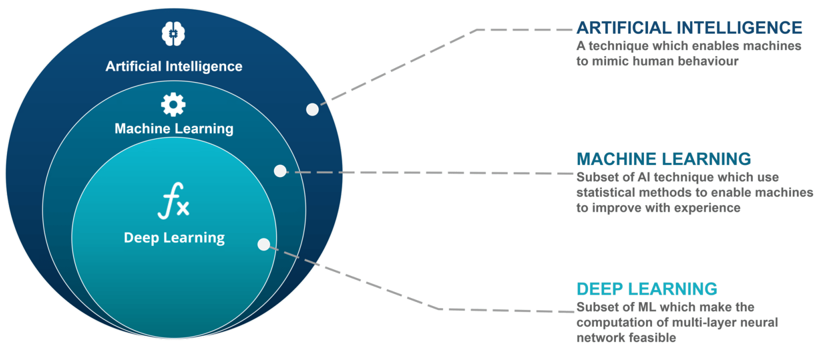

Artificial Intelligence
Machine Learning
Deep Learning
ARTIFICIAL INTELLIGENCE
A technique which enables machines to mimic human behaviour
MACHINE LEARNING
Subset of AI technique which use statistical methods to enable machines to improve with experience
DEEP LEARNING
Subset of ML which make the computation of multi-layer neural network feasible

Figure 2.1: Deep Learning in the scope of AI and ML [3]

[1]https://www.ibm.com/topics/computer-vision

Convolutional Neural Networks are composed of two main parts: *convolutional layers* and *fully-connected layers*. The former ones are responsible for taking the input images and shrinking them through convolution and pooling operations to extract the most important features of the image. The latter are common dense layers usually seen in Artificial Neural Networks, with the purpose of classifying the flattened image output of the convolutional layers.

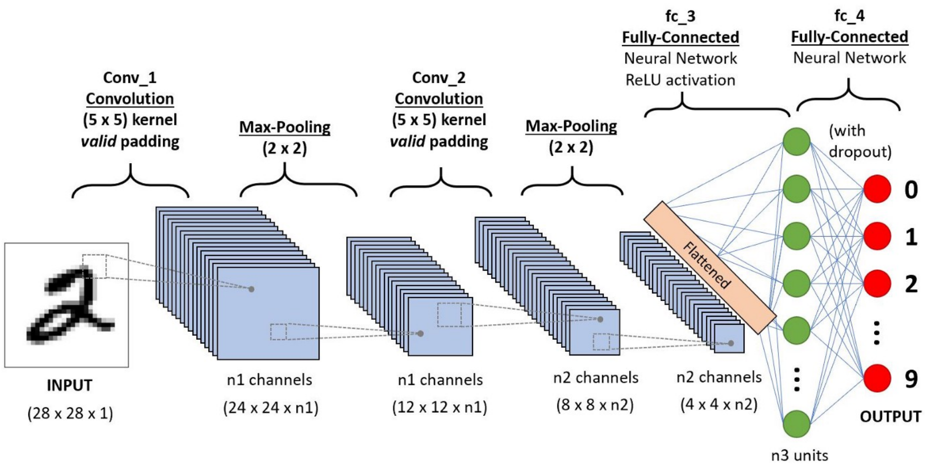


Figure 2.2: A CNN sequence to classify handwritten digits [21]

In turn, *convolutional layers* are composed of two operations: convolutions and pooling.

- **Convolutions**: the convolution operation is performed on the input data through a filter or kernel (these terms are used interchangeably) to build a feature map. We do a convolution by sliding the filter over the input. At every point, it computes an element-wise matrix multiplication and sums the result into the feature map [7]. As shown in Figure 2.3, every element in the kernel matrix is multiplied by the elements of the patch of the same size and then summed up and placed in the corresponding pixel, resulting in a new matrix of lower dimensions.

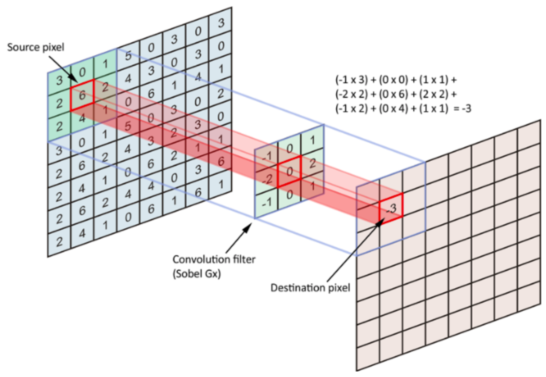

Source pixel
(-1 x 3) + (0 x 0) + (1 x 1) +
(-2 x 2) + (0 x 6) + (2 x 2) +
(-1 x 2) + (0 x 4) + (1 x 1) = -3
Convolution filter
(Sobel Gx)
Destination pixel

Figure 2.3: Convolution operation with a 3x3 kernel [7]

- **Pooling**: the pooling operation reduces the size of the feature maps. It is useful to decrease the computational power required to process the convolved maps. Moreover, it is also useful for picking dominant features (like a dimensionality reduction technique) [21].
  - There are two types of pooling: *Max Pooling* and *Average Pooling.* Max Pooling returns the maximum value from the portion of the image covered by the chosen size for pooling whereas Average Pooling returns the average of all the values from the portion.

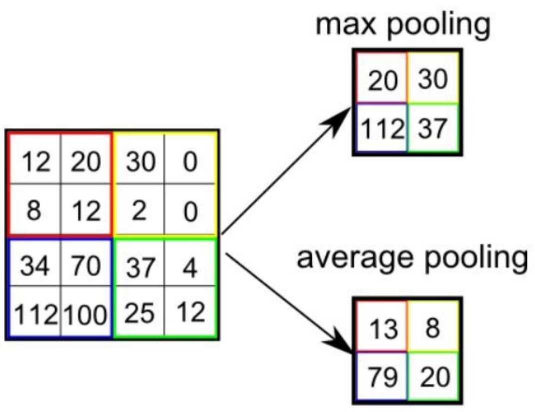


Figure 2.4: *Max Pooling* and *Average Pooling* [21]

After the *convolutional layers*, there come the *fully-connected layers.* As stated before, they are capable of taking the flattened image output of the convolutional layers as input and learn high-level features to serve as the classification part of the network. Through common dense layers and weights, it will be able to classify each image to a label through the Soft-Max activation function in the output neurons (the red neurons in Figure 2.2).

Thus, each neuron in the output represents the probability of the image of belonging to one of these labels, and all the probabilities of each one sum up to 1. The neuron with the highest probability is considered to be the *final prediction*, that will then be compared to the actual label in order to calculate the error function and adjust the weights based on that error through *backpropagation.*

Audio processing with CNNs

It is clear that CNNs are great at extracting image features and making predictions based on them, but its functionality is not limited to images. CNNs can be further used on other fields such as NLP for sentence classification using 1D Convolutions [15] and audio processing.

As CNNs are flexible and can be applied on multiple dimensions, they are also suitable for its use on audio processing. In the end, as we have seen before, convolutions are just operations performed on matrices, such as element-wise multiplication and dot products. Images are just a composition of matrices (a 3D matrix) for each color channel in RGB (Red-Green-Blue), each of them which have values in the range $[0, 255]$, as we can see in the below Figure 2.5

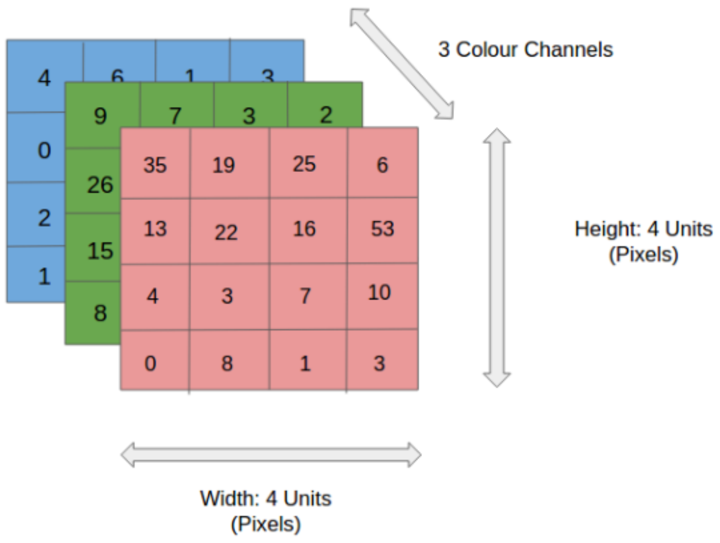


Figure 2.5: RGB representation of an image [21]

How is this related to audio? As CNNs expect images that are just 2D arrays of single or multiple channels, it is possible to extract features from audio and transform them into images in order to feed the CNN [18]. Both 1D and 2D CNNs can perform audio classification tasks as we will see next.

1D CNNs Audio signals can be treated directly as an input for this one-dimensional networks. Raw audio data captured from sound devices is most of the times a function of sound pressure variation over the time domain. From this raw audio data the usual features of sound can be extracted: sound intensity (dB), frequency (Hz) and so forth.

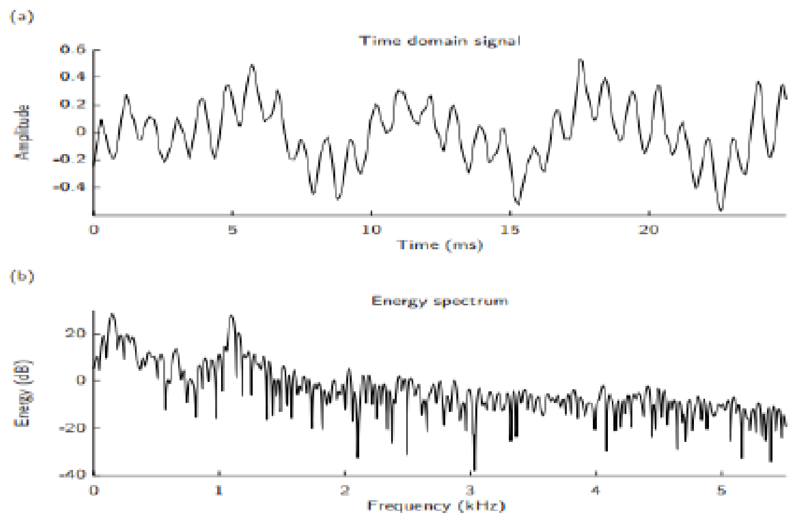


Figure 2.6: Representation of raw audio data (a) and its energy spectrum (b) [9]

The idea that lies behind these 1D CNNs is to store these audio samples as one-dimensional *arrays* to feed the neural network that will extract the features. One interesting approach can be observed in the paper "1D CNN Architectures for Music Genre Classification" [2] that builds a model for classifying music genres from raw audio input and pass a sliding window through it to get several predictions from the 1D-CNN and aggregate all the outputs into a single final prediction as shown in Figure 2.7

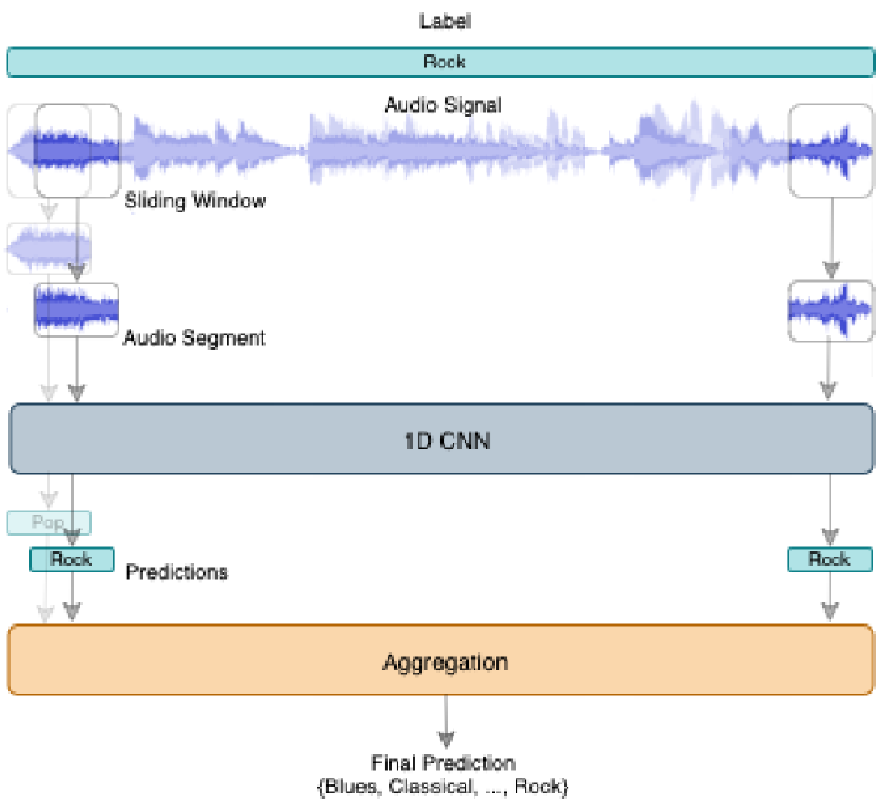


Figure 2.7: 1D-CNN example for music genre classification [2]

2D CNNs As shown before, this is one of the most common ways to build a convolutional model. In this case, the challenge relies upon finding a way to convert the 1D raw audio signals into an image shape in order to feed the network. Fortunately, there are multiple features in the field of audio signal processing that are represented the expected way. They can be extracted by playing with audio parameters such as frequency, time and amplitude. A couple of these features are to be described next:

- **Spectrograms**: A spectrogram is a visual way of representing the strength of a signal over time and at different frequencies. It is represented as a two dimensional graph, in which the third dimension (strength) is represented by colors. The energy of the signal varies horizontally along the time axis and vertically along the frequency axis[2].

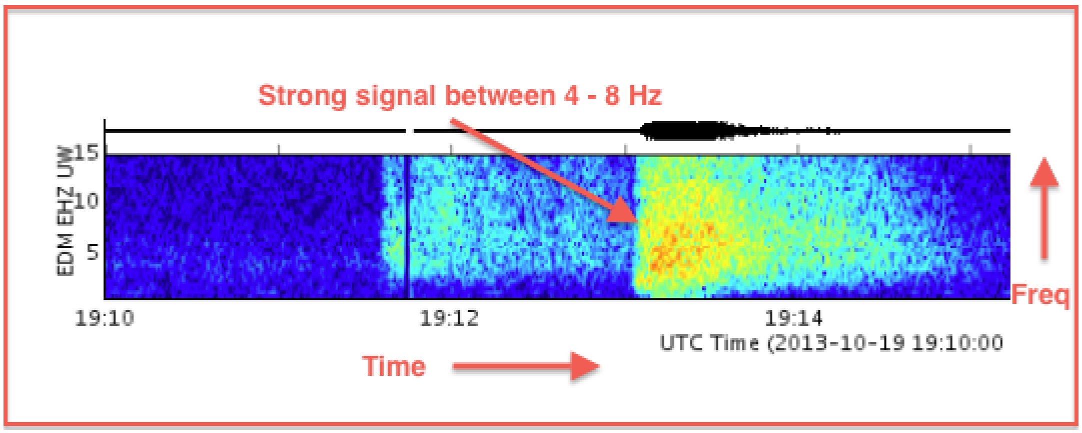


Figure 2.8: Spectrogram representation of raw audio signals[2]

We can clearly observe in Figure 2.8 that a spectrogram is simply an image with two dimensions and RGB color channels as shown previously, which is the expected outcome one would want to feed a 2D-CNN.

- **MFCCs**: Mel-frequency cepstral coefficients, also known as MFCC is a very well-known technique for extracting features from audio signals. MFCCs are capable of representing phonemes in the human voice, that is why they are widely used in Speech Recognition tasks and literature.

  MFCCs generation is a little more complex and less straightforward than spectrograms, but to put it simply, it is about taking the original "spectrogram", slicing it into multiple subparts and pass each slice through a series of mathematical transformations in order to get the Cepstral coefficients. These coefficients effectively manage to represent the shape of the vocal tract generated by humans [17].

[2]https://pnsn.org/spectrograms/what-is-a-spectrogram

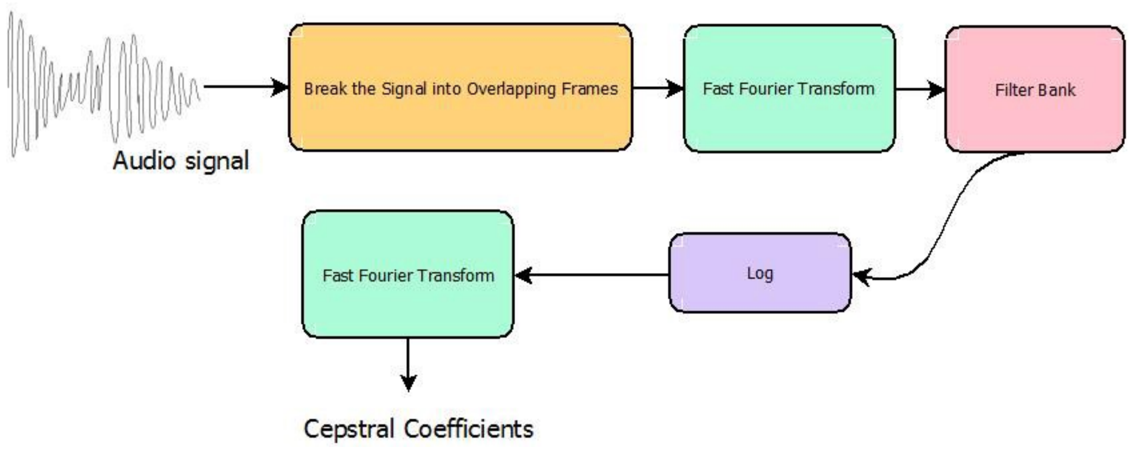


Figure 2.9: Steps to generate MFCCs out of raw audio signal [17]

The transformations shown in the diagram above will not be explained in detail, but we can get a grasp on what each step does: slicing the audio signal, applying FFT to the broken signals, Filter Bank applies Mel-Scale (thus generating a different kind of spectrogram, called the Mel-spectrogram), then this Mel-spectrogram is applied to a logarithm function to scale it, and finally another Fast Fourier Transform. The final desired outcome these operations is again an image representable in RGB channels as shown below:

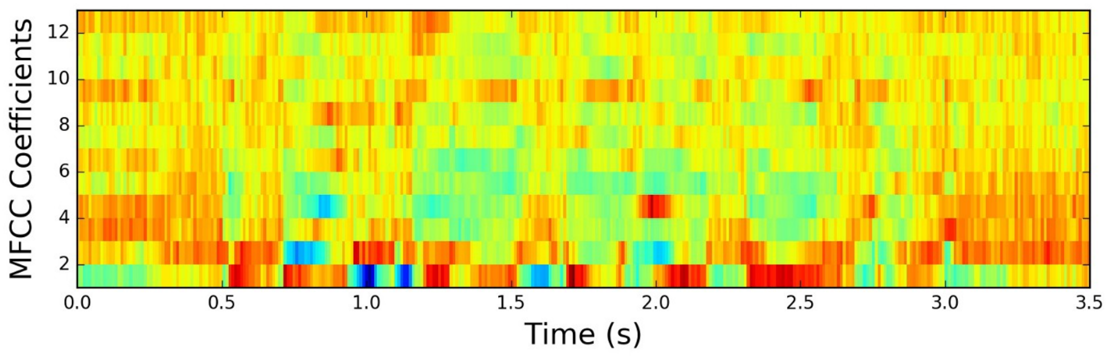


Figure 2.10: MFCC final outcome [17]

- **Other features**: Some other famous and widely used features in speech tasks that will be named but not detailed in this paper are: Short-Time Fourier Transform (STFT), Chromagram, Spectral Centroid and Zero Crossing Rate.

2.2 Speech Emotion Recognition

Speech Emotion Recognition (SER) is the task of identifying the emotional aspects of speech excluding the contents of the speech (it does not recognize words or phrases, just sounds). Whereas humans perform this task naturally as a part of communication, the idea of machines and devices doing it automatically is still a subject of research [13].

Although conventional SER approaches were done with ML approaches such as Support Vector Machine(SVM), Gaussian Mixture Model (GMM), and shallow Neural Networks (NNs), this project focuses on a Deep Learning (DL) approach with Deep Convnets (DCNNs). Supervised *deep learning* models have proved to exceed classical approaches in a series of classification tasks, inside which the image classification stands out for its successfulness [13].

Real-Time SER with CNNs

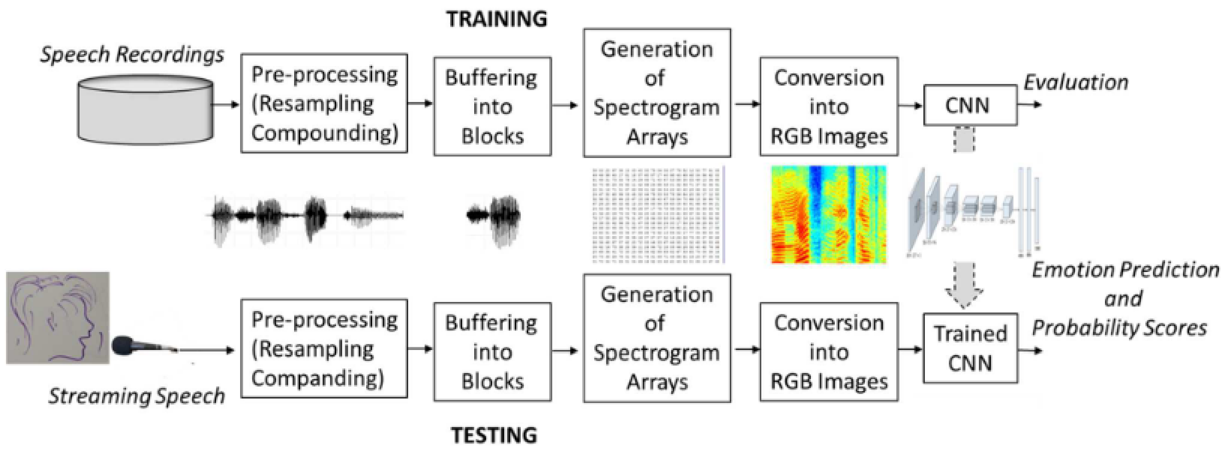

TRAINING
Speech Recordings
Pre-processing (Resampling Compounding)
Buffering into Blocks
Generation of Spectrogram Arrays
Conversion into RGB Images
CNN
Evaluation
Emotion Prediction and Probability Scores
Streaming Speech
Pre-processing (Resampling Companding)
Buffering into Blocks
Generation of Spectrogram Arrays
Conversion into RGB Images
Trained CNN
TESTING

Figure 2.11: Streaming (real-time) SER model workflow [13]

The current project aims to implement a nearly real-time Speech Emotion Recognition (SER) model to build an application on top of this model. The model workflow that represents the expected model is similar to the one depicted in Figure 2.11. For the use of CNNs for the task of real-time SER, this approximation seen in literature takes the following steps:

a). During training, the audio samples along with their corresponding emotion label (e.g. the emotion could be appended to the filename: "audio001_sadness.wav") are preprocessed through some steps that were explored in the previous section, such as converting the audio data to different 2D features as spectrograms.

b). These spectrograms are fed into the convnet to train its parameters. The output layer of the CNN will have the same number of neurons as the number of emotions present in the dataset. Each output neuron will represent a number between 0 and 1 which is the probability of that emotion in the audio. Then the most probable emotion is picked as the prediction to evaluate and tune the model.

c). Once the CNN model is trained and fine-tuned, it can be used to make predictions on streaming speech. When audio signal is received from an input audio, this signal is chopped into slices of a certain size to match the convnet input shape (which must be always the same). Afterwards, the sliced audio pieces pass through the exact same preprocessing steps as the training audio samples, and it is further passed to the CNN to make the final prediction.

2.3 Serverless computing

Serverless is a Cloud development service that allows developers to build and run applications without the need of managing servers and resources.

The term *serverless* is not literal, as there are servers involved in serverless, but their management is abstracted from the development process. The Cloud provider is responsible for handling and providing the server infrastructure that lies behind the process, whereas developers just simply deploy their code packaged in containers.

When deployed, serverless apps automatically scale as demand grows or decays. These apps are often executed by an event-driven execution, meaning when the function is idle (not being called), the cost is zero[3].

2.3.1 Attributes of serverless

Serverless architectures are characterized by the following features[4]:

1. **Small units of code**. Serverless architectures' functionality is often packaged in a single function.

2. **Event-driven execution** The infrastructure that supports the function execution is created when the function is called. When an event is received a little execution environment is created for the purpose of serving that request. This environment can persist for a little period of time to attend more events, commonly between 5/10 minutes.

3. **Scale to zero**. As mentioned before, when the function is inactive for a short period, the infrastructure and execution environment are taken down. This saves lots of costs, and for that reason users only pay for usage time.

4. **Autoscale**. The FaaS 2.3.2 service auto-scales the infrastructure adding additional instances when demand grows. This relieves developers from the task of thinking about scaling when designing the app.

5. **Other services**. Apart from FaaS functions, serverless architectures offer another type of services for different purposes in applications such as file storage, database management, queueing and more. This is called BaaS 2.3.3.

2.3.2 Function-as-a-Service (FaaS)

Function-as-a-Service (FaaS) is the serverless way to execute code on Cloud. It allows developers to write and update code easily, code that is then executed in response to events, such as calls from APIs or triggers. It is easily scalable and a *low-cost* implementation[5].

[3]https://www.redhat.com/en/topics/cloud-native-apps/what-is-serverless
[4]https://about.gitlab.com/topics/serverless/
[5]https://www.cloudflare.com/en-gb/learning/serverless/glossary/function-as-a-service-faas/

The logic developed by programmers is then deployed into containers that are managed by a Cloud provider. These containers share the following traits[6]:

- Stateless, which makes integration easier.
- Short-lived functions, that are executed for a very short period of time.
- Event-driven, they run automatically in response to events.
- Fully managed by a Cloud provider, so developers pay for use (execution time).

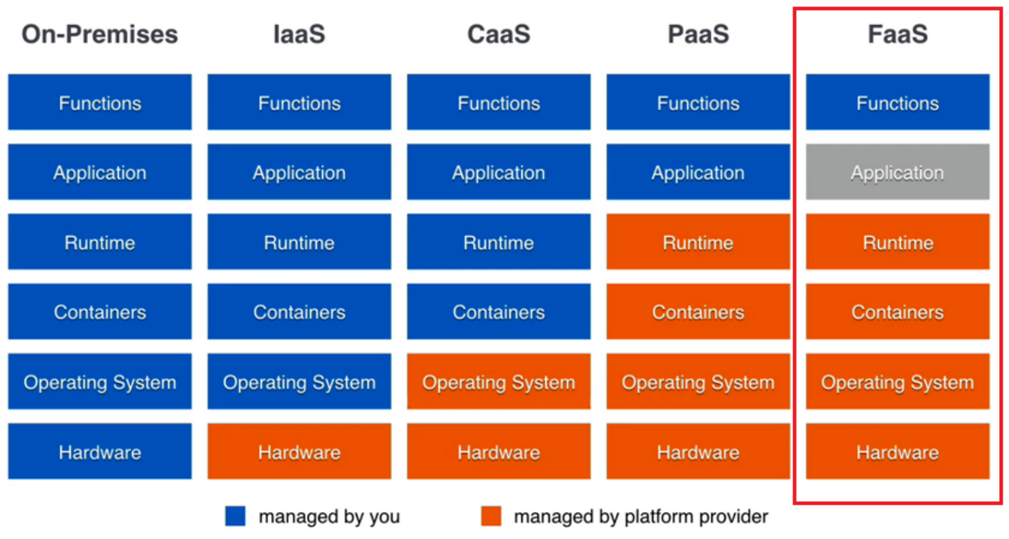


Figure 2.12: FaaS compared to other Cloud models [16]

As shown in the chart above, FaaS goes one step beyond the rest of the Cloud models, letting the Cloud provider manage almost all the resources way up to the application. At this point, the only thing the programmer must do is build the function that will be deployed in a container and let the provider manage all the rest, from infrastructure matters to application builds and running. FaaS is the middle step between PaaS (Platform-as-a-service) made for developers, and SaaS (Software-as-a-Service) which is fully user-oriented.

2.3.3 Backend-as-a-Service (BaaS)

Although this project is FaaS-oriented, it is useful to know it is not the only serverless architecture publicly known. Backend-as-a-Service (BaaS) is a Cloud service in which developers can delegate several parts that involve web or mobile development tasks so that they only have to design and maintain the frontend. BaaS providers use pre-written software for such tasks as authentication, database management, notification in mobile apps, storage and hosting[7]

[6]See footnote 3
[7] https://www.cloudflare.com/en-gb/learning/serverless/glossary/backend-as-a-service-baas/

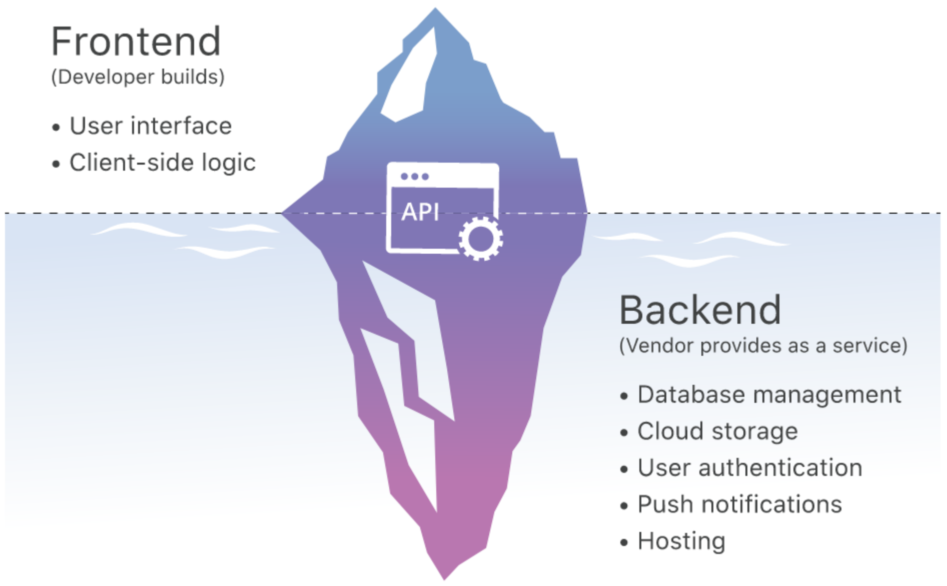


Figure 2.13: Frontend vs Backend services[8]

With the power of BaaS services, developers are only required to write the frontend app (the tip of the iceberg in Figure 2.13), which is usually formed by the client code and UI. Then, the frontend built by developers is fully-integrated with the backend via APIs and SDKs that are also provided by the Cloud provider. It is neither necessary to manage servers, containers or VMs to keep the application functioning.

2.3.4 Cloud providers

Almost every top Cloud provider nowadays have some serverless or FaaS service open to the public. Usually all these services differ in matters as billing conditions, internal infrastructure and the APIs, SDKs and interfaces used to communicate with the provider's services. Just to name some examples of the most famous FaaS services:

Cloud provider	**Service**
AWS	Lambda
Google Cloud	Cloud Functions
Microsoft Azure	Azure Functions
IBM	IBM Cloud Functions

Table 2.1: FaaS most famous Cloud providers

[8]See footnote 7

Chapter 3

System Architecture

3.1 Cloud architecture

As previously seen in section 3.4, this project relies on Amazon Web Services for the development of its architecture. In this section the whole structure of the interconnected Cloud services is to be described, as well as the function of each part of the whole system.

3.1.1 AWS diagram

First of all, we will observe the overall system architecture on the following simple diagram describing each functioning service and its relation with the other parts. Then we will be detailing each component and its purpose on subsequent sections.

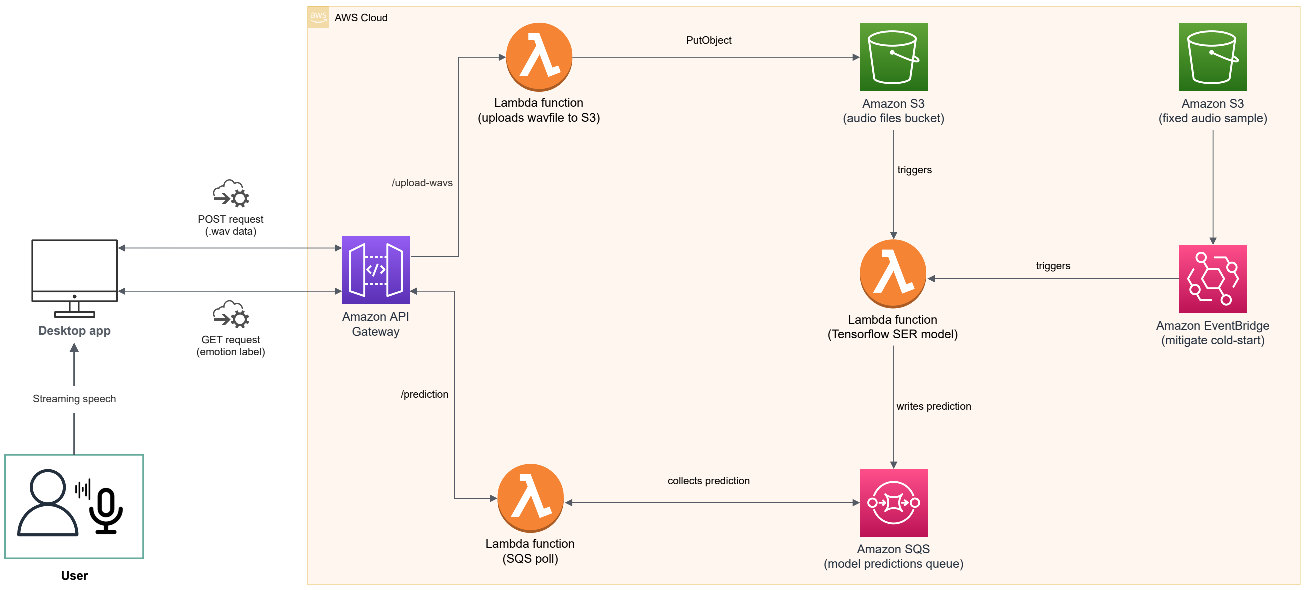


Figure 3.1: Descriptive AWS architecture diagram

3.1.2 Workflow description

The detailed step-by-step description of the process shown in Figure 3.1 is summarized here:

1. First, users must set up a microphone as input device and run the client-side desktop app in their computers. From there on, the program **records the user's voice and stores the audio into 2-second temporary wav files** that will be deleted after the program execution.

2. Each generated audio file is sent *online* through a POST request to an **open Amazon API Gateway endpoint** (/upload_wavs in this case). Amazon API Gateway serves as an API REST that communicates and sends or retrieves data from the other services. Further detail is provided in the next section.

3. When audio data is received through POST request to API Gateway, it is **forwarded to a Lambda function** whose purpose is to store **this wav file into a Amazon S3 bucket temporarily** to make it accesible to the Tensorflow model deployed in another Lambda function.

4. The upload to the S3 bucket **causes a trigger to the AI model Lambda function**. This function is responsible for reading, preprocessing and making a prediction out of the audio data that has been uploaded to the bucket. Further detail about the model is provided in section 3.3.

5. **The final prediction** of the model (a label containing one of the following results: sadness, disgust, fear, anger, neutral, surprise or happiness) **is written to an Amazon SQS queue**. This prediction is pushed into the queue in order to be read by another process. Further detail about SQS can be found in the next section.

6. Parallel to the process that sends audio data to API Gateway, another process is sending continuous GET requests to another API Gateway endpoint every 2 seconds. This second endpoint is responsible for **retrieving the predictions from the SQS queue**. To do that, it delegates its function to another Lambda function that *polls* the queue every time it receives a request from the API. The queue then responds with the first message in the head of the queue (the oldest message in a FIFO structure), and this prediction is forwarded to the API than in turn forwards it to the user app.

7. A final element is set to mitigate or reduce the effects of *cold start*. An Amazon EventBridge rule is set to send a custom audio file from a separate bucket every a certain amount of time to *wake up* the model function periodically and allow the users to wait less time when running the application. The cold start topic is covered in section 3.4.3.1.

3.1.3 Detailed parts

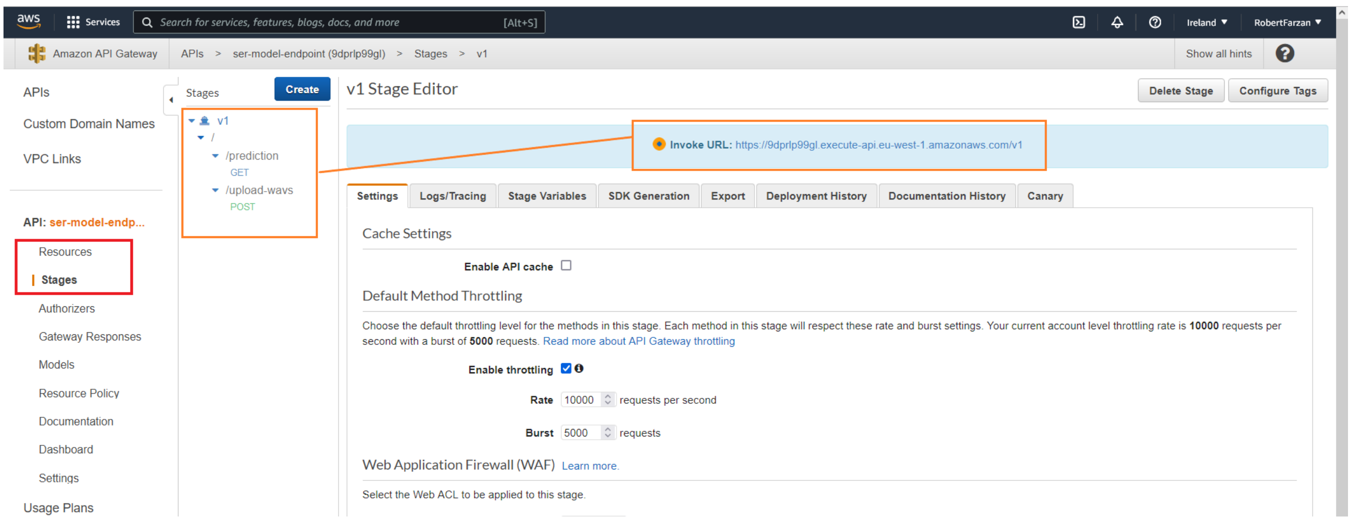
API Gateway endpoints

The project uses API Gateway as a way to communicate the user application and allow it to interact with the rest of the services of AWS without the need for authentication or Amazon credentials. If the user needed to interact with the Lambda functions directly, it would have to use some library that would require credentials such as `boto3` in Python, thus requiring to create a specific IAM Role and IAM User for each user running the program, which complicates its usability.

One single REST API with **two endpoints** was created for this project:

- `/prediction`: This is an endpoint that only allows GET requests. It is used to make periodic calls to retrieve elements from the SQS Queue of emotion labels. When the queue is empty, the associated Lambda function invoked by the endpoint returns `204 No Content`. Otherwise, it returns `200 OK` with the corresponding prediction emotion label.
- `/upload-wavs`: This endpoint serves as the gateway to send our audio data through only POST requests. When a 2-second piece of audio is recorded, its binary data is embedded in the request and the endpoint is called. Then, the gateway forwards this data to a Lambda function responsible for updating it to a S3 Bucket.

These two endpoints are defined as **resources** for API Gateway. To deploy the actual API for its use, a **stage** with these resources is deployed, in this case `v1`. After that, an invocation URL is provided for its use, as shown in Figure 3.2.



v1 Stage Editor

Invoke URL: https://9dprlp99gl.execute-api.eu-west-1.amazonaws.com/v1

Cache Settings

Enable API cache ☐

Default Method Throttling

Choose the default throttling level for the methods in this stage. Each method in this stage will respect these rate and burst settings. Your current account level throttling rate is **10000** requests per second with a burst of **5000** requests. Read more about API Gateway throttling

Enable throttling ☑

Rate 10000 requests per second

Burst 5000 requests

Web Application Firewall (WAF) Learn more.

Select the Web ACL to be applied to this stage.

Figure 3.2: API Gateway console example

Although no authentication is required, several **security measures** have been taken to protect the architecture from attacks such as DDoS or too many requests that may end up causing great costs in the monthly bill:

1. Creation of an **API Key** to allow the requests calls to the API. Without the key included in the headers of the request, when you invoke the endpoint URL an HTTP `403 Forbidden` is retrieved, thus the request is blocked and doesn't trigger invocations to Lambda. The API Key value is only provided to the users of the program, making it impossible for attackers to make DoS or DDoS attacks to the API.

2. **Enable throttling** to limit the rate of requests to the API. As shown in Figure 3.2, rate is limited to 10,000 requests per second, avoiding the possibility of attacks based on *flooding* of requests.

3. Configuration of an **usage plan**. The usage plan allows us to limit rate and burst of requests for our endpoints. It also allows to set up a maximum quota of requests per month and user to our API, so when one exhausts the monthly requests, one must wait until the next month to use the API.

4. **Restriction of data formats in POST requests** to only formats related with audio such as wav or x-wav to avoid possible poisoned binaries or exploits.

S3 upload

The main bucket that stores the generated WAV files is configured to trigger the main Lambda function on all object create events. It is also configured to only allow .wav file format to avoid possible anomalous or erratic behaviour.

As **security and privacy measures** some rules have been applied:

1. **Data Pseudonymisation** is applied to every piece of audio. For the sake of **digital privacy protection**, the names of the stored audio files are generated with **hashes**, not containing any information about the user using running the application or the origin of the audio. It also doesn't include any ordering ID of the audio sequence, so it would be very difficult to put together the sliced pieces and reconstruct the original complete recording. This way the anonimity of the users is guaranteed, even if someone managed to steal the data from the bucket.
   (a) **Note:** data anonymization could not be applied directly on the audio data, for instance by applying distortion or noise into the voice, because it would alter the predictions' performance.

2. Through the S3 Management menu we have defined two **lifecycle rules** that enable auto-deletion of elements in the bucket after 1 day:
   (a) The first rule applies to all objects in the bucket and sets an expiry date exactly 24 hours after its creation (1 day is the minimum granularity).
   (b) The second rule auto-deletes all the items that have expired.

3. **All public access is blocked** in the bucket, including the bucket name, elements and proprietary.

The European General Data Protection Regulation (GDPR) defines **pseudonymisation** as *the processing of personal data in such a manner that the personal data can no longer be attributed to a specific data subject without the use of additional information, provided that such additional information is kept separately and is subject to technical and organisational measures to ensure that the personal data are not attributed to an identified or identifiable natural person*, from Article 4(5) (Definitions) of the GDPR [8]

It is also worth to mention that the second rule of **auto-deletion** of information after 1 day complies with the Article 5(1)(e) of the GDPR which states that *Personal data shall be kept in a form which permits identification of data subjects for no longer than is necessary for the purposes for which the personal data are processed; personal data may be stored for longer periods insofar as the personal data will be processed solely for archiving purposes in the public interest, scientific or historical research purposes or statistical purposes in accordance with Article 89(1) subject to implementation of the appropriate technical and organisational measures required by this Regulation in order to safeguard the rights and freedoms of the data subject ('storage limitation');* [8]

SQS polling

The SQS queue of this project is configured as a **Standard Queue**. The characteristics of Standard Queues are defined in section 3.4.3.5.

For the sake of ordering the timeline that shows emotions that the user sees, it would have been more suitable to use a FIFO Queue, as it has First-In-First-Out Delivery and doesn't allow duplicates in the delivery of a message, but the reason not to choose a FIFO Queue is that it isn't yet available to use as a Lambda Destination, only Standard Queues are available for that purpose.

This means that, because the Best-Effort Ordering implemented in Standard Queues, there is a little chance of getting some results unordered in time, but the chance of getting duplicates is completely removed by the implementation of the Lambda function that retrieves the elements from the queue. The Lambda function makes sure that every message read from the queue is deleted after its use.

It is also good to remark that the method used for polling the queue is **Short Polling**. Short Polling means that there is no wait time when requesting an element from the queue, if the queue is empty, it returns immediately without waiting for a message. More about long and short polling can be read in the Amazon information page[1]

Main Lambda function and EventBridge

The "main" Lambda function to which the whole architecture revolves around is the function responsible for receiving an audio sample and running all the *pipeline* to get a prediction. This includes tasks that were performed to train the CNN model such as pre-processing, feeding the neural network and post-processing.

This function carries great costs of overhead in time when running **cold-start** for several reasons:

[1]https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-short-and-long-polling.html

- The **pre-processing phase** uses parameters that are not hard-coded but instead stored as files in another S3 bucket, such as the mean and standard deviation of the data for **data normalization**; a dictionary generated with a LabelEncoder, containing the correspondence between an integer number and the emotion label associated with it; and the Speech Emotion Recognition model itself, whose neurons, parameters and weights are stored in one HDF5 file and another JSON file with the structure of the neural net. The latency of the I/O operations and network operations between the function and the bucket causes a little overhead.

- The function, as shown in section 3.4.3.2, is deployed on a Docker container that is uploaded to Amazon ECR. As stated in the cold start section 3.4.3.1, when there is a heavyweight backend or dependency **the time of cold-start increases**. Unfortunately, TensorFlow 2.0 is a very heavywewight library, which can take up to approximately 1.1GB of storage. Also TensorFlow is a backend on its own, so when Lambda starts this backend **it can take up as much as 20 seconds**.

A waiting time of approximately 20 seconds is unacceptable for any application to start functioning. As no user would desire to use an application with such overhead, it must be mitigated somehow. **The way cold start times are mitigated is by setting an EventBridge rule**.

As shown in section 3.4.3.6, Amazon EventBridge allows us to set rules that can **create events**. As Lambda functions process events, an EventBridge rule can be set to send events to our Lambda function. This rules can be set to send an event on a specific hour every day or on a fixed rate (for instance 10 minutes). Our rule sends a constant input that is only an audio sample from the dataset previously used for training the model. This WAV file is stored on a separate bucket.

This way, if we set this event to be sent every 5 minutes, it will trigger the "cold start" on a successful invocation, thus allowing the user that runs the application to skip this waiting time, and start the application on a *warm* state of the execution environment.

3.2 Client-side application architecture

The figure below shows a sequence diagram representing the workflow of the software developed for end-users that comes with a GUI and allows to use the Speech Emotion Recognition model deployed in Cloud in a streaming fashion.

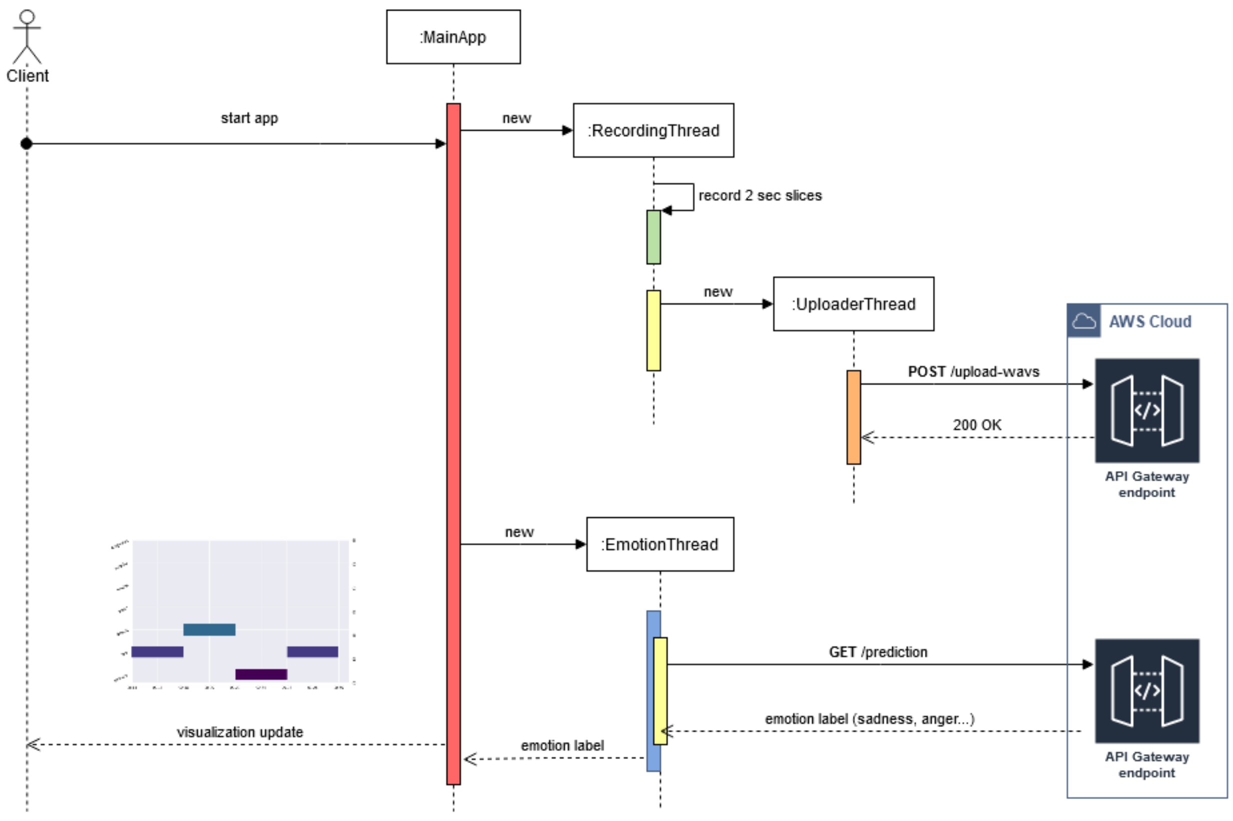


Figure 3.3: Sequence diagram of the end-user application

The steps represented in the figure go as follows:

1. When the client opens the application and presses the 'Start' button, **the main process creates two threads**: the RecordingThread and the EmotionThread. While these two threads are running, the main process is responsible for showing the GUI.

2. The RecordingThread is responsible for the continuous recording of audio through PyAudio. Every 2 seconds of recording, it stores the chunk in a **temporary .wav file that is deleted after the program execution**.

3. After the creation of a WAV file, an UploaderThread is created. This thread is necessary for uploading the WAV file to API Gateway with `POST /upload-wavs` **asynchronously**. This is because if the RecordingThread was in charge of the upload, it would have to interrupt the recording until the upload completion, which is not a desirable behaviour (synchronous). Instead, it delegates this functionality cre-

ating an UploaderThread and not interrupting the recorder, making the behaviour asynchronous.

4. Parallel to the RecordingThread execution, the EmotionThread is in charge of polling for emotion label responses from Amazon. This thread makes periodic calls every 2 seconds to the other API Gateway endpoint with `GET /prediction`. As explained in the AWS workflow section 3.1.2, these calls trigger polling to the SQS queue that stores the emotion predictions from the audios that are sent through the UploaderThread.

5. When polling with the EmotionThread, when an emotion prediction is retrieved, then the visualization Matplotlib chart in the PyQT5 GUI is updated (as displayed in Figure 3.4), and the user can watch the evolution of the emotions through a timeline in a **streaming fashion** (that is to say, while the user is talking, he is receiving feedback simultaneously).

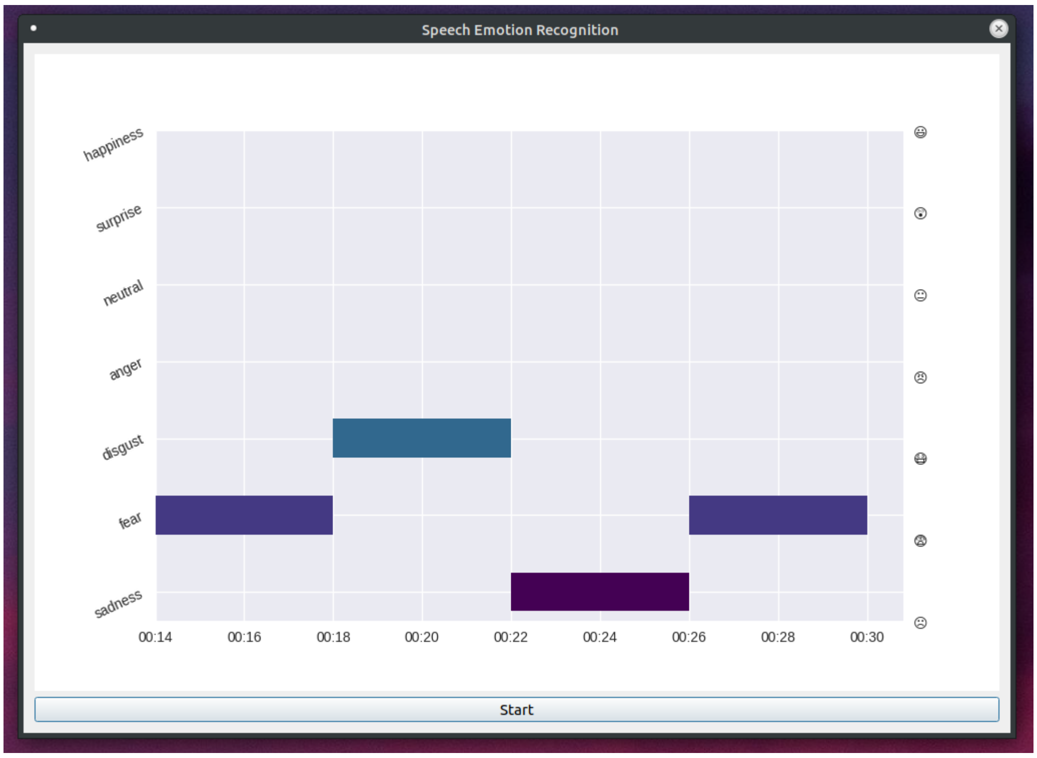


Figure 3.4: Demo of the application GUI

3.3 Speech Emotion Recognition model

3.3.1 Datasets

The proposed Speech Recognition model of this project has been trained and evaluated over two different datasets that have been merged: the **RAVDESS** (Ryerson Audio-Visual Database of Emotional Speech and Song) [14] dataset and the CREMA-D (Crowd-Sourced Emotional Multimodal Actors Dataset) [5] dataset.

The merged set from the two previously mentioned datasets contains **8882 samples** of short clips in between a range from 1 to 2 seconds from professional voice actors. Each sample is rated and given an emotion: anger, disgust, fear, happiness, neutral, sadness or surprise. Emotions levels or intensities are not taken into account for our experiments.

As shown in Figure 3.5, the dataset is balanced overall with same number of samples on each class with the exception of the *surprise* class.

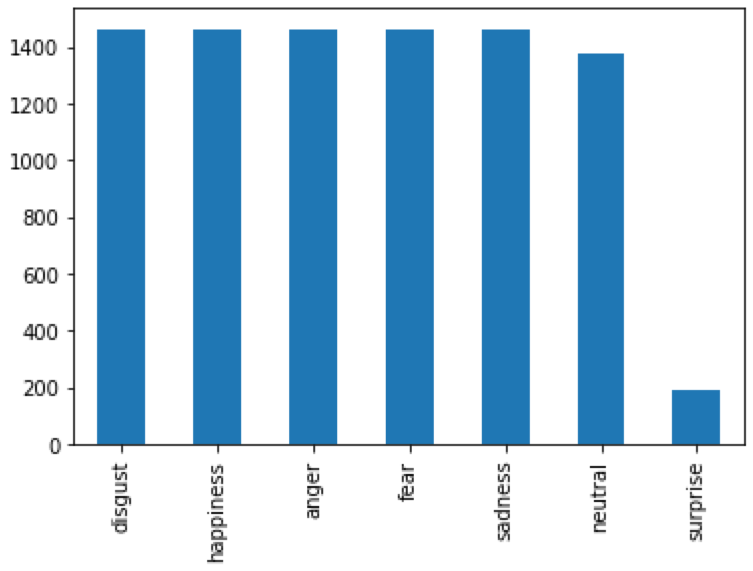


Figure 3.5: Labels bar-plot counts

RAVDESS dataset

The RAVDESS dataset is an audio-visual dataset, which means it has both video and audio files. It contains 7356 files with the voices of 24 professional actors (half male and half female) performing two sentences with American accent, each one with one of the eight emotions available: happiness, sadness, anger, fear, calmness, surprise as well as neutral.

This dataset includes both speech and song samples, each of them can be played in three modalities: Audio-Only, Audio and Video or Video-Only. From the 7356 files **only 1440** were used, for this project does not include songs and it does not treat video either. Only the speech files (16bit, 48kHz in .wav format) were chosen for the task of training. Each of the 24 actor performs 60 trials, leaving us with the total of 1440 samples.

The actors repeat two sentences ("Kids are talking by the door" and "Dogs are sitting by the door") each time with the seven emotions and two intensities (normal or strong), plus the neutral emotion. They also repeat this process twice. Emotion intensities are not taken into account for the classification tasks, but all audios are used.

The subsample of the dataset was downloaded from Kaggle[2].

CREMA-D dataset

Participants rated the emotion and emotion levels based on the combined audiovisual presentation, the video alone, and the audio alone. Due to the large number of ratings needed, this effort was crowd-sourced and a total of 2443 participants each rated 90 unique clips, 30 audio, 30 visual, and 30 audio-visual. 95% of the clips have more than 7 ratings.

The CREMA-D dataset is composed of **7442 clips from 91 different actors** (48 male, 43 female) with different ethnicities and accents such as hispanic, asian, african american, caucasian or unspecified (as opposed to RAVDESS which only had American accents). Actors performed 12 different sentences with six emotions each one: anger, fear, disgust, happiness, neutral and sadness, and four emotion levels from low, medium, or high to unspecified.

Same case as RAVDESS, each clip has its corresponding video file, and Audio-Only, Video-Only and Audio-Video formats are available, although we will be using just audio-only samples. **Every sample of the 7442 clips is used in this case for training**.

It is worth to point out that voluntary participants were chosen to rate the emotion and emotion levels of each audiovisual, video alone and audio alone samples. The bewildering fact resides in the subjectivity nature of the issue, for **95% of the clips had more than 7 different ratings**. This means in the 95% participants were not on the same page when categorizing each input. This fact shades light on the issue and **difficulty of labelling emotions from the voice** that it is not exclusive for machines, but for humans too.

The dataset files were also downloaded from Kaggle[3].

3.3.2 ML pipeline

A Machine Learning pipeline is the process or series of steps to create a Machine Learning (or Deep Learning in this case) model. It is similar to the concept of **workflow**. Although the defined steps usually vary in literature and are usually modified depending the case, the following Figure 3.6 represents the **fundamental ground steps**.

[2]RAVDESS Kaggle: https://www.kaggle.com/datasets/uwrfkaggler/ravdess-emotional-speech-audio
[3]CREMA-D Kaggle: https://www.kaggle.com/datasets/ejlok1/cremad

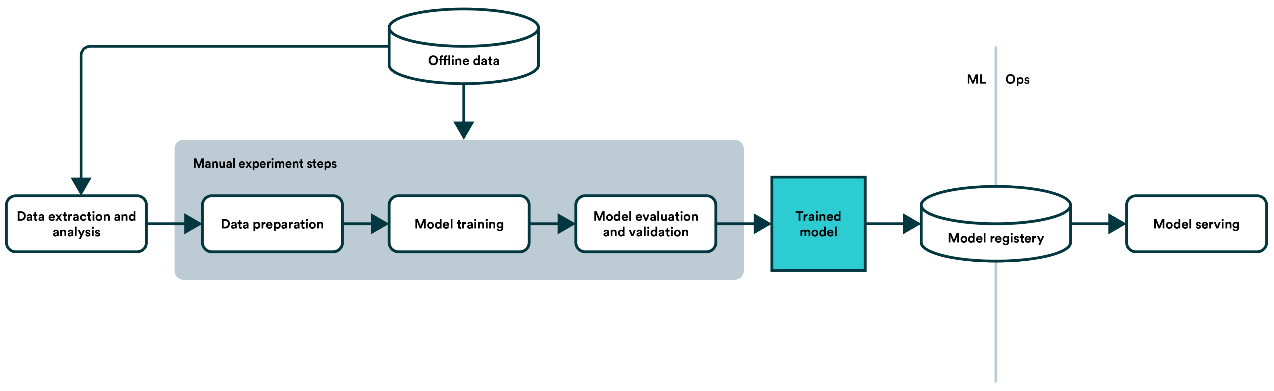


Figure 3.6: Steps of a Machine Learning pipeline[4]

Excepting 'Model Registry' and 'Model serving' steps, which are done in the Cloud architecture section 3.1, the rest of them are described in the subsequent sections. The developed ML pipeline developed in this project is **available in the Notebooks folder inside the project folder** and is written as a Python notebook and tested on Google Colab (see section 3.4.2 for more information).

3.3.3 Data extraction and analysis

The step on data extraction is very brief as this project does not build its own dataset or collect data as others do, for example, from data sources such as remote sensing systems, IoT devices, streaming data etc. What could be considered 'data extraction' was the process of looking up sources of speech data and collecting the datasets previously summarized.

The analysis was pretty simple as well. As shown in Figure 3.5, emotion labels count values were analysed to ensure the data was **balanced**. Sound properties or raw sound data from the sources were not analysed, as both datasets are benchmark, well-known in literature and reliable. The process of merging them was also an easy task as both have well balanced classes and the duration and quality of the recordings is pretty similar. They also work in English language which allows us to avoid problems that should be handled otherwise, such as **normalization and generalization**.

3.3.4 Data preparation

Preparing the data for its use and feeding of the model was the largest and most complex part. It could be broken down into the following pieces:

a). Loading and merging the datasets with their labels into a Pandas Dataframe.

b). Normalizing the classes of both datasets into one single dictionary.

c). Testing librosa functions for manipulating the data and applying it to samples.

[4]https://valohai.com/machine-learning-pipeline/

d). Creating different distortions with librosa for **data augmentation**.

e). Trimming and padding the audio samples to get a fixed length for each one, that will make training process more efficient.

f). Extracting and storing MFCC features for each sample to build the datasets.

g). Splitting data into a **training set** and a **validation dataset**.

h). Generating *augmented* data samples from a subsample of the dataset, extracting its MFCCs and appending the distorted data to the **training set**.

i). Transforming the categorical emotion labels to numerical values with **one-hot encoding** so that they can be processed by the CNN.

j). Applying **data normalization** to the MFCC features. This can be done with two methods: *Min-Max scaling* or *Z-Score normalization.*

k). Expanding the dimensions of the dataset to fit with the 2D CNN model requirements of shape.

Next we are not going to explore thoroughly and in detail each of these steps as they are uploaded and explained further in the notebook, but we are going to make some notes about some important topics mentioned in the list.

Mapping and merging classes

The process of mapping and merging classes consists in the task of building a common dataset with homogeneous and balanced labels.

To do this, two tasks were carried:

- Extracting **labels from filenames**. Each dataset has the label's information embedded inside the filename, and each one has a different naming convention (e.g. '1001_DFA_ANG_XX.wav' for CREMA-D and '03-01-01-01-01-01-01.wav' for RAVDESS). Following every naming convention, the label of each file was extracted and concatenated in a Pandas DataFrame.
- Building homogeneous and balanced classes. Every dataset has its own set of classes and naming conventions, for example RAVDESS contains a **calm** class which CREMA-D does not. This would cause unbalanced data and decrease in our model's performance. This problem is fixed by homogenizing labels into one single set and map or discard emotions that are not present in both datasets.

Data augmentation

Data augmentation refers to a set of techniques that allows us to generate more data than it is available (therefore "*augmenting it*"). This is done either by **making copies**

of data and slightly modifying or distorting them or by **creating new synthetic data from current existing data**.

This is very helpful when one does not have too much data to feed a model, as one of the main requirements of a *deep learning* model to work and train properly is to have **great amounts of data**.

But it is even more useful to **avoid the problem of overfitting**. Overfitting occurs when a model does not generalize well, that is to say, it "learns a lot" from training samples but when tested with other samples it shows poor performance. This is due to many factors, one of them is a training set's lack of diversity. When a training set is too homogeneous on its features, the model might **get used** to this kind of data, suppressing its ability to **generalize**. When new samples that do not follow the same patterns, it does not predict well. Introducing distorted or modified samples into the training set increases this *diversity*, allowing for **generalization**.

One example of data augmentation with images is shown in the below figure.

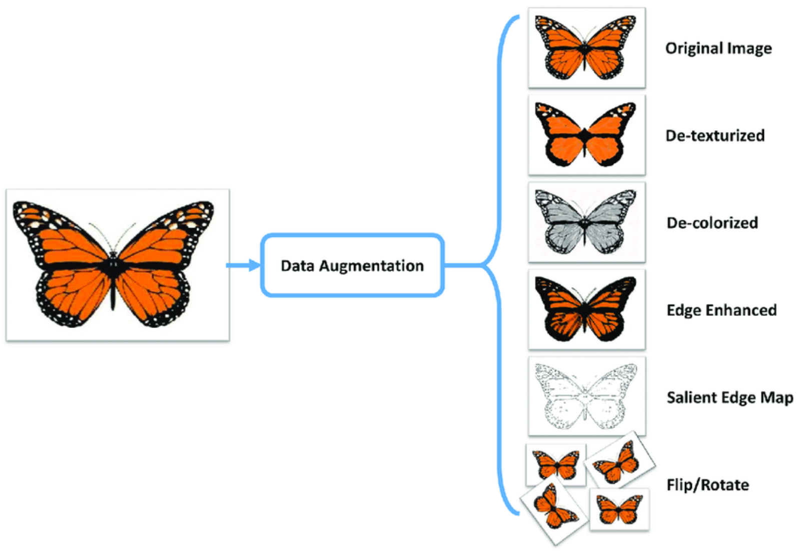


Figure 3.7: Data augmentation on images [1]

In our case, data augmentation is applied to **audio signals**. There where three techniques that were applied on 20% of the data, but there exist more of them:

- Adding **noise** to the signal.
- **Time-stretching** the signal by a fixed rate.
- Shifting the **pitch** of the signal.

Train, test and validation sets

All datasets must be split into separate sets in order to effectively build the model.

- A **training set** is used for the process of training the model. It must be the largest of the three in order to feed the model properly.

- A **validation set** is used to validate the model after the training and observe its performance. After that, you can draw conclusions on what could have failed in the model and you **tune the hyperparameters** of the model to train it anew and get better results, as in an iterative process. Once the model shows best performance validation set you stop iterating.
- Once the model is fully trained, the **test set** helps us evaluate its **real performance**. This means that the test set works as kind of a *real world scenario*, when you can clearly evaluate if your model performs well. It is not used to keep tuning the hyperparameters.

Two models of splits can be followed, as shown in the figure below.

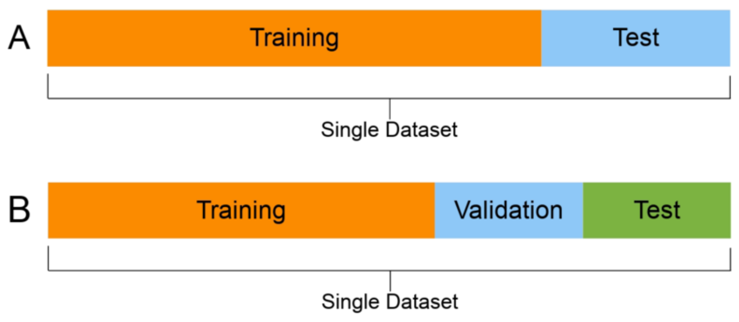


Figure 3.8: Ways to partition a dataset [6]

In this project, **only a training and test set are used**, following model A in Figure 3.8. This is due to the lack of size in the dataset that could cause decreased performance if split in three sets. The training set represents an 80% split of the original data plus augmented data samples, and the test set only contains 20% of the samples in the original data. This test set serves both as a validation and test set, helping us to tune our model and to make final tests.

One-hot encoding

Categorical variables are variables represented in text. These are a special kind of variables that must always be pre-processed in order to use them. The reason for this is that nor Machine Learning neither Deep Learning models are able to process or do computations over text, they only work on numerical representations. We can come to the conclusion that we must transform text representations into some kind of numerical representations.

One-hot encoding is the most straightforward manner to transform words into numerical representations. A single word is represented as a vector of size N, being N the size of the vocabulary (the number of unique words in the text we are trying to *translate*). Every position in the vector represents one word in the vocabulary; thence, a single vector will be composed of zeros in all its positions except for a one in the position of the word.

One graphical example can be seen below for better understanding.

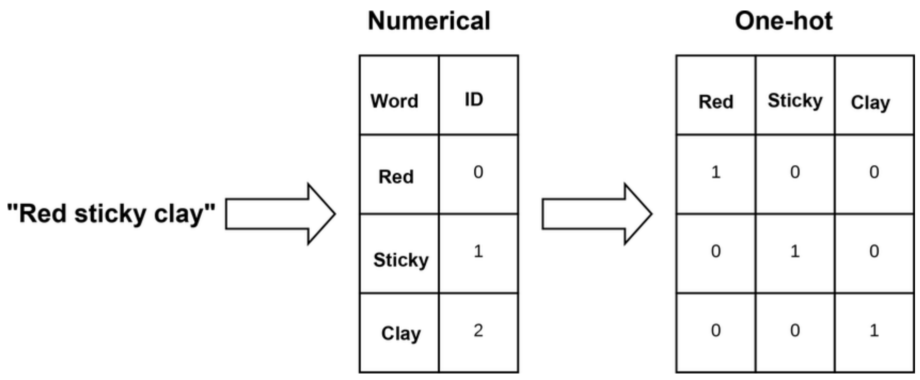


Figure 3.9: Label Encoding (first) and One-Hot Encoding (second) [19]

In the project, the categorical variables we must transform is the set of emotions which are the labels to predict in the model: *{anger, disgust, sadness, fear, neutral, happiness, surprise}*. To do that, we apply the two transformations shown in Figure 3.9. Firstly, we assign each unique label an integer ID (this is called Label Encoding or Numerical Encoding). Secondly, we apply one-hot encoding to each integer ID. In our case, we end up with vectors of 7 elements (corresponding to the 7 emotions).

Data normalization

Normalization is the process of bringing all data into a single **scale**. This step is critical and it can dramatically affect a ML or DL model performance. This is due to the fact that if two features have totally different scales, the biggest one in quantities could be "considered" more important by a certain model, altering the weights and parameters in favor of this bigger feature, thus decreasing performance, as it might not be true that this feature is "more important", leading to a *biased* demeanour.

For example, a model might have two features: distances measured in **km** and temperatures measured in Celsius **ºC**. Let's assume in our data distances are counted on thousands (e.g. the mean is around 1500km) and temperature has a mean around 50ºC. A *Deep Learning* model might be biased by these differences, as quantitatively speaking, distances are bigger, and thus they are more important. Scaling all these features into a single scale is the solution to give each feature *equal importance*, and let the model decide and select the weights for each one.

There are two famous types of normalization that are widely used because of their effectiveness in ML models: *Z-score normalization* and *Min-Max scaling.*

- **Z-score normalization**. It centers data around the mean (like centering data points around the axes). For a given set x, it projects each data point such that the new set has a mean of 0 and a standard deviation of 1. It follows the following formula.

$$x' = \frac{x - mean}{std} \tag{3.1}$$

- **Min-Max Scaling**. For a given set x, it projects each data point into a range between *[0, 1]* using the minimum and maximum value of the set. It follows the following formula.

$$x' = \frac{x - min(x)}{max(x) - min(x)} \tag{3.2}$$

For the project, the data that must be normalized are the **MFCCs** associated with each audio sample. Each method above was experimented, but **Z-score normalization** showed slightly better results. The global mean and standard deviation along each 2D MFCC was computed and used to normalize each sample.

3.3.5 Model architecture

The architecture of a *Deep Learning* model is composed of layers of neurons. As shown in section 2.1, our model will be a 2D-CNN based model. Therefore, our model will be a sequence of alternated Dense (common layers in Tensorflow, see 3.4.1 for more information) and convolution and pooling layers.

The model from which this project's architecture is based is AlexNet [11] (what could be regarded as the *baseline model*). However, there are some differences. Firstly, we are going to review this architecture.

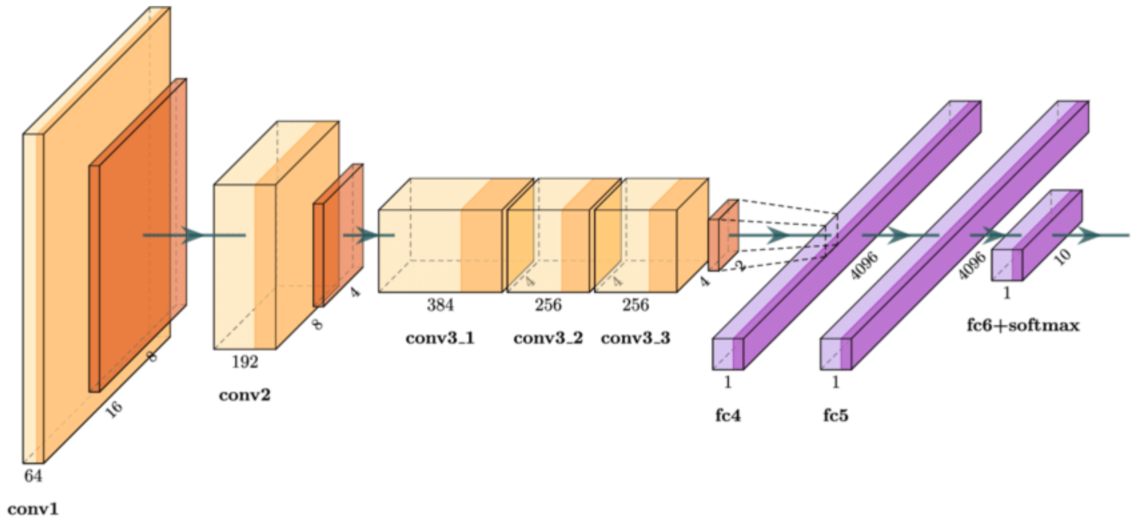


Figure 3.10: AlexNet architecture [23]

As a *classical* CNN architecture and studied in the state of the art, AlexNet shares the following features:

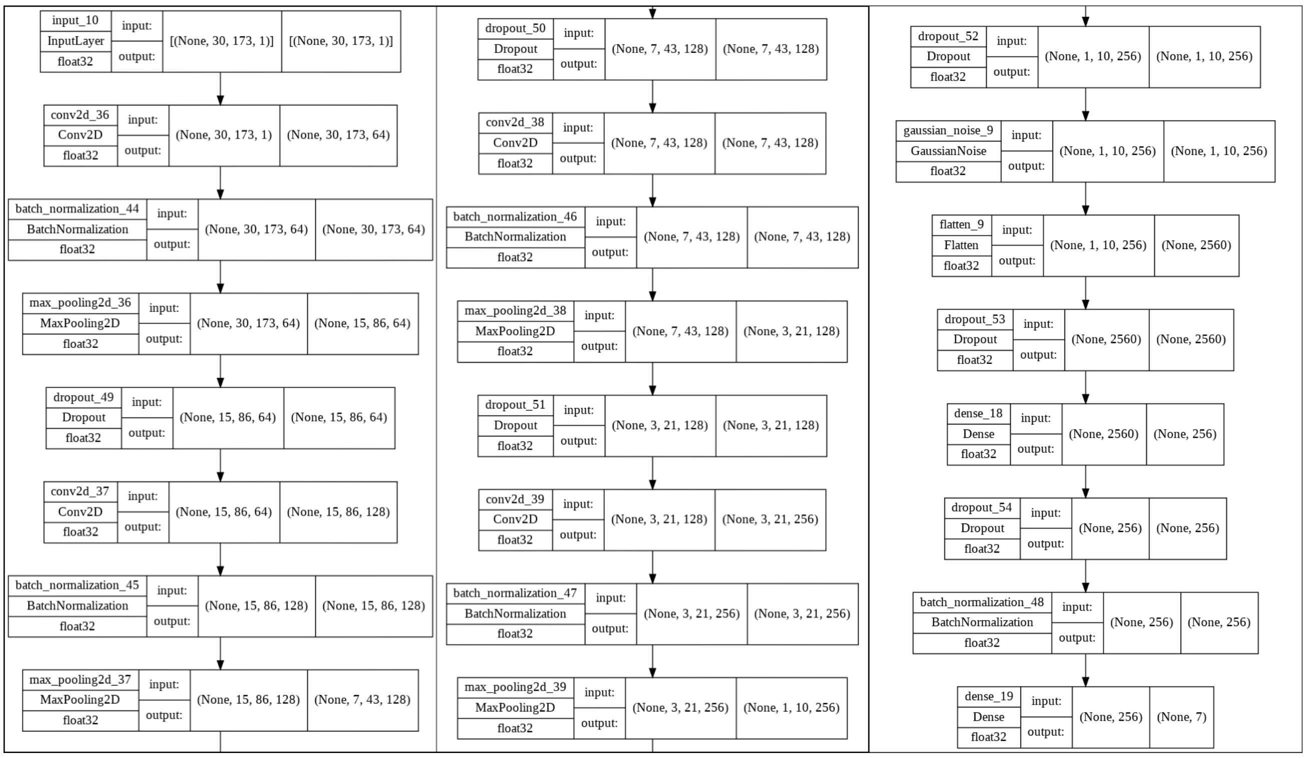
- **Nested Convolution and Pooling layers**. Three sequences of convolution followed by pooling layers are described. Also, in the final sequence, three convolutions are nested before pooling, this is a feature that **our model does not share**.

- **Fully connected layers**. After cutting down the size of the image through conv+pooling layers, the final product is flattened in a 1D vector and fed into

common Dense layers. The final layer (output) contains the same number of neurons as classes and makes classification with *softmax*.

- **Decreasing kernel sizes**. In each convolution and pooling step, kernel size decrease to avoid overfitting. In the original AlexNet, it starts with square kernels of 11x11, then are reduced to 5x5 and finally to 3x3. As our model is simpler, it starts on 5x5 kernels and then 3x3 kernels.

Our model has the following architecture, as graphically represented by Keras.



input_10 InputLayer float32 input: [(None, 30, 173, 1)] output: [(None, 30, 173, 1)]
conv2d_36 Conv2D float32 input: (None, 30, 173, 1) output: (None, 30, 173, 64)
batch_normalization_44 BatchNormalization float32 input: (None, 30, 173, 64) output: (None, 30, 173, 64)
max_pooling2d_36 MaxPooling2D float32 input: (None, 30, 173, 64) output: (None, 15, 86, 64)
dropout_49 Dropout float32 input: (None, 15, 86, 64) output: (None, 15, 86, 64)
conv2d_37 Conv2D float32 input: (None, 15, 86, 64) output: (None, 15, 86, 128)
batch_normalization_45 BatchNormalization float32 input: (None, 15, 86, 128) output: (None, 15, 86, 128)
max_pooling2d_37 MaxPooling2D float32 input: (None, 15, 86, 128) output: (None, 7, 43, 128)
dropout_50 Dropout float32 input: (None, 7, 43, 128) output: (None, 7, 43, 128)
conv2d_38 Conv2D float32 input: (None, 7, 43, 128) output: (None, 7, 43, 128)
batch_normalization_46 BatchNormalization float32 input: (None, 7, 43, 128) output: (None, 7, 43, 128)
max_pooling2d_38 MaxPooling2D float32 input: (None, 7, 43, 128) output: (None, 3, 21, 128)
dropout_51 Dropout float32 input: (None, 3, 21, 128) output: (None, 3, 21, 128)
conv2d_39 Conv2D float32 input: (None, 3, 21, 128) output: (None, 3, 21, 256)
batch_normalization_47 BatchNormalization float32 input: (None, 3, 21, 256) output: (None, 3, 21, 256)
max_pooling2d_39 MaxPooling2D float32 input: (None, 3, 21, 256) output: (None, 1, 10, 256)
dropout_52 Dropout float32 input: (None, 1, 10, 256) output: (None, 1, 10, 256)
gaussian_noise_9 GaussianNoise float32 input: (None, 1, 10, 256) output: (None, 1, 10, 256)
flatten_9 Flatten float32 input: (None, 1, 10, 256) output: (None, 2560)
dropout_53 Dropout float32 input: (None, 2560) output: (None, 2560)
dense_18 Dense float32 input: (None, 2560) output: (None, 256)
dropout_54 Dropout float32 input: (None, 256) output: (None, 256)
batch_normalization_48 BatchNormalization float32 input: (None, 256) output: (None, 256)
dense_19 Dense float32 input: (None, 256) output: (None, 7)

Figure 3.11: Project's model architecture divided in 3 columns

The project's model differs from AlexNet in some points:

- **Different input and output shapes**. The input shapes are adapted to the ones that form MFCCs, which are not square. Output shape is the number of classes of the dataset, or the same as number of emotions.

- **No nested convolutions**. This model was firstly designed with nested convolution layers like in the last phase of AlexNet, but that experiment showed worse results.

- **Kernel sizes**. Kernel sizes only go from 5x5 to 3x3.

- **Overfitting avoidance**. Our model introduces some layers that control overfitting such as Dropout, Batch Normalization and Gaussian Noise. Each of these measures showed increase in performance.

3.3.6 Model evaluation

The model evaluation is about measuring different metrics to find out if our model is working well. For a *Deep Learning* model, the critical task is to **minimize the model's loss** in order to improve the metrics.

Loss. The loss of a model measures how overall distant are the predictions made by the model from the real labels (in other words, the error). This is calculated by a **loss function**. There are several loss functions for regression and classification problems such as ours: the loss function used for our model is **categorical crossentropy**, which is canon for multi-class classification problems.

Metrics. Several metrics are used to assess different characteristics of a model's performance. These metrics are highly correlated with the loss, as they have an proportionally inverse: as loss drops, metrics tend to increase. There are some common metrics widely used in ML such as **accuracy, precision, recall and F1 score**. For this project, we will only be using **accuracy**, as for our model we do not distinguish between Type I or Type II errors. Accuracy can be defined as the proportion between the correct predictions with respect to all the predictions made by the model, following the next formula.

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{3.3}$$

Being TP true positives, TN true negatives, FP false positives and FN false negatives.

A final ingredient that a model must have in order to learn and evaluate is an **optimizer**. The optimizer is the key piece to minimize the loss function. On each epoch of the training process, the optimizer helps tuning the parameters such as the **weights and biases** of the model based on a **learning rate**, which is another hyperparameter.

The most famous optimizer is **Stochastic Gradient Descent**, but for this project it was more convenient to use an optimizer called **AdaGrad**, which let us tune the learning rate hyperparamter for better performance.

To see the results of the model, see the Results chapter 5

3.4 Used technologies

3.4.1 Tensorflow 2.0

TensorFlow is an open source platform built by Google for machine learning and deep learning. More than a library, Tensorflow is a backend on itself, containing tools, libraries and resources (docs) that allow developers and researchers to easily build and deploy Machine Learning applications[5].

Tensorflow 2.0 provides you with:

1. **Easy model building**. TensorFlow allows to intuitively build and train ML models using its high-level APIs such as Keras.
2. **Robust ML production**. It is available in multiple languages and ready to deploy in different environments such as Cloud, on-premises, browsers or mobile devices.
3. **Powerful experimentation for research**. It provides a simple architecture to easily implement ideas and state-of-the-art models into code.

Although Tensorflow is built with C++, thanks to its high-level APIs as Keras, it is highly supported in Python, which is the main language that is used to make our models in this project.

3.4.1.1 Keras

Keras is a high-level API for Machine Learning development written in Python. It is an open-source neural-network library that allows fast experiments with *deep learning*, and as a high-level API it can be run on multiple backends such as CNTK, Theano or TensorFlow.

Keras is known for its modularity and user-friendliness. It does not handle low-level operations (such as TensorFlow would do), which are delegated to the previously mentioned backend [24].

In this context, TensorFlow is a framework that offers both high and low-level APIs, so both Keras and Tensorflow could be decoupled and work separately. One of the key differences is that Keras needs a backend "engine" to work. As stated before, there could be several possible backends as Theano or CNTK, but since Keras 1.1.0 release, Tensorflow works as the default backend for Keras.

However, with the Tensorflow 2.0 release, Keras became fully integrated with Tensorflow with the "tf.keras" module inside the package. Moreover, since Keras 2.3.0 release, Google encourages developers to switch from Keras to Tensorflow 2.0 keras submodule, and discourage them to keep on using older APIs from TF 1.0 that did not include Keras.

In the figure below we can comprehend the relationship between the two more graphically:

[5]https://www.tensorflow.org/

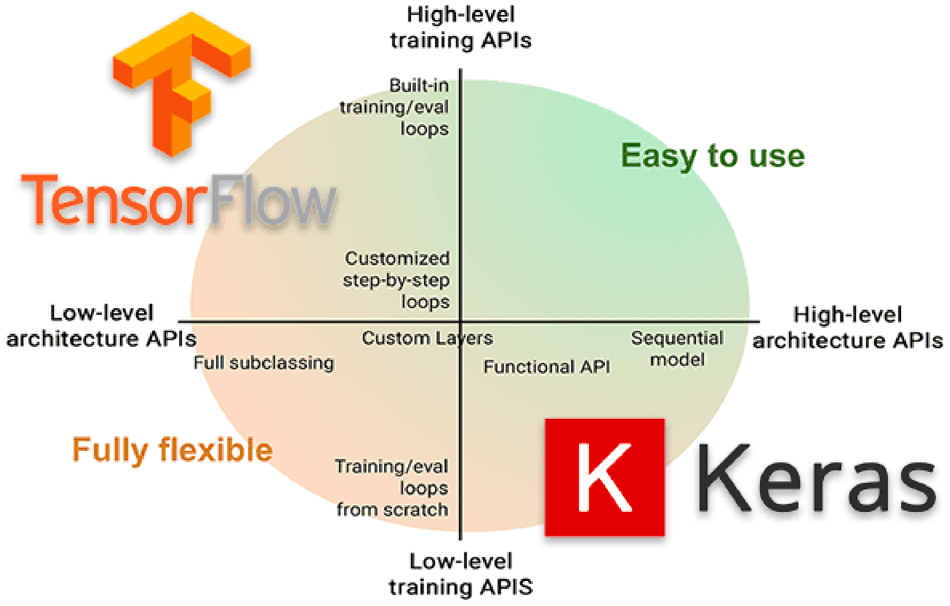


Figure 3.12: Keras and Tensorflow libraries difference [20]

3.4.1.2 Functional API

The (TensorFlow) Keras functional API is a flexible way to create models, that is more flexible than the usual Sequential API. The functional API allows models with non-sequential topologies, such as shared layers or multiple input or output layers.

A *deep learning* model is usually seen as a directed acyclic graph (DAG) of layers. This functional API allows to build this kind of graphs[6].

This functional API provides an simple and simple way to write more complex models that would otherwise be impossible to write with one Sequential model (we would require to concatenate several Sequential models).

An example of the differences between the two models can be seen below:

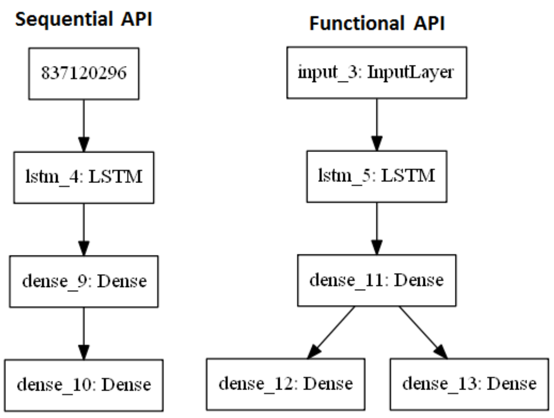


Figure 3.13: Sequential vs Functional API difference [12]

[6]https://www.tensorflow.org/guide/keras/functional

3.4.2 Google Colaboratory

Google Colab or Colaboratory is an online SaaS tool for developers to write notebooks in Python[7]. It has great advantages in comparison to other local notebook environments like Jupyter Notebook such as:

- It doesn't require to build a local *kernel* or to **install many of the libraries** that are typically used in ML or data science because they are already integrated, for instance: Pandas, TensorFlow, Sci-Kit Learn, Matplotlib or Librosa (sound library).

- It is **free** and all resources are allocated on Cloud, which means that all the code is run in Google servers, so **is not consuming CPU or RAM from your computer**. Free-tier notebooks are always available although they have limited memory and CPU.

- You can use execution environments with **GPUs and TPUs accelerators for free**. This is extremely useful when training a *deep learning* model, which makes the process orders of magnitude faster in time. This is because GPUs and TPUs have excellent abilities for parallelism, a quality that is intensively utilised in matrices operations, that DL use very frequently.

- It fits together very well with other Google services such as Google Drive, which can be useful for storing and retrieving datasets or saving results without the need to store big amounts of data into your computer.

Colab also has some limitations in its free-tier. For instance, all resources are shared, which means if there is a lot of demand at one point, availability of service is not always guaranteed. This mainly affects GPUs and TPUs, which are continously demanded. Limited RAM memory and CPU usage is another downside. However, this limitations can be skipped with a suscription to Google Colab Pro paying a monthly fee. For this project, only the free-tier service was enough and GPU execution was used.

3.4.3 Amazon Web Services

Amazon Web Services (AWS) is one of the most widely known Cloud providers that offers over 200 services globally[8]. AWS provides on-demand cloud computing platforms and APIs to users and companies. These services provide an abstraction of technical infrastructure and tools and distributed computing [22].

The AWS technology is implemented at server farms distributed around the world. Fees are based on a "Pay-as-you-go" model, following a combination of usage time, hardware, OS, software and other features chosen by the subscriber options. Users can pay for single virtual computers, physical computers or clusters [22].

Now we are going to explore each AWS service that was relevant to the development of the project and give a brief overview of the features of each one.

[7]https://colab.research.google.com/
[8]https://aws.amazon.com/what-is-aws

3.4.3.1 Lambda Functions

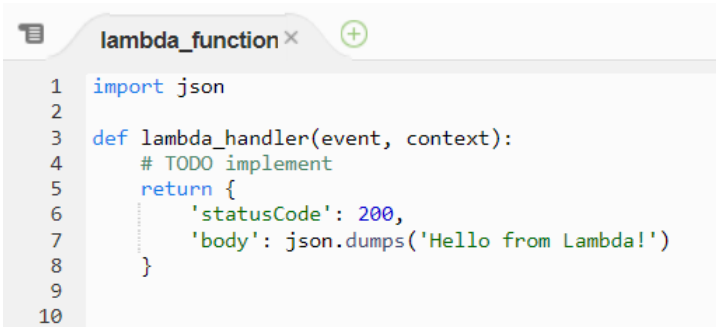
Amazon Lambda is a compute service that allows users to run code without managing servers or infrastructure. Lambda runs code on a high-availability computing infrastructure and it performs all the provisioning and administration of resources tasks, including maintenance and *autoscaling*, monitoring and logging. Any type of app or service can be implemented with Lambda if the code is written in one of the Lambda supported languages[9]. One of the languages supported by Lambda and it is used throughout this project is **Python 3**.

All the developer needs to do is to organize the code into Lambda functions. Lambda runs this function only when requested and autoscales if the demand requires it, from a few requests per day to thousands per second.

Functions

A function is the main resource that is invoked to run your code with Lambda. The code of the function has to process the events that are sent as arguments by an application or another AWS service[10]. That is, you can have any number of functions into your code, but you have to wrap them up into a main function that will be invoked in every call to Lambda (the *handler* function).

- Usually, a Lambda function (in any of the available languages) takes at least one argument: `event`. This event comes as a JSON with information regarding the call to Lambda. Then, usually this function must return another JSON in a HTTP response fashion (with an status code and a body).
- The name of the function must be set up. By default in Python 3 it is called `lambda_handler`, but it can be changed by the user.



```python
import json

def lambda_handler(event, context):
    # TODO implement
    return {
        'statusCode': 200,
        'body': json.dumps('Hello from Lambda!')
    }
```

Figure 3.14: Example of a Lambda function in Python 3

Events

An event is a JSON document that contains information that the Lambda function uses and processes. The execution environment converts this event to an object that is passed to the main function. When the function is invoked, the structure and content of the event are determined[11].

[9]https://docs.aws.amazon.com/lambda/latest/dg/welcome.html
[10]https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-concepts.html
[11]See footnote 9

- When invoked by another AWS service the event template is determined by the service. Nonetheless, you can configure your own custom event template as shown below:

```
        {
          "Widthpx": 281,
          "Heightpx": 250,
          "Lengthpx": 52,
        }
```

Listing 3.1: Custom JSON event template - weather data

Code deployment

There are two ways of exporting our code to Lambda in order to use it afterwards. This is done by using a *deployment package*. Lambda supports two types of deployment packages[12]:

1. A .zip file archive that contains the function code and its dependencies (libraries that are necessary to run the code). Lambda provides the OS and runtime for the function.
2. A container image compatible with the Open Container Initiative (OCI) specification. The code and dependencies mentioned must be added to the image. In this case, the OS and Lambda runtime are included by the user.

As AWS have *quotas* for the direct upload of a .zip file, allowing just 250 MB (unzipped)[13] for the *deployment packages*, and our package plus dependencies such as Tensorflow overtake this boundary by far, **the project was deployed by the container method**. Further detail about container deployment with ECR and Docker in section 3.4.3.2.

Cold start

When a function is invoked for the first time after a period in time (usually 5 minutes in Lambda), it has to perform a few steps before being able to run the function. These steps cause little overhead in time and that is what is called *cold start*.

First, the function needs to **download the function code**, either directly from Lambda or from a container stored in Elastic Container Registry. Either case, after that it needs to **start an execution environment**, which is similar to opening a shell or running `docker run` [4]. Lambda has its own execution environment which is similar to a "light Linux shell" available to the public. In our case, this is a Python shell, using the official public Docker base image `public.ecr.aws/lambda/python:3.8` published by Amazon.

If the backend or dependencies of the container are heavy, the cold start time increases.

[12]See footnote 9
[13]https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html

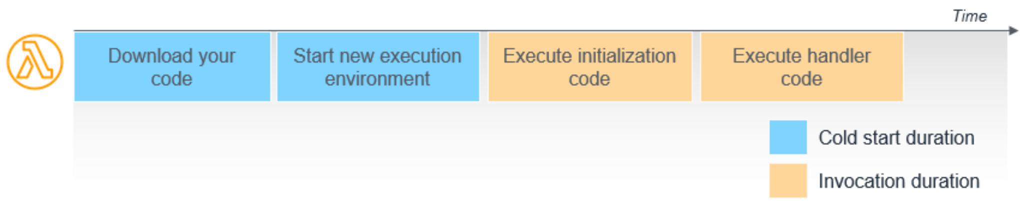


Figure 3.15: Cold start and invocation diagram [4]

Triggers

A trigger is a Lambda resource that is configured to invoke the function under certain circumstances, like in response to events, external requests or scheduled events. A function can have multiple triggers. Each trigger simulates the behaviour of a client invoking the function, and the events passed to the function has data from only one source[14].

Some of the different services that can trigger a Lambda function are: **Amazon SNS, Amazon EventBridge, Amazon CloudWatch Logs and Amazon S3 Buckets**. Different service *callers* might have different invocation types as:

- **Synchronous invocation.** The caller *waits* for the Lambda function response after sending the request. Some examples are API Gateway or calls through the AWS CLI.
- **Asynchronous invocation.** The caller doesn't wait for a response from Lambda, instead, Lambda places the event in a queue of events and returns an immediate response without additional information. Another process reads the events from the queue and sends them to your function[15]. Some examples are S3 buckets, Amazon SNS or Amazon EventBridge.

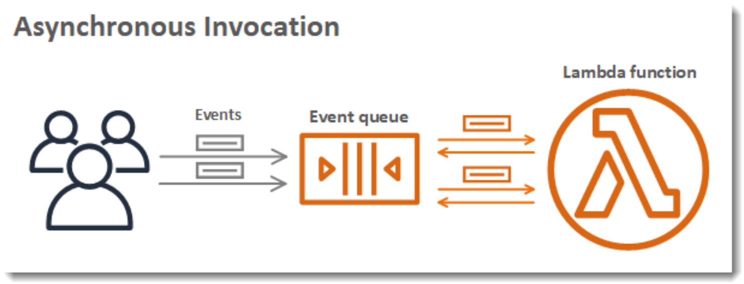


Figure 3.16: Lambda Asynchronous invocation[16]

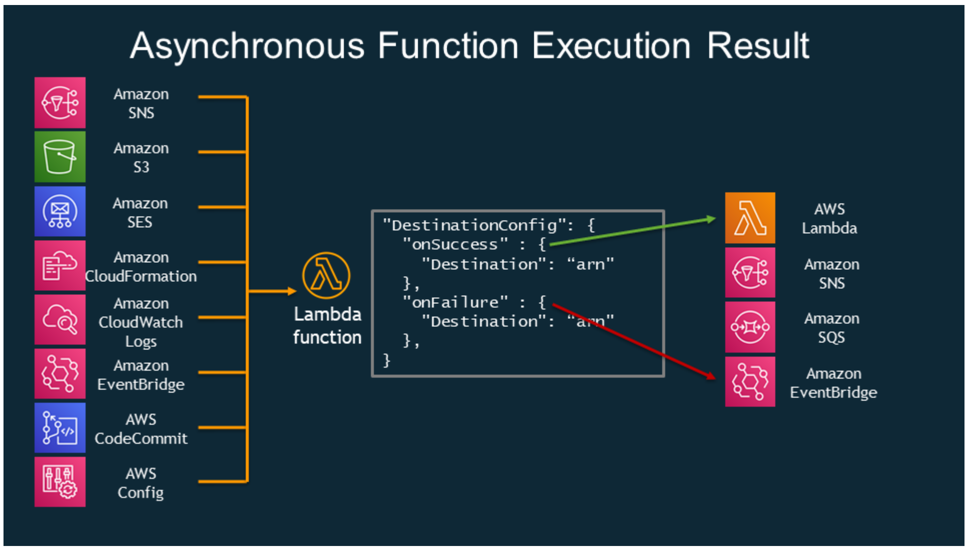
Asynchronous invocations are particularly useful along with the use of **Destinations**, which will be the next topic covered.

[14]https://docs.aws.amazon.com/lambda/latest/dg/lambda-invocation.html
[15]https://docs.aws.amazon.com/lambda/latest/dg/invocation-async.html
[16]See footnote 14

Destinations

In every of the asynchronous invokations of a function, Lambda sends each event to a queue of events. Another process reads these events from the queue and runs the function. When an event is inserted into the queue, Lambda previously returned a successful 200 HTTP code to confirm the arrival of the event to the queue. There is no additional information or response from the function to know if the function run was successful.

Lambda Destinations allows to route the results of the execution to a specific resource without writing more code. There are four different destinations to be chosen on either a failure or success case: Amazon SNS, SQS, EventBridge or another Lambda. It can also be configured to route different results to different destinations [26].



```
"DestinationConfig": {
  "onSuccess" : {
    "Destination": "arn"
  },
  "onFailure" : {
    "Destination": "arn"
  },
}
```

Figure 3.17: Lambda Triggers (left) and Destinations (right) [26]

As shown in the figure above, we can invoke our functions through the services appearing on the left and store our results on other services. As we will see later in section 3.1, in the project we use S3 events as triggers and queues from SQS to store the results.

Pricing

As Amazon explains: *With AWS Lambda, you are charged based on the number of requests for your functions and the duration it takes for your code to execute.*[17] This means you are not charged everytime you create a function or allocate your dependencies in Lambda.

How does Lambda count requests and execution time?

- **Requests**. Lambda counts each execution in response to events or triggers such as SNS or EventBridge, or invocation calls from API Gateway as requests (including test invocations from the AWS console).

[17]https://aws.amazon.com/lambda/pricing/

- **Execution time**. Duration of execution is computed as the time your code starts to run until it returns or terminates (by some exception or failure), rounding up to the nearest 1 ms. The price depends on the maximum amount of memory that is assigned to the function (chosen by the user), from which CPU power and other parameters are calculated

AWS also offers a **free tier** service for Lambda. As Amazon states: *The AWS Lambda free tier includes one million free requests per month and 400,000 GB-seconds of compute time per month, usable for functions powered by both x86, and Graviton2 processors, in aggregate.*[18]

The scope of this project is subject to this free tier boundaries as it is more convenient for reducing costs.

3.4.3.2 Amazon Elastic Container Registry (ECR)

Amazon Elastic Container Registry (Amazon ECR) is a container image registry service provided by AWS. It supports private repositories with permissions using AWS IAM users, roles and policies. This allows specific users from EC2 instances to access the container repositories and images. You can push, pull and manage Docker images or OCI images[19].

As stated in part 3.4.3.1, AWS ECR is a solution to the direct upload of our dependencies and code to our Lambda Function. With the help of **Docker**, all you have to do to upload code dependencies with Tensorflow to Lambda is the following:

1. Create an AWS ECR repository to store our image.
2. Create a *Dockerfile* from a Lambda with Python base image and write the commands to store the dependencies and the main Lambda function code.
3. Build the Docker image based on the *Dockerfile*.
4. Upload the Docker image to the ECR repository.
5. When creating the Lambda function, select "Create from image" instead of creating from scratch and select the image in the ECR repository.

3.4.3.3 API Gateway

Amazon API Gateway is the AWS service for creating and managing secure REST, HTTP and WebSocket APIs that are scalable. Developers can create this APIs to access other AWS services or data stored in the Cloud. Your APIs can be used in your own client apps[20].

In the scope of this project, API Gateway allows the client application (the *end-user* part) to communicate with the Amazon services without the need of signing up into AWS or having to share any account or credentials, as we will see later.

[18]See footnote 16
[19]https://docs.aws.amazon.com/AmazonECR/latest/userguide/what-is-ecr.html
[20]https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html

3.4.3.4 Amazon S3 Buckets

Amazon Simple Storage Service (Amazon S3) is the object storage service offered by Amazon. It comes with features such as security, performance, availability and scalability. Developers and clients can use this service to store and protect any data for many use cases like mobile apps, websites, backups, IoT, analytics, data lakes and more[21].

3.4.3.5 SQS Queues

Amazon Simple Queue Service (SQS) is the queueing service of Amazon. It enables to decouple and make scalable microservices, serverless apps and distributed systems. With SQS, you can send and receive messages between different components (and Amazon services) in small or large volumes, without losing information or requiring other services[22].

There are two types of queues offered by SQS[23]:

1. **Standard Queues**: This kind of queues are characterized by two main features:
   - **At-Least-Once Delivery**: As Amazon explains: *A message is delivered at least once, but occasionally more than one copy of a message is delivered.*
   - **Best-Effort Ordering**: In some occasions, messages can be delivered in different order from which they were sent.

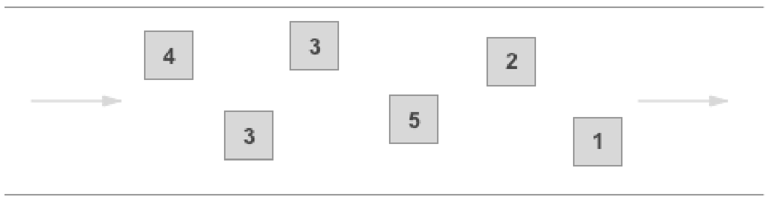


Figure 3.18: SQS Standard Queue diagram[23]

2. **FIFO Queues**: unlike Standard Queues, FIFO is characterized by FIFO delivery and Exactly-Once Processing:
   - **Exactly-Once Processing**: A message is delivered once and it is available until a consumer processes and deletes it. There are not duplicates in the queue.
   - **First-In-First-Out Delivery**: The order in which messages were received is preserved when they pop out of the queue (First-In-First-Out).

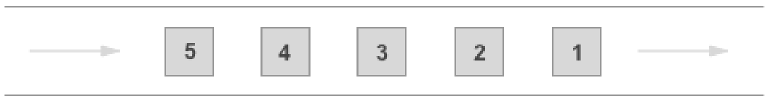


Figure 3.19: SQS FIFO Queue diagram[23]

[21]https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html
[22]https://aws.amazon.com/sqs/
[23]https://aws.amazon.com/sqs/features/

3.4.3.6 EventBridge Rules

Amazon EventBridge is an event bus made for serverless that facilitates the task of building scalable event-driven apps using events generated from your applications, SaaS apps or other AWS services. Routing rules can be set up to determine where to send the data to build architectures that react in real-time with the sources with producer and consumer decoupled[24].

In the context of this project, EventBridge will be useful to keep our Lambda functions *awake*, as cold-start in the main function comes with a great overhead in time for a user that opens the application.

3.4.4 PyQT5

Qt is a set of libraries written in C++ that implement high-level APIs for developing diverse aspects of modern applications and systems, such as location and positioning, multimedia, NFC and Bluetooth connectivity, but most importantly, User Interfaces development.

PyQt5 is a set of Python bindings for Qt v5. It enables GUI development with Python as an alternative to C++ on all supported platforms included mobile ones such as iOS and Android and it is implemented with more than 35 extension modules[25].

In this project, PyQT5 helps to develop the GUI for the end-user to use and interact with.

3.4.5 librosa

From librosa docs: *librosa is a python package for music and audio analysis. It provides the building blocks necessary to create music information retrieval systems*[26].

In the context of this project, librosa is used in the pre-processing phase of the AI model. It allows to manipulate audio data (inserting noise and distortion is very useful for *data augmentation*), extract features as the previously explained MFCC or spectrograms, trim and pad audio and convert it to array formats to feed it into the network.

3.4.6 PyAudio

From PyAudio docs: *PyAudio provides Python bindings for PortAudio, the cross-platform audio I/O library. With PyAudio, you can easily use Python to play and record audio on a variety of platforms, such as GNU/Linux, Microsoft Windows, and Apple Mac OS X / macOS*[27].

PyAudio is useful in the end-user application to record the voice in a simple manner, without the need of managing audio drivers or writing complex I/O operations in code.

[24]https://aws.amazon.com/eventbridge/
[25]https://pypi.org/project/PyQt5/
[26]https://librosa.org/doc/latest/index.html
[27]https://people.csail.mit.edu/hubert/pyaudio/

3.5 Discarded technologies

3.5.1 PyTorch

PyTorch is a deep learning framework and library for Python based on Torch. It was developed by Facebook's AI research group. PyTorch is known for its simplicity, ease of use and computational efficiency, both on memory and CPU usage. It makes coding ML and DL more manageable and increases processing speed [24].

Both PyTorch and TensorFlow have upsides and downsides. For instance, PyTorch outperforms TensorFlow in optimizing performance with paralellism, therefore making it faster.

Nonetheless, the choice of using TensorFlow instead of PyTorch is based on the fact that TensorFlow is better provided with documentation, practical examples, tutorials, and even lots of pre-trained models with TensorFlow Hub and datasets integrated in the library. For a starter in the field of *deep learning*, TensorFlow seems to be more beginner-friendly. If this project sought optimality for the model PyTorch would have been a prime option to consider.

3.5.2 Other languages

TensorFlow is written in C++ and CUDA, and it has APIs available in Python, C++ and Java, which means any model could be built in any of these languages. However, not all the features are available in the Java API, which is a compelling reason to rule out this option.

The main reason to choose Python is because, besides having all the necessary features to build ML and AI (which is the reason why it is the most widely used language for that purposes), its simplicity and flexibility make the learning curve much smoother and user-friendly than C++ or Java. It also has integrated libraries for managing complex math operations and array matrices operations such as NumPy, SciPy and Pandas, which are quite useful for pre-processing and post-processing of data.

C++ solutions on *deep learning* are more research-oriented and seeking speed and optimality in exchange for complexity in code.

3.5.3 Other Cloud providers

As shown in Table 2.1, any top Cloud provider nowadays offers FaaS services in their catalogue.

After careful consideration, Amazon was chosen mainly for a matter of convenience, as I had previously worked with AWS machines and services such as EC2 and S3, and thus I acquired ease of use with the AWS console. Other factors were also taken into account, such as differences in costs which are little significant between the competitors; supported languages, all of them fully supporting Python in their services and performability, all of them showing similar characteristics.

Chapter 4

Use cases

In this brief chapter we will see an overview of two possible use cases that the system should be able to handle.

4.1 UC1: An individual makes a 10 second recording

The step-by-step process from the moment the user starts the application until it receives predictions is described here.

1. First, the user sets up a microphone as input device and starts the GUI desktop app. When the window of the application is open, the user presses the 'Start' button to begin the process.

2. When the 'Start' button is pressed, the recording starts and anything that the input device gathers is stored and used as input. At any moment, the 'Stop' button can be pressed so the recording and storing is terminated from that moment.

3. As shown in 3.3, in the back, the client application is generating a temporary file of the last 2 seconds of recording on one thread. When a temporary file is created, another thread is responsible for making the call to the API Gateway endpoint with a POST request containing this file. In this use case, our user generates 5 temporary files and 5 invocations to the API, as he records himself for 10 seconds.

4. From the moment API Gateway receives the first request, the architecture is started and the Lambda functions get warmed up, as shown in 3.1. While the functions get warmed up with the first piece of audio, the user might notice some delay in the first response due to *cold-start*.

5. The main Lambda function generates a prediction based on the pieces of audio that it receives, and sends them to the SQS queue. Under normal circumstances, the queue should store the 5 predictions from the 5 pieces in order from oldest to

newest, but due to the nature of a Standard Queue, it could occasionally return some prediction in wrong order or even duplicate.

6. The whole time another thread in the client application was making requests to the other API Gateway endpoint asking for predictions to the queue. When the queue is empty, no result is retrieved, but when it catches a prediction, it is stored in a temporary CSV file along with the timestamp that corresponds to the chronological order of the recording. This event causes an automatic re-render of the chronogram plot of the GUI, allowing the user to see the emotions in nearly real-time.

7. The user stops the recording after 10 seconds, leaving the GUI in a state with a 10-second window with the emotions represented in each timestamp. The user closes the application.

4.2 UC2: An individual generates a single recording

This use case is similar to the previous one, with the difference that in this case a user does a 2 second or less recording. This causes several consequences related to the architecture:

- As the user only generates one sample, if the architecture was previously in an idle state of *cold start*, the user will experiment the delay until the functions are warmed-up and return the prediction. This means such user would not experiment the nearly real-time experience, as others do when more samples get *pipelined* by the architecture and the difference in response time is noticeable.

- The prediction that the user will get will be 100% reliable, as there will only be one sample queued in the SQS queue. As discussed earlier, this is not such the case always when there is more than one prediction queued in a Standard Queue, in that case the reliability of the prediction/timestamp correspondence drops.

Chapter 5

Results

In this chapter we are going to be discussing some results obtained from the proposed system such as our *Deep Learning* model performance.

5.1 Speech Emotion Recognition model results

We are going to explore the results of **losses and metrics** explained in section 3.3.6.

In graphs below we observe two curves with different colors: blue and orange. The horizontal axis represents the number of epoch: an epoch is like a *step* during the process of training; the vertical axis represents either the total value of loss at a certain epoch, as computed by our chosen loss function in the first one or the accuracy of the model in the second one. We can observe that the process of training takes **100 epochs**. This is another hyperparameter that was chosen before training.

5.1.1 Loss

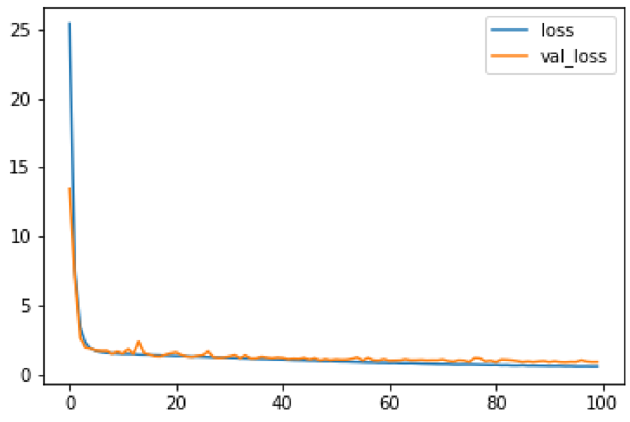


Figure 5.1: Model training and validation loss

Inside the loss graph, whose score is determined by the loss function (**categorical cross-entropy** in our case), we can observe two curves of different colors: blue and orange.

- The blue curve shows the loss **during the process of training**. It is computed as the mean loss between the batches of the corresponding epoch on training samples.
- The orange curve shows the loss **of the validation set**. After each epoch, every sample in the validation set (which is the same as our test set in this case) provided in `model.fit()` is evaluated as if there was a call to `model.evaluate()` after each call.

Watching the relation between the two curves, we can clearly observe a phenomenon called **convergence**. When a loss function rapidly decreases and remains near a certain value as an asymptotic value (ideally zero) we can say it **converges**. When both of the curves follow a similar "path", we can say it has a **good fit**. Otherwise, in some cases the validation loss decreases at the beggining but then it starts to increase over epochs, we would be observing **overfitting**, but this is not such case.

5.1.2 Accuracy

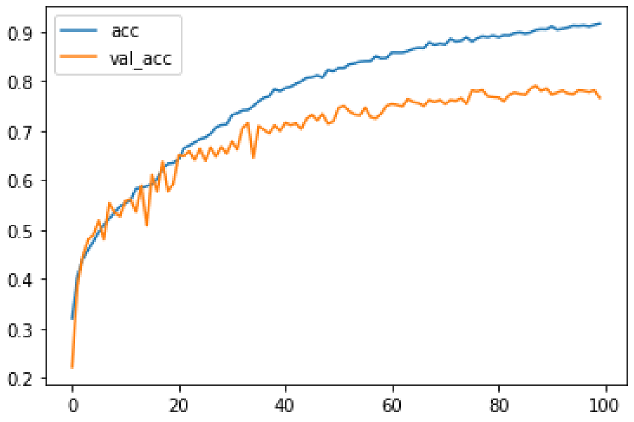


Figure 5.2: Model training and validation accuracy

As we stated before, loss and metrics usually have an inverse relationship: while we try to minimize loss, we usually maximize metrics; so when loss decreases due to **backpropagation** and weights adjustment, metrics tend to get better.

This is such the case with accuracy (whose definition you can encounter on equation 3.3): both training and validation sets increase over epochs as loss function converges. However, we also observe that the two curves go along at the beginning but then they separate: the validation accuracy gets *stuck*, whilst training accuracy keeps growing. This means there is some **overfitting** in our model. Overfitting is produced when the model learns features from the training set that are not relevant out of the training scope. Thus, training predictions still keep getting better, but real-world samples start to fail: the model lacks **generalization**. Nonetheless, the difference between accuracies in this case is not much significant as there is no big gap.

As a final result, we end up with:

- **Training accuracy: 91,63%**
- **Validation accuracy: 76.65%**

As in our case our validation set is the same as the test set, we conclude our model has **76,65% accuracy**.

5.1.3 Confusion matrix

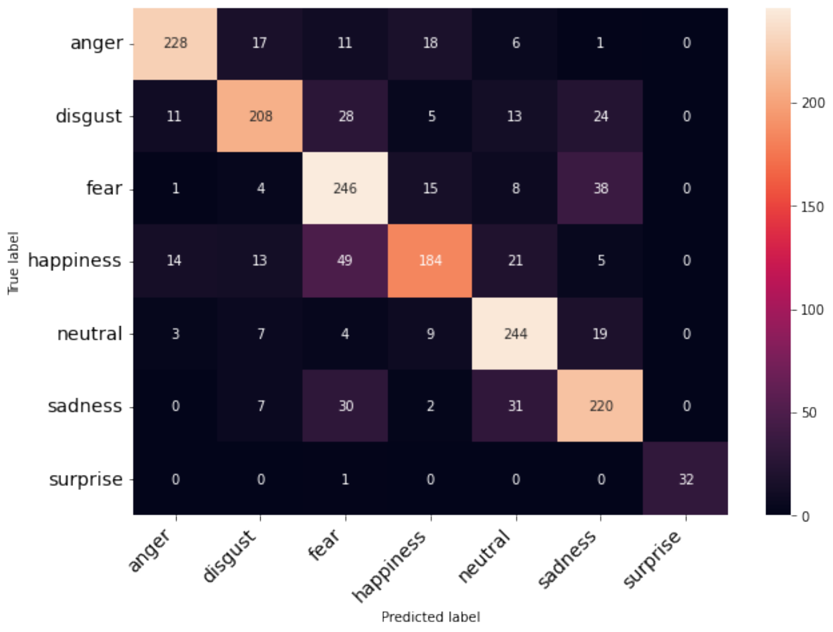


Figure 5.3: Confusion matrix of the model

Printing a confusion matrix is a good way to study in further detail how our model is behaving. We took a look at accuracy and loss curves but they did not give any information about *where* or *how* our model is failing predictions (or nailing them).

To do that, we generate a confusion matrix as follows:

1. With our model already trained, we call `model.predict()` on our **test set**.
2. Take the **true labels** of the test set and concatenate them with the **predicted labels** in an Numpy array or Pandas DataFrame.
3. Draw a confusion matrix which has one axis representing true labels and the other one predicted labels (there is no standard for choosing either X or Y axes for each of them). Drawing a heatmap as in Figure 5.3 helps visualization and clear interpretation.

How to interpret a confusion matrix? The main diagonal represents the **correct predictions**, as the true labels and predicted label coincides. Any other element in the matrix represents a **misprediction**. When the heatmap shows greater values in the main diagonal and low values on the rest (as in Figure 5.3) that is a **good indicator** of our model's performance.

In any case, what really provides useful and rich information about our model's behaviour that will help us comprehend how our model *understands* the world or what influences its decisions (a topic that has been increasing in popularity called XAI or Explainable AI [10]) are **the mispredictions**.

For instance, from the confusion matrix provided by our model, we could draw some conclusions:

- Negative emotions are often confused. For example, fear is confused by sadness in 38 occasions; disgust is confused for fear in 28 occasions; sadness is both confused for fear (30) and neutral (31).
- Surprise gets the most accurate predictions: it only failed one sample. This might be related to the fact that there were fewer surprise samples than other ones (as shown in Figure 3.5); or that surprise audio samples present very heterogeneous and distinguishable features that make it easy to predict.
- Happiness is the most mispredicted emotion: it gets confused by fear 49 times, and also for neutral, anger and disgust. This is for sure an anomalous demeanour, that would be worth to investigate. There could exist several hypothesis that could be analysed: maybe happiness samples have very similar characteristics in comparison to other emotions (e.g. pitch, tone, loudness, frequency); or it could be possible that audio samples for happiness were not enough representative; or that the features selected (MFCCs) fail to represent the samples faithfully.

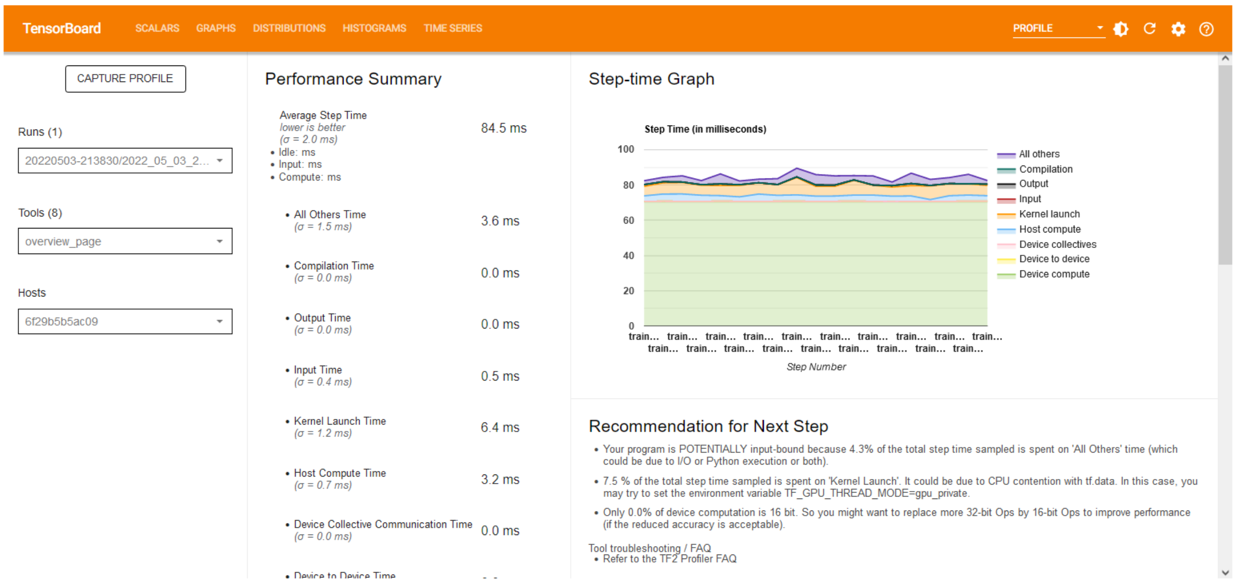
5.1.4 Testing model's execution time

Measuring how fast our model executes is crucial for this project, as it has to be rapid enough to build our 'near real-time system'. If deployed on *serverless*, the model performs bad in execution time, responses to the requests will be late and fail to arrive on time, thus causing a lot of overhead and destroying our idea of a streaming real-time system.

To measure time, two techniques were used:

- **Profiling with TensorBoard.** TensorBoard is a very powerful visualization tool embedded with TensorFlow. Used as a *callback* for our model, it generates useful logs to measure training time on each step (a **step** is the processing of one **single batch of size 32**). Also, as any other profiling tool in the market, it introduces *overhead* in time due to the time required to do the measurements and generate the logs.

  As shown in Figure 5.4, the average step time on the process of training was `84.5ms`. In the graph on the upper right side we can observe the breakdown of this averaged time into different tasks; fortunately in our case the majority of the time is invested in computing time (which indicates normal behaviour).



Runs (1)

20220503-213830/2022_05_03_2...

Tools (8)

overview_page

Hosts

6f29b5b5ac09

Performance Summary

Average Step Time *lower is better* *(σ = 2.0 ms)* • Idle: ms • Input: ms • Compute: ms	84.5 ms
• All Others Time *(σ = 1.5 ms)*	3.6 ms
• Compilation Time *(σ = 0.0 ms)*	0.0 ms
• Output Time *(σ = 0.0 ms)*	0.0 ms
• Input Time *(σ = 0.4 ms)*	0.5 ms
• Kernel Launch Time *(σ = 1.2 ms)*	6.4 ms
• Host Compute Time *(σ = 0.7 ms)*	3.2 ms
• Device Collective Communication Time *(σ = 0.0 ms)*	0.0 ms
• Device to Device Time	

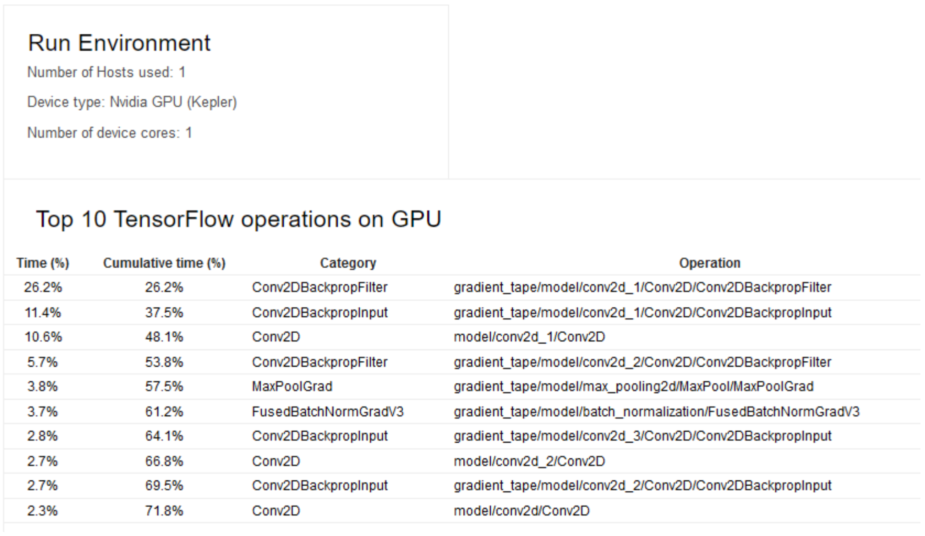
Recommendation for Next Step

- Your program is POTENTIALLY input-bound because 4.3% of the total step time sampled is spent on 'All Others' time (which could be due to I/O or Python execution or both).
- 7.5 % of the total step time sampled is spent on 'Kernel Launch'. It could be due to CPU contention with tf.data. In this case, you may try to set the environment variable TF_GPU_THREAD_MODE=gpu_private.
- Only 0.0% of device computation is 16 bit. So you might want to replace more 32-bit Ops by 16-bit Ops to improve performance (if the reduced accuracy is acceptable).

Tool troubleshooting / FAQ
- Refer to the TF2 Profiler FAQ

Figure 5.4: TensorBoard profiling results

Furthermore, in the following figure, TensorBoard provides us with an analysis in which operations does the model invest most of its time, which we can clearly see are convolution and backpropagation operations.



Run Environment

Number of Hosts used: 1

Device type: Nvidia GPU (Kepler)

Number of device cores: 1

Top 10 TensorFlow operations on GPU

Time (%)	Cumulative time (%)	Category	Operation
26.2%	26.2%	Conv2DBackpropFilter	gradient_tape/model/conv2d_1/Conv2D/Conv2DBackpropFilter
11.4%	37.5%	Conv2DBackpropInput	gradient_tape/model/conv2d_1/Conv2D/Conv2DBackpropInput
10.6%	48.1%	Conv2D	model/conv2d_1/Conv2D
5.7%	53.8%	Conv2DBackpropFilter	gradient_tape/model/conv2d_2/Conv2D/Conv2DBackpropFilter
3.8%	57.5%	MaxPoolGrad	gradient_tape/model/max_pooling2d/MaxPool/MaxPoolGrad
3.7%	61.2%	FusedBatchNormGradV3	gradient_tape/model/batch_normalization/FusedBatchNormGradV3
2.8%	64.1%	Conv2DBackpropInput	gradient_tape/model/conv2d_3/Conv2D/Conv2DBackpropInput
2.7%	66.8%	Conv2D	model/conv2d_2/Conv2D
2.7%	69.5%	Conv2DBackpropInput	gradient_tape/model/conv2d_2/Conv2D/Conv2DBackpropInput
2.3%	71.8%	Conv2D	model/conv2d/Conv2D

Figure 5.5: Profiling tasks breakdown

- **Testing single samples.** Measuring training time is very useful and gives a lot of insights, however, due to overhead caused by profiling and measurements of steps (batches) instead of single samples does not allow to see the whole picture. For the task of testing single sample time, a manual task was developed. A thousand calls to `model.predict()` were made on a single sample and the mean of the times was computed. We get the following results:

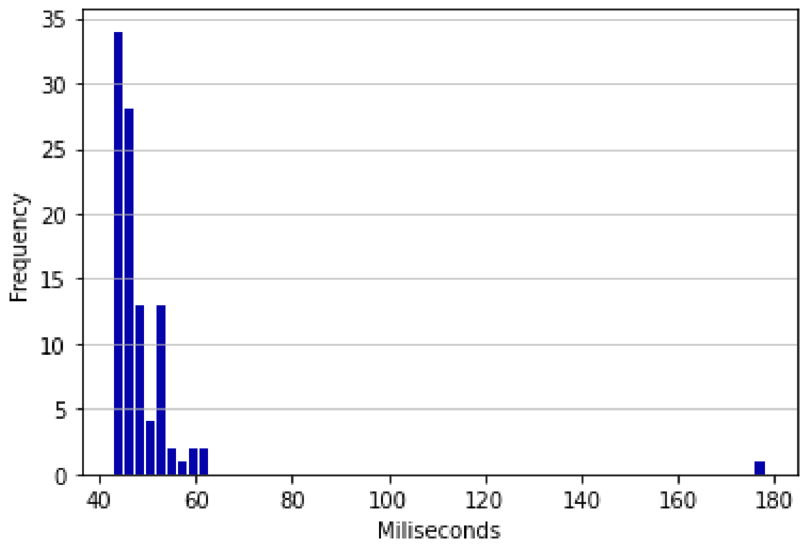


Figure 5.6: Histogram of execution times

- **Mean:** 49.07ms
- **Standard deviation:** 13.62ms
- **Skewness:** 8.50

From the histogram and the value of skewness we can infer that these execution times do not follow a normal distribution, but a right-skewed distribution. It is also worth of mention that in several runs of this test, the histogram always shows the *outlier* at the end around 180ms. This is due because the first run of the model always lasts longer, and then it is kind of *cached*.

This analysis was made **not taking into account preprocessing tasks** though. It is also interesting to know the times appending preprocessing tasks, as when deployed in *serverless*, raw audio comes as input, and then MFCCs need to be calculated and normalization computed inside the code. Let's see how much overhead is added along with preprocessing.

From the results from Figure 5.7 we can clearly see that preprocessing tasks slow the process a lot. This might be in great part due to I/O operations that come from the file load of `librosa.load()` function and the computation of MFCCs. Some samples even have taken up to 1 whole second.

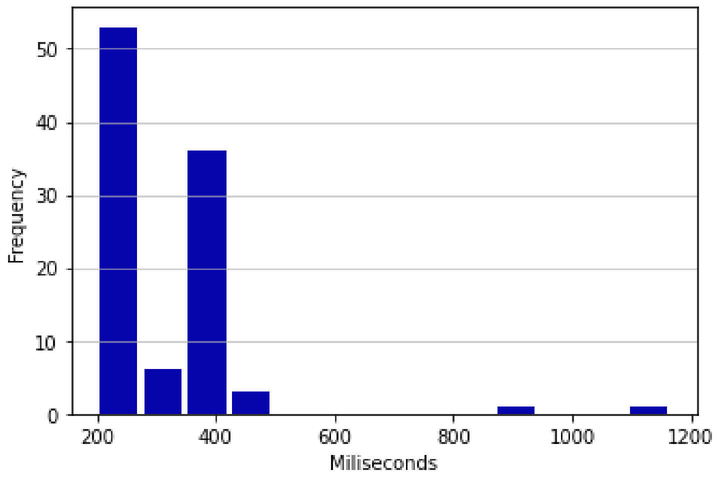


Figure 5.7: Histogram of execution times with preprocessing

- **Mean:** 308.52ms
- **Standard deviation:** 133.90ms
- **Skewness:** 3.36

Chapter 6

Conclusions

Developing a Speech Emotion Recognition model can be a tricky task, as it is not meant to be a fixed and objective classification task such as object detection (a car *is* a car in any case): emotions and their interpretation are subjective. As stated in the datasets section, even when hand labelling the audio clips, almost every one of them was labelled with 7 different emotions by the participants. Emotions in the voice also vary with the language, regional culture, intonation etc. (for instance Spanish people tend to use a flat tone whereas British people tend to give importance to intonation in phrases). In this aspect, I come to the conclusion that for making a very well-performing SER model, a lot of quality and varied data (e.g. different languages, intonations, lots of actors for the same language...) and a very well fine-tuned model should be clear requirements. In such a case, it would even be useful to acquire a deep understanding on human emotions in the voice through the knowledge of psychology experts and speech therapists, in order to extract the optimal features and make a good analysis of our data before using it.

Also, I discovered the usefulness of using a distributed architecture such as our Cloud *serverless* for deploying a model, as it facilitates a lot of tasks such as the resources' allocation, scalability problems or the creation of an execution environment or API to use the model. It also reasonably reduces costs, because you only pay for execution time, better than paying for the use of a machine that won't be used 24 hours. With a simple Amazon EC2 machine, not only a lot of configuration and set up would have to be made but also the cost would have been much greater. The *serverless* option has allowed this project to have modularity and compatibility, as the code for the processing and prediction of an audio clip is very well packaged and isolated from the code that the user uses to record the voice and call the APIs, thus allowing both parts to be used even in different versions of Python with no major issue. However, it also presents some disadvantages such as the overhead in latency that increases with the size of the architecture and the overhead introduced by the start of the execution environment (cold start).

Chapter 7

Future work

As with every project, it is clear that a lot of things could have been done better but they were not because of matters of time or lack of knowledge. Below, I present some improvements that might be done as future work:

- Developing a custom data pipeline with Python scripts using a versioning control system such as DVC[1] for better modularization of the steps, rather than having a single notebook.

- Improving the model by finding and using more quality data sources (datasets), preferably in different languages and lengths (not being limited to 2-second long clips). This would certainly improve model's generalization and performance, in exchange for more resources in the training process, as we will see in the next bullet point.

- Making training phase faster and more efficient, training in the Cloud with some Deep Learning AMIs such as the ones that AWS offer and machines with multiple GPUs such as Amazon P3 machines. This would make training and experimenting with hyper-parameters much faster than using our local machine or in this case, the Google Colab free tier. This would also allow us to experiment with more complex models with deeper layers and to use more data in our experiments. Nevertheless, this would reasonably increase costs, as these machines are much more expensive.

- Try other *deep learning* models that are used in literature such as LSTMs, simple RNNs or 1D-CNNs.

- Explore different features in audio apart from MFCCs (e.g. zero crossing rate, pitch, spectral centroid etc.) and make feature selection tasks to find the most suitable ones for maximizing performance.

- Experiment with Transfer Learning techniques (pre-trained models for the task), and compare results.

[1]https://dvc.org/

- Allow for concurrent users in the AWS architecture. With the current architecture, consistency is not guaranteed with multiple users, only a single user is allowed. This is due to two reasons: requests from each user are not labelled or identified by any ID, so it is not possible to track a prediction response for a certain user; when there is a single SQS queue, even if responses were tracked with IDs, if one user retrieves the first element of the queue (even if it does not belong to him), it gets removed from the queue, and the correspondent user cannot recover it. Also, if one user sent a request at the end of the queue, it should wait till all the other users removed the other elements of the queue to get the prediction, increasing latency. This behaviours should be modified by modifying the architecture itself.

- Improve the GUI of the application and add more functionality such as a summary of the emotions through the recording.

List of Figures

List of Tables

Bibliography

[1] Jamil Ahmad, Khan Muhammad, and Sung Baik. Data augmentation-assisted deep learning of hand-drawn partially colored sketches for visual search. *PLOS ONE*, 12:e0183838, 08 2017.

[2] Safaa Allamy and Alessandro Lameiras Koerich. 1D CNN Architectures for Music Genre Classification, 2021.

[3] Alan Davis Babu. Artificial Intelligence vs Machine Learning vs Deep Learning (AI vs ML vs DL). `https://medium.com/@alanb_73111/artificial-intelligence-vs-machine-learning-vs-deep-learning-ai-vs-ml-vs-dl-e6afb717743`, 2019.

[4] James Beswick. Operating Lambda: Performance optimization – Part 1. `https://aws.amazon.com/es/blogs/compute/operating-lambda-performance-optimization-part-1/`, 2021.

[5] Houwei Cao, David G. Cooper, Michael K. Keutmann, Ruben C. Gur, Ani Nenkova, and Ragini Verma. Crema-d: Crowd-sourced emotional multimodal actors dataset. *IEEE Transactions on Affective Computing*, 5(4):377–390, 2014.

[6] Km121220. Wikimedia Commons. Examples of ways to partition a dataset. dataset a only uses a training set and a test set. the test set would be used to test the trained model. dataset b depicts 3 subsets. for this option, the validation set would be used to test the trained model, and the test set would evaluate the final model., 2020. File: ML_dataset_training_validation_test_sets.png.

[7] Daphne Cornelisse. An intuitive guide to convolutional neural networks. `https://www.freecodecamp.org/news/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050/`, 2018.

[8] Council of European Union. REGULATION (EU) 2016/679 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation), 2016.
`https://eur-lex.europa.eu/legal-content/EN/TXT/?qid=1532348683434&uri=CELEX%3A02016R0679-20160504`.

[9] M. El-Alami, M Atwi, and Mohamed Ezzat. A proposed system to define student identity through sound recognition techniques. *Research Journal Specific Education*, 36:1066–1090, 10 2014.

[10] Explainable artificial intelligence. Explainable artificial intelligence — Wikipedia, the free encyclopedia, 2022. [Online; accessed 06-May-2022].

[11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, page 1097–1105, Red Hook, NY, USA, 2012. Curran Associates Inc.

[12] Pavan Kunchala. Functional API model with Tensorflow 2.0 (Y-network). `https://medium.com/analytics-vidhya/functional-api-model-with-tensorflow-2-0-y-network-ed59bfd810`, 2021.

[13] Margaret Lech, Melissa Stolar, Christopher Best, and Robert Bolia. Real-time speech emotion recognition using a pre-trained image classification network: Effects of bandwidth reduction and companding. *Frontiers in Computer Science*, 2, 2020.

[14] Steven R. Livingstone and Frank A. Russo. The ryerson audio-visual database of emotional speech and song (ravdess): A dynamic, multimodal set of facial and vocal expressions in north american english. *PLOS ONE*, 13(5), 05 2018.

[15] Derrick Mwiti. CNN Sentence Classification. `https://cnvrg.io/cnn-sentence-classification/`.

[16] Markus Mühlberger. What is Serverless. `https://code.tutsplus.com/tutorials/what-is-serverless--cms-30077`, 2017.

[17] Pratheeksha Nair. The Dummy's Guide to MFCC. `https://medium.com/prathena/the-dummys-guide-to-mfcc-aceab2450fd`, 2018.

[18] Papia Nandi. CNNs for Audio Classification. `https://towardsdatascience.com/cnns-for-audio-classification-6244954665ab`, 2021.

[19] José Padarian and Ignacio Fuentes. Word embeddings for application in geosciences: development, evaluation, and examples of soil-related concepts. *SOIL*, 5:177–187, 07 2019.

[20] Adrian Rosebrock. Keras vs. tf.keras: What's the difference in TensorFlow 2.0? `https://pyimagesearch.com/2019/10/21/keras-vs-tf-keras-whats-the-difference-in-tensorflow-2-0`, 2019.

[21] Sumit Saha. A Comprehensive Guide to Convolutional Neural Networks - the ELI5 way. `https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53`, 2018.

[22] Amazon Web Services. Amazon Web Services — Wikipedia, the free encyclopedia, 2022. [Online; accessed 12-May-2022].

[23] Nicola Strisciuglio, Manuel Lopez Antequera, and Nicolai Petkov. Enhanced robustness of convolutional networks with a push–pull inhibition layer. *Neural Computing and Applications*, 32:1–15, 12 2020.

[24] John Terra. Keras vs Tensorflow vs PyTorch: Key Differences Among the Deep Learning Framework. `https://www.simplilearn.com/keras-vs-tensorflow-vs-pytorch-article`, 2022.

[25] Arne Wolfewicz. Deep Learning vs. Machine Learning – What's the difference? https://levity.ai/blog/difference-machine-learning-deep-learning, 2022.

[26] Julian Wood. Introducing AWS Lambda Destinations. https://aws.amazon.com/es/blogs/compute/introducing-aws-lambda-destinations, 2019.

Appendix

Links to the code

The source code of this project can be found both in GitHub and Google Drive:

- GitHub repository[5]
- Google Drive[6]

Installation guide

This installation guide will help you to set up and get all the necessary tools to run the client program. However, this installation guide does not include instructions to run the notebook code to reproduce the experiments to develop the SER model, just the client app.

Prerequisites

- **Ubuntu 18.04** (probably compatible with other Linux distributions)
- **Python 3.8-dev** (comes with the installation script)

Clone or download the project

First of all, you'll have to **clone or download the repository** from GitHub manually or by typing:

```
$ git clone https://github.com/RobertFarzan/Speech-Emotion-Recognition-system.git
```

Setting up the execution environment

Once you have all the project files on your local machine, there are two ways of building the execution environment:

Using the `setup.sh` script

The script `setup.sh` contains all the necessary commands and dependencies to run the program, including a `python3.8-dev` installation, `python3-pip` and more. To run the script, open a shell, enter the project's root folder with `$ cd` command and just type:

[5]<https://github.com/RobertFarzan/Speech-Emotion-Recognition-system>
[6]<https://drive.google.com/file/d/1XobYLxcARE73EFwZ3VUr6Po7vum42ajh/view?usp=sharing>

```
$ source setup.sh
```

This will require `sudo` permissions. If the installation was successful, your current shell should be inside a Python virtual environment (venv) and you should see something like this:

```
$ (venv) yourusername@yourmachine:~/projectfolder
```

Otherwise, if the installation script failed, check the file to see if some package or library may conflict with your current set up. In this case, it might be more convenient to proceed with the manual installation shown in the next step.

Manual installation

In some cases, some dependencies or previous installations of the same software (e.g. another Python version) may conflict with the current installation, causing unexpected behaviour.

In such a case, it is best to install every dependency one by one.

1. Firstly, update your packages with

   ```
   $ sudo apt-get update
   ```

2. Secondly, install the following Python 3.8 **developer version**. It is important that it is the dev version instead of the usual installation because it includes some *header files* necessary for the project. If already have a Python 3.8 installation, it might be convenient to uninstall it to avoid conflicts.

   ```
   $ sudo apt-get install python3.8-dev $ sudo apt install libpython3.8-dev
   ```

3. Install the Python **venv** library to create the virtual environment necessary to run the program.

   ```
   $ sudo apt-get install python3.8-venv
   ```

4. Install the `pip` version for Python 3. If already installed, skip this step.

   ```
   $ sudo apt-get install python3-pip
   ```

5. Install the audio controllers necessary to record audio from the main script. If not installed, it will cause problems and errors when running the program.

   ```
   $ sudo apt-get install libasound-dev portaudio19-dev libportaudio2 libportaudiocpp0
   ```

6. Create a virtual environment with the same Python installation we did already.

   ```
   $ python3.8 -m venv venv
   ```

   IMPORTANT NOTE: if run with `python3` or `python` commands instead of `python3.8`, it might create the virtual env with another version you may have installed, causing conflicts later on.

7. Activate the virtual environment with

   ```
   $ source venv/bin/activate
   ```

8. Install the dependencies with `pip` inside the virtual environment.

   ```
   $ pip install –upgrade pip
   ```

   ```
   $ pip install -r requirements.txt
   ```

Running the program

Now we have all the necessary tools and dependencies installed, there are a couple of things to take into account before running the app.

1. **Always** activate the virtual environment before running. If you don't see `(venv)` in your command prompt, it is most likely it is not activated. In such case, remember to do:

   ```
   $ source venv/bin/activate
   ```

2. Set the *PYTHONPATHenvironmentvariabletothe*`src`*folderoftheproject.Thiswillallowthescr*

You can do this in two ways:

- Temporarily. This would require to execute the following command each time you would like to run the program:

  ```
  $ cd /path/to/project_root (not literal)
  ```

  ```
  $ export PYTHONPATH="$PWD/src"
  ```

- Permanently. This would only need to be done once. Take the full path of the src folder inside the project's root folder (e.g `/home/user/project/src`). Then open the `.bashrc` file located in your home directory with your favorite text editor and do as follows:

  ```
  $ nano $HOME/.bashrc (or vi, vim etc.)
  ```

  ```
  > write "export PYTHONPATH=/home/path/to/project/src" at the end (not
  literal). Save the file and exit.
  ```

  ```
  $ source $HOME/.bashrc
  ```

3. Check the PYTHONPATH variable is set correctly by running `$ echo $PYTHONPATH`. If the output is empty, something went wrong.

Now everything is set to go. You just have to run the program to see the magic with:

```
$ python3.8 app.py
```

Choose life.

Robert Farzan Rodríguez
10/05/2022
Ult. actualización May 26, 2022
LaTeX lic. LPPL & powered by TEFLON CC-ZERO