ARTIFICIAL INTELLIGENCE AND DATA ENGINEERING
project report for the examination
of
Symbolic and Evolutionary Artificial Intelligence

Student:
Alessandro Diana

AA. 2024/2025

Contents

List of Figures

1 Introduction

1.1 Motivation for Keyword Spotting (KWS)

Keyword Spotting (KWS) refers to the task of detecting specific spoken words within an audio stream. It plays a crucial role in modern speech-based interfaces, as it allows devices to continuously listen for predefined commands without requiring a full speech recognition system. Compared to large-scale automatic speech recognition (ASR), KWS systems are lightweight, faster, and suitable for real-time applications. This makes them particularly valuable for embedded systems, mobile devices, and IoT environments, where computational resources and power consumption are limited.

1.2 Applications of KWS

KWS has become a fundamental component of voice assistants, such as Google Assistant, Amazon Alexa, and Apple Siri, enabling the device to respond to "wake words" like "Ok Google" or "Alexa". Beyond consumer electronics, KWS is also applied in smart home automation, robotics, healthcare monitoring, and hands-free control in industrial or automotive contexts. These applications demonstrate how KWS provides a natural and intuitive interface between humans and machines, reducing the need for manual interaction and enabling seamless user experiences.

1.3 Goals of the Project

The primary goal of this project is to design and train a neural network capable of performing KWS for a set of predefined voice commands intended for remote control of multimedia devices such as televisions or MP3 players. The model is trained on a dataset of short audio clips, each containing a single spoken word, to learn to recognize different pronunciations and speaker variations. Within the dataset, particular emphasis is placed on detecting a subset of target command words (e.g., Start, Stop, Pause, Forward, Backward), while other words present in the dataset are treated as non-relevant and should be ignored by the system.
A second goal of the project is to evaluate the trained model under more realistic conditions by testing it on longer audio recordings that contain a mixture of background noise, relevant commands, and non-relevant words. This allows for assessing the robustness of the system, particularly its ability to correctly identify the target commands while avoiding false activations caused by noise or irrelevant speech.
All the developed code and documentation can be found at this link:
https://github.com/Arzazrel/Project_SEAI_25.

2 Background

2.1 Overview of Keyword Spotting Techniques

KWS is the task of detecting predefined words or short phrases within a continuous audio stream. Unlike large-vocabulary ASR systems, which aim to transcribe arbitrary speech, KWS is designed to recognize a limited set of commands with high accuracy and low computational cost.
Early KWS systems were based on template matching or hidden Markov models (HMMs), where speech features were aligned against pre-defined word models. While effective, these approaches were often computationally expensive and less robust to variability in speakers, background noise, or recording conditions.
With the rise of machine learning, techniques based on support vector machines (SVMs) and later DNNs became dominant. These methods leverage discriminative models trained directly on speech features, achieving higher accuracy and better generalization. More recently, end-to-end neural approaches that operate directly on raw waveforms or spectrogram representations have emerged, enabling compact and efficient models suitable for real-time deployment on embedded devices.

2.2 Convolutional Neural Networks for Audio Classification

Convolutional Neural Networks (CNNs), originally developed for image processing tasks, have proven highly effective for audio classification, including keyword spotting. Their strength lies in their ability to exploit the time–frequency structure of speech features such as spectrograms or Mel-frequency cepstral coefficients (MFCCs). By applying convolutional filters, CNNs can learn local patterns that correspond to phonetic events or spectral characteristics, which are crucial for distinguishing between different keywords.
Compared to fully connected networks, CNNs require fewer parameters and exhibit greater robustness to shifts in time and frequency, making them well-suited for processing speech signals. Pooling operations further provide invariance to small temporal or spectral variations, while deeper architectures allow for hierarchical feature extraction. In the context of keyword spotting, CNNs strike a balance between accuracy and efficiency: they outperform traditional methods while remaining lightweight enough for real-time execution on resource-constrained devices. For this reason, CNN-based architectures are widely used as the foundation of modern KWS systems, often serving as the baseline before exploring more advanced recurrent or attention-based models.

3 Dataset

3.1 Description of the Google Speech Commands Dataset

The dataset used in this project is the Google Speech Commands V2, which can be downloaded from Kaggle at the following link: Google Speech Commands V2 Dataset.
The Google Speech Commands Dataset V2 is a widely used benchmark for keyword spotting tasks. It contains approximately 105,000 one-second audio recordings of 35 short spoken words, collected from thousands of different speakers, thus ensuring significant variability in pronunciation, accent, and recording conditions. Each audio file is sampled at 16 kHz and stored in WAV format, providing a standardized and high-quality corpus for training and evaluating speech recognition models.
In addition to the main set of keywords, the dataset also includes recordings of background noise, as well as non-speech utterances (such as "silence" and the "unknown" class, which groups together words outside the target vocabulary). These additional samples are essential for training robust keyword spotting models, as they allow the system to better discriminate between relevant commands and irrelevant or noisy inputs.
From a practical perspective, each of the 35 keyword classes contains on average, 3,000 samples, although the exact distribution is not perfectly balanced. The audio clips are exactly 1 second long, and the dataset is organized into a clear folder structure where each folder corresponds to a specific keyword. The inclusion of noise files (e.g., white noise, pink noise, and background speech) allows for more realistic training and evaluation scenarios.
Due to its balanced design, standardized format, and inclusion of noise conditions, the Speech Commands V2 dataset is particularly suitable for the development and benchmarking of lightweight neural networks aimed at low-latency, on-device speech recognition applications.

3.2 Background Noise in the Dataset

The Google Speech Commands dataset V2 includes a dedicated folder named '_background_noise_', which contains long audio recordings intended to simulate realistic environmental noise. These files can be used either as background signals to increase robustness during training or as negative examples in classification tasks. The folder contains the following recordings:

- doing_the_dishes: 1 minute and 35 seconds of kitchen-related noise, such as clattering dishes and water sounds.
- dude_miaowing: 1 minute of a man meowing and related domestic sounds.
- exercise_bike: 1 minute of mechanical and ambient noise from an exercise bike in use.
- pink_noise: 1 minute of artificially generated pink noise.
- white_noise: 1 minute of artificially generated white noise.
- running_tap: 1 minute of water running from a tap, producing a continuous stream-like sound.

White noise and pink noise are two types of artificially generated broadband noises that are often used in audio processing experiments.

- White noise is characterized by having equal energy distributed across all frequencies, resulting in a constant 'hissing' sound similar to static. It is particularly useful for testing system robustness against uniform interference.

- Pink noise has equal energy per octave, meaning that lower frequencies carry more energy compared to higher ones. This makes it closer to the distribution of frequencies typically observed in natural environments, and thus more realistic for simulating background noise.

The inclusion of such heterogeneous background recordings (domestic sounds, environmental noise, and artificial broadband noise) provides valuable resources for training and evaluating keyword spotting models. By introducing these noises during training, models can learn to distinguish target keywords even in non-ideal conditions, thereby improving their generalization and robustness in real-world applications.

3.3 Selection of Classes

For this project, we want to achieve voice recognition of specific commands to be used to control an MP3 player or TV. From the dataset, I decided to implement the commands listed below, along with the possible operations associated with them.

- 'forward': move forward within the track/movie
- 'backward': go backward within the track/movie
- 'stop': stop audio/movie track playback
- 'go': start playing the audio track/movie
- 'up': turn up the volume
- 'down': turn down the volume
- 'left': moves to the next audio track/movie
- 'right': moves to the previous audio track/movie

All other words belong to the class 'unknown'.
With this division, approximately 75% of the dataset will belong to the 'unknown' class, with the remaining 25% split almost equally among the 7 command classes (backward and forward have half as many samples as the other classes).

3.4 Preprocessing Steps

Before extracting features for use in neural network training, the raw audio files undergo several essential preprocessing steps to ensure consistency, quality, and compatibility:

- Resampling: All audio files are uniformly resampled to 16 kHz, matching the native sampling rate of the Speech Commands V2 dataset, which ensures consistent frequency representation across samples and compatibility with the expected input rate for feature extraction.
- Mono Conversion: Any audio with multiple channels (if present) is converted to a single-channel (mono) format. This ensures homogeneity in input data and avoids potential discrepancies from stereo or multitrack sources. This also reduces computational complexity and memory usage, which is essential for real-time processing on low-power devices. Stereo information (spatial cues, panning) adds redundancy without improving recognition performance, as keyword detection primarily relies on spectral features.

- Trimming or Padding: The recordings are standardized to a duration of 1 second. Shorter files are padded (with silence), and longer files are trimmed or windowed to ensure uniform duration across all samples for batch processing in neural networks.
- Amplitude Normalization: The audio signals are normalized (e.g., zero-mean, unit variance, or peak normalization) to reduce variation in volume levels between recordings while preserving the integrity of the speech content.

These preprocessing steps ensure that the subsequent feature extraction stage (e.g., computing spectrograms or Mel-frequency cepstral coefficients) receives clean, consistent, and well-formatted input data, improving model learning and generalization.

3.5 Audio Chunking of Long Files

In order to ensure consistency across the dataset and to preserve useful information from extended recordings, a dedicated preprocessing procedure was developed through the implementation of the script 'chunk_audio_in_ds.py'. This tool addresses the issue of audio files exceeding the standardized duration of 1 second, which is the reference length adopted for training the KWS model.

The procedure is governed by two global parameters: the first defines the target duration to which all audio files must conform (1 second in this work), while the second specifies a maximum threshold above which a file is no longer truncated but instead divided into multiple segments of the required length (set to 2 seconds in this case).

The script systematically traverses all class folders within the dataset and evaluates each .wav file. When a file exceeds the defined threshold, it is segmented into consecutive chunks of the specified duration. Each resulting segment is exported as an independent .wav file, whose name is derived from the original filename with the addition of a numerical suffix indicating its sequential order. For example, a five-second audio file named 'white_noise' is partitioned into five one-second segments saved as: 'white_noise_0', 'white_noise_1', 'white_noise_2', 'white_noise_3', and 'white_noise_4'. All generated files are stored in the same directory as the original, thereby preserving the correspondence with the original class label.

This preprocessing step is essential to avoid the information loss that would occur if long recordings were truncated to their first second. By retaining the entirety of the original signal through segmentation, the dataset provides a richer and more representative training set, ultimately contributing to improved performance and generalization of the neural network.

3.6 Audio Composition Program

Given the objective of designing a network capable of recognizing specific commands embedded within irrelevant words or background noise, the trained models must be tested in more realistic conditions than isolated one-second clips. To achieve this without relying on real-time recordings, longer audio files were generated by concatenating multiple words, noises, and commands.
For this purpose, the program 'create_audio_track.py' was developed. The program concatenates short audio samples from the dataset to construct a single longer track. The way in which this track is later processed and analyzed is described in subsequent chapters.
At initialization, the program scans the dataset, storing the available samples together with their corresponding class labels. If a sample belongs to one of the target command classes, it is retained as such; otherwise, it is assigned to the 'unknown' class, which includes both background noise and non-relevant words. The user is first asked to specify the number of audio segments to

concatenate. For each segment, the program requests the desired class and then selects a random sample from that class. The chosen samples are concatenated sequentially to create a long audio track in which each component is preserved in full.
Additionally, the program prompts the user to define the name of the output file and generates a log file describing the composition of the created audio track. This log specifies, for each time window, the corresponding class label as well as the number of recognized commands contained within the track. In this project, the window length, equal to the duration of the original dataset samples, is set to 1 second.

4 Methodology

4.1 Feature Extraction

In order to train the KWS networks, it is necessary to extract spectral features from the raw audio signals contained in the dataset. These features transform one-dimensional waveforms into two-dimensional time–frequency representations, which can be interpreted as images and directly processed by CNNs. Such representations capture both temporal and spectral characteristics of speech, providing a richer input space than raw audio.
The preprocessing begins by standardizing the sampling rate to 16 kHz and the duration of each file to 1 second, ensuring uniformity across the dataset. Subsequently, a sequence of transformations is applied to compute increasingly compact and perceptually relevant features.

4.1.1 Short-Time Fourier Transform (STFT)

The Short-Time Fourier Transform (STFT) is used to obtain the initial time–frequency representation of the signal. The audio waveform is divided into overlapping windows, and a Fast Fourier Transform (FFT) is applied to each segment. In this project, the STFT was computed with a frame_length of 256 samples (16 ms at 16 kHz) and a frame_step of 128 samples (8 ms hop). This configuration results in a spectrogram with dimensions [num_frames, num_freq_bins], where the number of frequency bins is given by (frame_length / 2) + 1 = 129. The magnitude spectrogram is used, as it represents the amplitude distribution of the signal over time and frequency.

4.1.2 Mel-Spectrogram

While the STFT provides a linear frequency representation, it does not align well with human auditory perception. To address this, the Mel scale is applied, which projects the linear spectrum onto a set of perceptually spaced frequency bands. In this work, 64 Mel bands were used, derived by applying a triangular filterbank to the magnitude spectrum.
Two frequency bounds are specified when computing the Mel-spectrogram:

- fmin = 80 Hz: Frequencies below 80 Hz are typically dominated by low-frequency noise or vibrations, and they carry little relevant information for speech. The human auditory system is also less sensitive in this range. Removing these components reduces noise and computational complexity.
- fmax = 7600 Hz: The maximum frequency is chosen slightly below the Nyquist limit (8 kHz for 16 kHz sampling). This avoids potential aliasing and focuses on the frequency range that contains most of the phonetic information. Human speech is particularly rich in the 300 Hz – 7.5 kHz range, which includes the formants and other features necessary for distinguishing phonemes and words.

The Mel-spectrogram thus provides a compact and perceptually motivated representation of speech, which is particularly suitable for KWS tasks.

4.1.3 Mel-Frequency Cepstral Coefficients (MFCCs)

For additional feature compression, MFCCs can be derived from the Mel-spectrogram. The process consists of three main steps:

- Compute the Mel-scaled spectrum.
- Apply the logarithm to capture relative energy variations.

- Perform a Discrete Cosine Transform (DCT) to decorrelate the features and obtain a compact set of coefficients.

The MFCCs capture the spectral envelope of speech, which encodes information such as formants and vowel structure.
The first coefficient (c_0) corresponds to the overall energy of the signal.
The next few coefficients (c_1-c_{12}) represent coarse spectral shapes and are the most relevant for distinguishing speech sounds.
Higher-order coefficients capture fine spectral variations, which are often dominated by noise and less useful for classification.
For this reason, KWS and speech recognition systems typically use only the lower-order coefficients, as they provide a compact yet highly discriminative representation of speech.

4.2 Data Augmentation

To enhance the robustness and generalization capability of the KWS model, data augmentation techniques were employed. Data augmentation artificially increases the diversity of the training set by introducing controlled variations into the input data. This is particularly important in speech recognition tasks, where background noise, temporal misalignment, and speaker variability can significantly impact model performance. In this project, three augmentation strategies were implemented: the addition of background noise, time shifting, and pitch shifting. Although these methods were integrated into the processing pipeline, they have not yet been experimentally tested, and their evaluation is planned as future work.

4.2.1 Background Noise Addition

This technique consists of adding low-level background noise to the original audio samples. The objective is to simulate realistic acoustic environments, where speech signals are often corrupted by ambient noise such as room reverberation, electronic interference, or other environmental sounds.

- Advantages: increases robustness to noisy conditions and prevents the model from overfitting to clean, idealized recordings.
- Limitations: if the added noise is too strong, it may obscure the essential phonetic content of the speech, reducing classification accuracy.

4.2.2 Time Shifting

Time shifting involves displacing the speech signal within the fixed one-second analysis window. This simulates situations in which the keyword is pronounced slightly earlier or later in the recording.

- Advantages: improves the model's tolerance to temporal variability, ensuring that detection is not overly dependent on keyword alignment.
- Limitations: excessive shifting may lead to truncation or distortion of the speech signal, thereby reducing intelligibility.

4.2.3 Pitch Shifting

Pitch shifting alters the fundamental frequency of the speech signal without affecting its duration. This augmentation technique addresses natural variability across speakers, such as differences between male and female voices or tonal fluctuations in pronunciation.

- Advantages: enhances generalization across speakers with different vocal characteristics, making the model more adaptable to unseen voices.
- Limitations: large pitch modifications may result in unnatural-sounding speech, which could reduce the usefulness of the augmented data.

In summary, these data augmentation techniques target three major sources of variability in real-world audio: background noise, keyword misalignment, and speaker pitch differences. Their adoption is expected to increase the resilience of the trained KWS model, although their actual impact has yet to be validated experimentally.

4.3 CNN Architecture Design

This section describes the CNN architectures developed within this project, along with the rationale behind their design. All networks were implemented as different versions of the same base class, referred to as SirenNet, located in the '/code/net/_classes' directory. The networks were progressively refined to evaluate how general-purpose and task-specific design choices affect KWS performance.

4.3.1 SirenNet v.0 – Baseline Network

The first network architecture was inspired by GoogLeNet, an established CNN model originally designed for image recognition tasks. The network consists of an initial convolutional layer followed by max pooling, four inception modules with additional max pooling operations after the first and third modules, and a final global average pooling layer. The output is then processed through a fully connected layer with dropout regularization before reaching the classification layer.
In terms of size, the network comprises 72,785 trainable parameters (284 KB).
This version was deliberately included as a baseline model to assess how a network optimized for visual pattern recognition performs on spectrogram-based audio data. Although not specifically designed for speech or KWS, it provides a valuable benchmark, allowing comparisons with subsequent architectures that incorporate domain-specific considerations for audio processing.

4.3.2 SirenNet v.1 – Domain-Specific CNN for KWS

The second network introduces architectural elements explicitly designed for the characteristics of audio spectrograms. Its design is based on parallel convolutional branches with rectangular filters, which selectively operate along either the frequency or the time dimension. This separation allows the network to capture complementary information:

- filters spanning multiple time frames but a single frequency band emphasize temporal dynamics,
- filters spanning multiple frequency bands but a single time frame emphasize spectral structure.

The network architecture consists of:

- First layer: two parallel convolutional branches, one operating along frequency and one along time. This stage is followed by a square convolution and max pooling along the time axis.
- Second layer: again two parallel convolutional branches, with larger filters than in the previous layer. These are followed by a square convolution and max pooling along the frequency axis.
- Third layer: two inception modules, followed by square max pooling.
- Fourth layer: a single inception module, followed by global average pooling.
- Final stage: a small, dense layer before the classification output.

The model comprises 95,561 trainable parameters (373 KB).
The motivation behind this design is to process spectrogram data in a hierarchical manner. Early layers use small rectangular filters to capture fine-grained local details without prematurely mixing information across time and frequency. Intermediate layers employ larger rectangular filters to model longer-term dependencies. Finally, square filters are applied in deeper layers to jointly integrate temporal and spectral information, enabling the network to capture complex correlations.
The inclusion of inception modules further enhances the network's ability to detect spectrogram patterns at multiple scales. Parallel convolutions with different receptive field sizes allow the model to capture:

- short-term variations (e.g., transient bursts);
- harmonic structures spanning multiple frequency bands;
- long-term temporal correlations.

This design reflects the intuition that, in speech signals, frequency and time carry distinct but complementary forms of information. Preserving them separately in the initial stages, before merging at higher levels, increases the network's discriminative power for keyword spotting.

4.4 Training Setup

Different strategies can be adopted to manage the training process of KWS networks, depending on the tradeoff between memory consumption and training speed. The simplest and most direct method would consist of reading the audio files from the dataset, computing their spectrograms (Mel or MFCC), and storing them entirely in memory. While this approach minimizes training latency, it is extremely memory-intensive and can easily saturate the available VRAM, especially with large datasets or high-resolution spectrograms.
An alternative approach is to store only the paths of the audio files in memory and compute the spectrograms at runtime during training. Although this method significantly reduces memory consumption, it substantially increases training time, as the spectrogram computation is more than eight times slower than the raw audio reading process.
In this project, a hybrid solution was implemented to achieve a balance between efficiency and resource usage. Specifically, the audio files are preprocessed only once when the dataset is initially read, and the resulting Mel or MFCC representations are stored on disk. During training, only

the paths to these precomputed features are stored in memory, and the data are loaded directly from disk when needed. This approach greatly reduces memory consumption while avoiding the overhead of recomputing the features at each epoch, at the cost of slightly higher disk usage. For the experiments carried out in this project, Mel spectrograms were adopted, while MFCCs were not employed in the final training.
The training procedure allows the user to configure several hyperparameters, including the batch size, the maximum number of epochs, and the patience parameter for early stopping. Early stopping is implemented based on the validation loss: if the validation loss does not improve for a specified number of consecutive epochs, training is halted to prevent overfitting and unnecessary computation.
The network is trained using the Adam optimizer with the categorical cross-entropy loss function, and accuracy is employed as the main performance metric. Furthermore, the training pipeline supports repeated runs of the same model in order to compute average performance metrics, thereby increasing the robustness and reliability of the evaluation.
At the end of training, several visualizations are generated to analyze the network's behavior. These include accuracy and loss curves for the training, validation, and test sets across epochs, as well as the confusion matrix, which provides detailed insight into the classification performance for each keyword.

5 System for Keyword Spotting in Long Audio

5.1 Strategy for Long Audio Analysis

To evaluate the networks without relying on real-time microphone acquisition, an offline protocol was adopted based on long audio tracks composed of samples taken from the dataset. The goal is to identify the presence of target commands within a stream that also contains irrelevant words and background noise, thus replicating realistic operating conditions.
Since the model was trained on segments of 1s, the analysis of longer tracks relies on a segmentation into windows of identical duration, thereby ensuring consistency between the training and inference phases. This protocol is directly transferable to a real-time scenario with minimal modifications to the acquisition subsystem (e.g., circular buffering, block-based reading, streaming inference).

5.2 Sliding Window and Chunking Approach

The long track is divided into windows of 1s with a hop size of 0.5s. This configuration provides a balance between temporal coverage (a larger number of observations) and computational cost (lower redundancy), while reducing the risk of missing commands that fall across window boundaries.
Each window is converted into the same spectral features used during training and then classified by the network, yielding a probability distribution over the classes. It is expected that, in correspondence with a command, the probability of the relevant class will exhibit a bell-shaped profile: increasing as the window approaches the command, peaking when the command is centered, and decreasing thereafter. This scheme is equivalent to a temporal scan (sliding window) of the entire audio track, allowing the estimation of command likelihoods at each instant (hop size).

5.2.1 Border Effects

A relevant aspect concerns the management of audio track borders. In the implemented system, the first window always covers the interval [0, 1s], while subsequent windows progress with a step of 0.5s until the end of the signal. To ensure consistency, the last window is positioned with center in (audio_size − hop_size) and not with center in audio_size. In this way, it always covers a full 1s segment, thus avoiding the generation of incomplete windows at the end of the signal.
If the remaining portion of the track is not perfectly divisible into 1s windows, the system can operate in two modes: truncation(discarding the last part) or padding (completing the segment with zeros). The latter is set as the default to preserve the input size required by the model. This ensures segmentation uniformity and minimizes the impact of border effects on recognition performance.

5.3 Post-processing for Keyword Detection

To limit spurious activations, a confidence threshold is applied: a command is recognized only if the probability of its class exceeds 90%. However, in the presence of a command spanning multiple consecutive windows, the threshold alone may yield multiple adjacent activations of the same class.
To mitigate this effect, a cooldown mechanism (refractory period) is introduced. Once an event is detected, the system inhibits further detections of the commands for a predefined number of windows. The cooldown represents a tunable hyperparameter: larger values reduce duplicate

detections but may suppress valid, closely spaced commands, while smaller values increase sensitivity but expose the system to repeated false positives.
The simplest implementation of cooldown consists of inhibiting the recognition of commands for a predefined number of windows, or even preventing the network from performing predictions on those windows. This approach is straightforward in both logic and implementation, and with an appropriately chosen cooldown value, it can yield good results.
A more advanced implementation restricts the cooldown only to the recognition of the command that has just been detected. In this way, the risk of multiple detections of the same command is mitigated without compromising the recognition of different commands occurring in rapid succession.
Both methods have their own contexts of applicability. For instance, the simpler method may be preferable in scenarios where, either by design choice or for other practical reasons, the recognition of commands in rapid succession or overlap—whether identical or different—is not desirable. In such cases, the simpler cooldown implementation would not only be lighter but also more effective.
The second case, for example, is particularly effective in scenarios where it is important to accurately recognize rapid sequences of different commands without skipping elements of the sequence and without introducing redundant detections. In both cases, determining the appropriate cooldown value for the specific application is crucial to achieving optimal recognition performance.
In this project, both versions have been implemented.
The confidence threshold and cooldown improve precision by reducing false positives, while maintaining adequate recall in scenarios involving closely spaced or partially overlapping commands.

6 Implementation Details

6.1 Tools and Frameworks

The implementation of this project was carried out using the Python programming language. Python was chosen because of its versatility and the wide availability of libraries specifically designed for machine learning and deep learning tasks.
The main framework employed was TensorFlow. The choice of TensorFlow was driven by my prior experience with this library, which allowed for faster prototyping, straightforward debugging, and an efficient integration of neural network training on GPU hardware. Keras, included within TensorFlow, provided a higher-level interface that simplified model construction and experimentation.
On Windows, the available TensorFlow release with GPU support was limited to version 2.10, requiring CUDA 11.2 and cuDNN 8.1.
To overcome these limitations and ensure access to more recent releases, the Windows Subsystem for Linux (WSL) was configured via Miniconda. In this environment, TensorFlow 2.19 was employed together with CUDA 12.9 and cuDNN 9.3. The configuration is not intended as a recommendation but rather as a clarification of the setup adopted during the project.
Regarding hardware resources, the training was conducted on a laptop equipped with an Intel(R) Core(TM) i7-10870H CPU @ 2.20GHz, 16 GB of RAM, and an NVIDIA RTX 3060 GPU with 6 GB of VRAM. The use of the GPU was essential to significantly reduce training times, although the configuration of TensorFlow with NVIDIA drivers proved to be a delicate process that required adherence to the official installation guidelines[1].

6.2 Codebase Structure

The codebase of the project was organized to ensure modularity, reproducibility, and clarity in the workflow. The repository is structured into several directories, each with a specific role:

- dataset: folder stores both raw and preprocessed data, ensuring separation between original inputs and their transformed counterparts (e.g., Mel spectrograms).
- code: folder contains the Python programs implementing preprocessing, data augmentation, neural network architectures, and training pipelines.
- model: folder stores trained network models in various formats (.keras, .h5). It is subdivided into train_hdf5/, which includes intermediate models saved at checkpoints, and trained, which contains the final models ready for inference in keyword spotting tasks.
- results: folder stores the outputs of dataset analysis and the performance metrics of the trained networks, including plots and evaluation results.

In this project, particular attention was paid to maintaining a modular structure, where each functionality is implemented in a dedicated Python program. This design choice was made to improve readability, reusability, and debugging efficiency.
Within the 'code' folder, the following components can be found:

- net_classes: subfolder containing the implementation of all neural network classes used in the project.

[1] `https://www.tensorflow.org/install/pip?hl=it#windows-native`

- audio_classifier.py: designed to process audio files longer than one second, analyzing them to detect spoken commands. The results are shown both in the command line and through graphical outputs such as plots and heatmaps.
- chunk_audio_in_ds.py: checks whether the dataset contains audio samples longer than one second and, if so, splits them into shorter segments suitable for training.
- create_audio_track.py: concatenates dataset samples to generate longer audio tracks, according to user-defined specifications.
- dataset_analysis.py: provides a concise analysis of the dataset, highlighting its key characteristics.
- fit_KWS_model.py: gathers all functionalities related to network training, performance evaluation, and visualization of results.

This modular approach allowed the project to be developed in a structured and scalable way. Each script is responsible for a single task, ensuring clarity of purpose while also enabling independent testing and extension of the system.

7 Results and Evaluation

7.1 Training and Validation Curves

In this section, we analyze the training process and the performance obtained for the different network architectures. The implemented training program allows the network to be trained multiple times in succession, after which the performance data collected are aggregated to compute averages and generate the corresponding performance curves. This strategy ensures statistically more robust results, enabling more reliable comparisons and a consolidated view of the actual performance, rather than being influenced by the variability of a single training run.

Due to the limited hardware resources available and the long training times required (ranging from 5:30 to 9:30 hours per run), the training process was necessarily constrained. All training sessions were performed with the following parameters: batch size of 32, a maximum of 200 epochs, and an early stopping criterion of 10 epochs based on the validation loss. The results of all training experiments are stored in the folder 'results/train'. Both network versions developed for this project were trained once individually and then repeated (five or three times, respectively) to obtain more stable and statistically meaningful results.

7.1.1 SirenNet Version 0

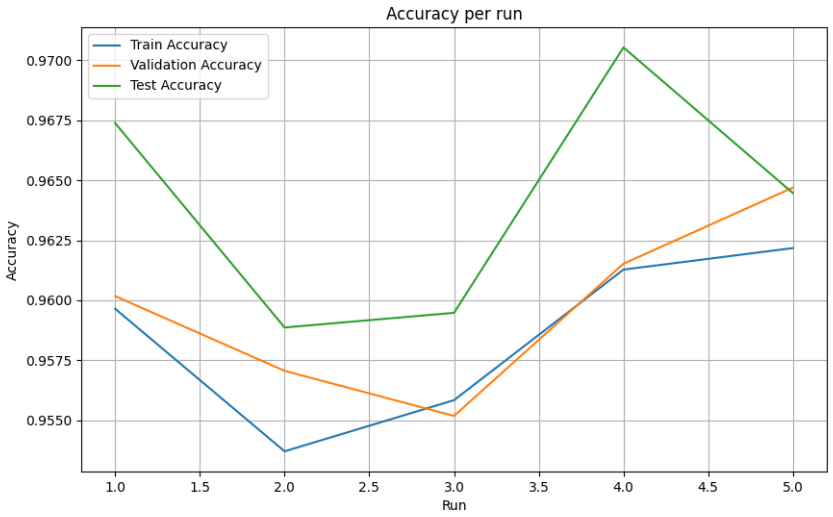

Figure 1: Line graph relating to the accuracy for SirenNet version 0.

This subsection reports the results of five training runs conducted with SirenNet version 0, an architecture originally designed for image recognition tasks. The accuracy and loss curves (fig. 1 and fig. 2) show a fluctuating behavior, characterized by occasional spikes. Overall, the three curves follow a broadly similar trend, although the test set tends to deviate more noticeably compared to the training and validation sets. The differences across runs remain relatively small, staying within a range of less than 2% in accuracy.

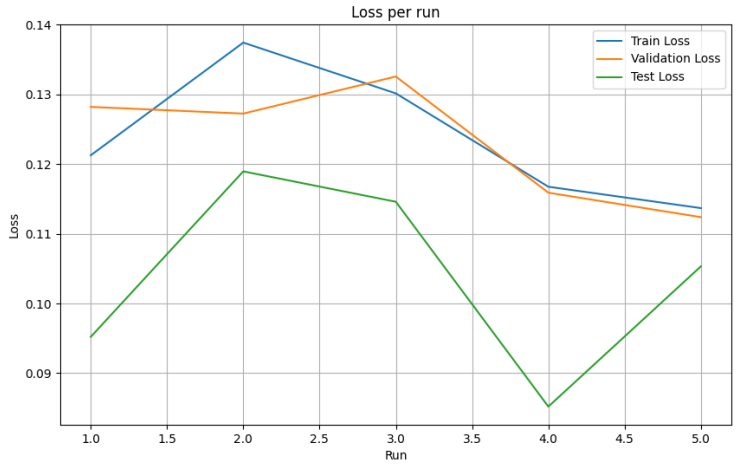

Figure 2: Line graph relating to the loss function for SirenNet version 0.

Interestingly, the test set metrics are almost always slightly better than those observed on the training and validation sets. The training set generally exhibits the weakest performance among the three, while the validation set follows a trajectory similar to the training curve but with marginally higher values. On average, each run terminated between the 45th and 55th epoch due to early stopping.
These observations indicate that SirenNet v0, while capable of learning some discriminative features, struggles to achieve stable convergence in the audio domain and exhibits less consistent behavior across training runs.

7.1.2 SirenNet Version 1

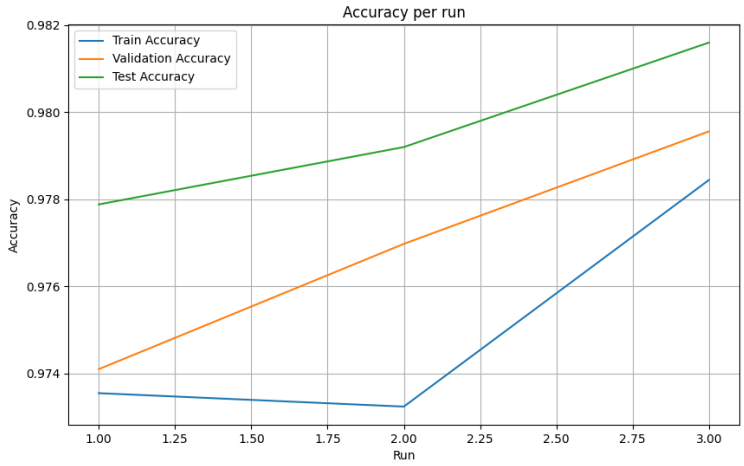

Figure 3: Line graph relating to the accuracy for SirenNet version 1.

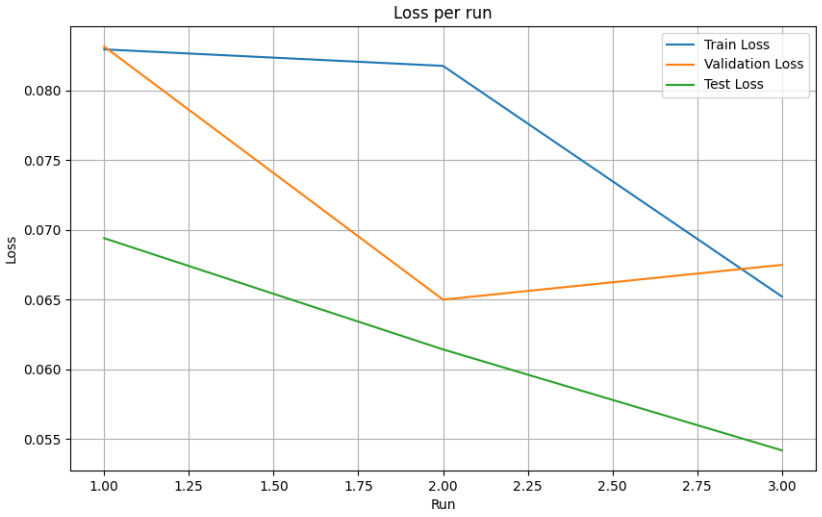

Figure 4: Line graph relating to the loss function for SirenNet version 1.

This subsection reports the results of three training runs conducted with SirenNet version 1, an architecture specifically optimized for speech and audio recognition tasks. The accuracy and loss curves (fig. 3 and fig. 4) show a much smoother and more linear trend, reflecting a steady and consistent improvement across epochs. In general, the three curves follow a similar pattern (except for the loss, where the training and validation curves occasionally exhibit opposite behaviors), while the test set curve diverges slightly but predictably from the others. The differences across runs are minimal, remaining within a range of less than 1% in accuracy.

As observed in SirenNet v0, the training set metrics tend to be the weakest, while the validation set is usually slightly better and closely aligned with the training set. The test set metrics consistently outperform both, demonstrating the strong generalization ability of this architecture. On average, each training session lasted between 60 and 70 epochs before early stopping was triggered.
Of particular note is the first training run, which extended up to 100 epochs (before being manually stopped due to an earlier training limit), achieving a test set accuracy of **98.68%**. This confirms the excellent capability of SirenNet v1 to adapt to the keyword spotting task, with high stability and strong predictive performance.

7.1.3 Conclusions

The comparative analysis of the training and validation curves highlights the superiority of SirenNet v1 over its predecessor. While SirenNet v0, inherited from an image recognition framework, struggled to adapt to the audio domain and showed instability in its learning dynamics, SirenNet v1 consistently demonstrated steady improvements across multiple runs, with higher stability, lower variance, and significantly better accuracy. The differences in early stopping behavior further confirm that the optimized architecture not only generalizes more effectively but also sustains reliable performance throughout extended training. These findings validate the choice of SirenNet v1 as the reference model for subsequent evaluation tasks and practical applications.

7.2 Confusion Matrix

In this section, we analyze the confusion matrices obtained from the two different versions of the network, evaluating performance on a per-class basis. It is important to note that the 'unknown' class comprises slightly more than 75% of the samples in the dataset. In contrast, the command-related classes each represent approximately 3.6%, except for 'forward' and 'backward', which each account for 1.5%. Due to this class imbalance, it is crucial to consider not only overall accuracy but also other metrics such as *precision*, *recall*, and the *F1-score*. These metrics are defined as follows for a given class I:

$$\text{Precision}_i = \frac{\text{TP}_i}{\text{TP}_i + \text{FP}_i}, \quad \text{Recall}_i = \frac{\text{TP}_i}{\text{TP}_i + \text{FN}_i}, \quad F1_i = 2 \cdot \frac{\text{Precision}_i \cdot \text{Recall}_i}{\text{Precision}_i + \text{Recall}_i}$$

dove:

- TP_i = True Positives of class i (number of samples of class i correctly classified)
- FP_i = False Positives of class i (number of samples from other classes incorrectly classified as i)
- FN_i = False Negatives of class i (number of samples of class i incorrectly classified as another class)

These metrics are particularly useful in imbalanced datasets, as they provide a more nuanced view of class-level performance than accuracy alone. The macro-averaged F1-score, which gives equal weight to each class regardless of its frequency, is especially informative in our case.

7.2.1 SirenNet Version 0

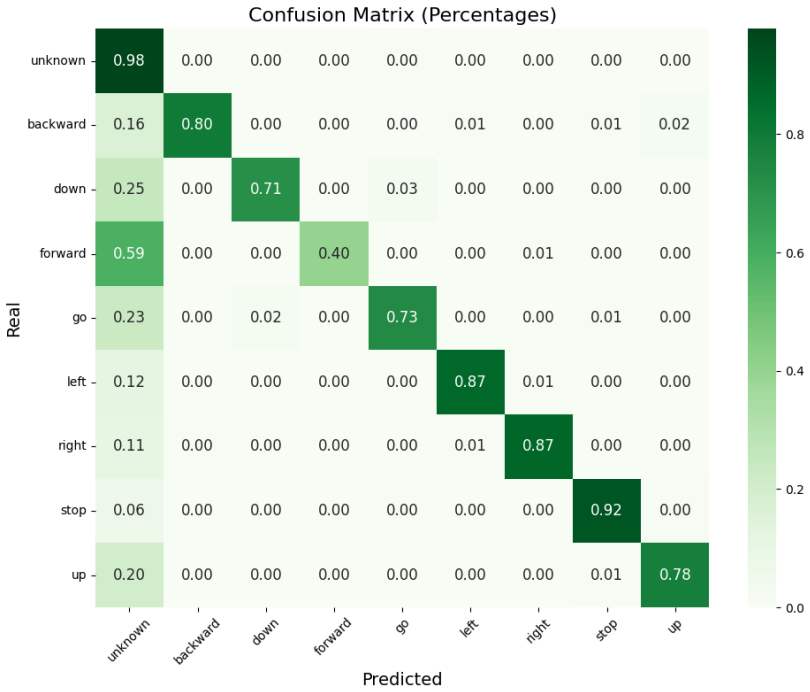

Figure 5: Heatmap of the confusion matrix relating to SirenNet version 0.

The confusion matrix (shown in 5) for SirenNet version 0 highlights that the 'unknown' class, being the majority, is recognized most accurately. Among the command classes, 'go' achieves the highest accuracy at 92%, followed by 'left' and 'right' at 87%. The class with the lowest accuracy is 'forward' at 40%, indicating that the network often misclassifies it as 'unknown'. A general trend observed is that most misclassifications for command classes fall into the 'unknown' category, while confusion between different commands is minimal. The most notable example is between 'go' and 'down', where 3% of 'down' samples were predicted as 'go', and 2% vice versa. Other misclassifications between commands are very rare, typically under 1–2%.

In the following section, we present and analyze the per-class performance metrics for the evaluated model. These metrics provide a detailed understanding of how well each class is recognized, highlighting potential weaknesses that may be obscured by overall accuracy. The metrics and their results are:

- 'unknown': Precision 0.942, Recall 0.978, F1 = 0.960
- 'backward': Precision 0.922, Recall 0.801, F1 = 0.857
- 'down': Precision 0.919, Recall 0.715, F1 = 0.804
- 'forward': Precision 0.896, Recall 0.399, F1 = 0.553
- 'go': Precision 0.853, Recall 0.734, F1 = 0.789
- 'left': Precision 0.913, Recall 0.865, F1 = 0.888
- 'right': Precision 0.906, Recall 0.875, F1 = 0.890
- 'stop': Precision 0.906, Recall 0.925, F1 = 0.915
- 'up': Precision 0.889, Recall 0.781, F1 = 0.832

Now show the aggregate metrics with Macro average (equal weight to each class):

- Precision 0.905
- Recall 0.786
- F1 = 0.832

In an imbalanced dataset like ours, the macro F1-score (0.832) is more representative of overall model quality than accuracy alone, as it accounts for the performance across all classes, including minority ones.
These values confirm the considerations made above. Among the command classes, 'forward' exhibits the lowest recall (0.399), highlighting difficulty in reliably detecting this particular command, while other classes, such as 'go', 'left', and 'right' maintain both high precision and recall. Overall, most misclassifications occur between command classes and 'unknown', rather than between distinct commands, indicating the model rarely confuses different commands with one another.

7.2.2 SirenNet Version 1

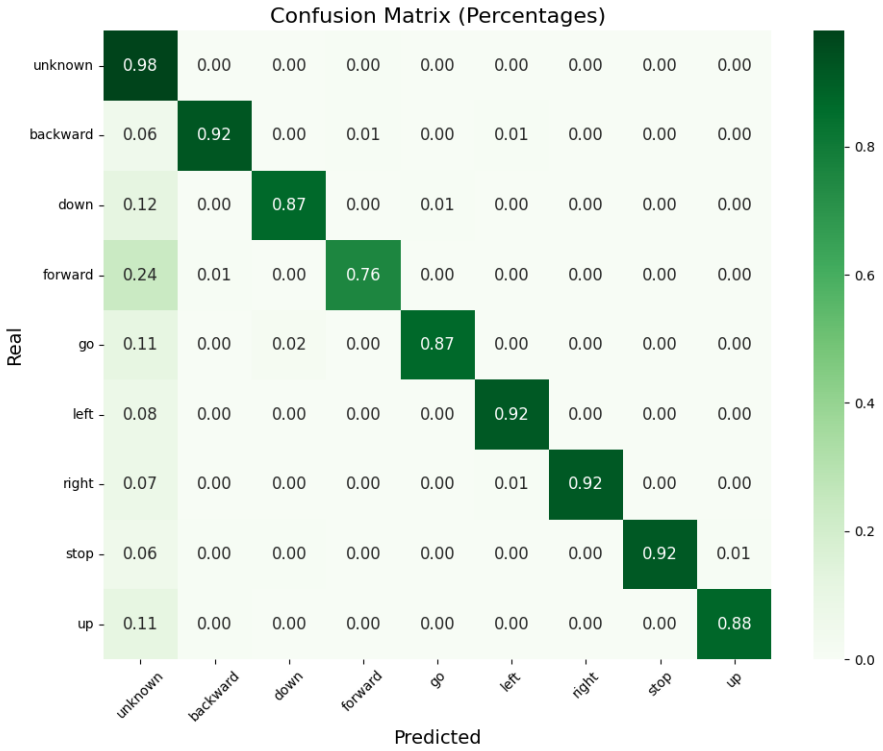

Figure 6: Heatmap of the confusion matrix relating to SirenNet version 1.

The confusion matrix (shown in 6) for SirenNet version 1 highlights that the 'unknown' class, being the majority, is recognized most accurately. Among the command classes, 'go','backward','left', and 'right' achieve the highest accuracy at 92%. The class with the lowest accuracy is 'forward' at 76%. A general trend observed is that most misclassifications for command classes fall into the 'unknown' category, while confusion between different commands is minimal. The most notable example is between 'go' and 'down', where 1% of 'down' samples were predicted as 'go', and 2% vice versa. Other misclassifications between commands are very rare, typically under 1%.

In the following section, we present and analyze the per-class performance metrics for the evaluated model. The metrics and their results are:

- 'unknown': Precision 0.9681 Recall 0.9809, F1 = 0.9745
- 'backward': Precision 0.8825, Recall 0.9242, F1 = 0.9029
- 'down': Precision 0.9266, Recall 0.8697, F1 = 0.8972
- 'forward': Precision 0.75, Recall 0.7576, F1 = 0.7538
- 'go': Precision 0.9312, Recall 0.8681, F1 = 0.8985
- 'left': Precision 0.9279, Recall 0.9185, F1 = 0.9232

- 'right': Precision 0.944, Recall 0.9159, F1 = 0.9297
- 'stop': Precision 0.9811, Recall 0.9195, F1 = 0.9493
- 'up': Precision 0.9378, Recall 0.8758, F1 = 0.9057

Now show the aggregate metrics with Macro average (equal weight to each class):

- Precision 0.9166
- Recall 0.8922
- F1 = 0.9037

Compared to the previous version, all the aspects discussed earlier show clear improvements. In particular, it can be observed that:

- all classes achieve equal or better accuracy than in version 0;
- among the commands, the maximum accuracy remains at 92%, but it is now reached by three commands instead of just one ('go'), while all other commands improved and do not fall below 87%, with the only exception of the class 'forward', which reaches 76%;
- the class 'forward' remains the most challenging to classify, but its performance has nearly doubled compared to the previous network, marking a substantial improvement;
- mismatching between commands, which was already very limited in version 0, has been further reduced and almost completely disappeared in version 1.

7.2.3 Comparison Between SirenNet Version 0 and Version 1

When comparing the two network versions, it becomes evident that SirenNet version 1 substantially outperforms version 0 across almost all evaluated metrics. The macro F1-score increased from 0.832 to 0.9037, representing a significant gain in overall classification quality. This improvement is particularly important given the imbalanced dataset, as it highlights better recognition of minority classes without sacrificing the performance of the majority class ('unknown').
Another important point is the reduced variance between classes: unlike version 0, where certain commands, such as 'forward', performed disproportionately worse, version 1 achieves a more uniform performance distribution, ensuring robust recognition across all classes.
Overall, SirenNet version 1 not only demonstrates greater accuracy but also greater robustness and reliability, making it a superior architecture for real-world keyword spotting applications.

In summary, SirenNet version 1 demonstrates clear advancements in robustness, balance, and overall recognition quality, making it a superior choice for real-world keyword spotting applications.

7.3 Evaluation on Long Audio Tracks

The evaluation on long audio sequences aims to assess the effectiveness and robustness of the proposed keyword spotting system in scenarios that closely resemble real-world usage. This section presents different experimental setups designed to highlight specific aspects of the system, including the impact of cooldown strategies, the ability to handle consecutive commands correctly, and the performance on extended audio tracks with varying levels of command density.

In addition to recognition accuracy, particular attention is devoted to inference times and segmentation overhead, to provide a comprehensive view of both detection reliability and computational efficiency.
The evaluation of long audio tracks was carried out using the SirenNet version 1 architecture. Specifically, the model employed is `'SirenNet_1.keras'`, which achieved an accuracy of 98.68% on the test set. This trained model is available in the project repository under the `'model/trained'` directory.
All the long audio tracks used in this section are available within the project repository on GitHub, under the 'dataset/long_audio' directory.
All the results and visualizations used in this section are available within the project repository on GitHub, under the 'results/audio_classifier' directory.

7.3.1 Cooldown Example on a Single Command

In this subsection, I evaluate the effectiveness of the previously introduced cooldown technique. As anticipated, once a command is detected, it may happen that in subsequent windows the same command is recognized again, since its confidence score remains above the threshold. Such redundant detections may lead to multiple spurious outputs, potentially resulting in significant issues in practical applications. To prevent this, a cooldown period is applied, i.e., a defined number of windows during which no further detections are allowed after a command has been identified.

For this experiment, we consider short audio tracks of 3/5 seconds, each containing a single target command located in the central segment (1s-2s), while all other windows correspond to the 'unknown' class. The goal is to verify whether multiple spurious detections occur, to test the ability of the cooldown mechanism to suppress them, and to determine the minimum number of inactive windows required to achieve the desired outcome.

For clarity, in this and subsequent subsections, each analyzed track is accompanied by two visualizations: a line plot and a heatmap representing the confidence values for all command classes on each generated window. It should be noted that the system explicitly recognizes only the target commands when their confidence exceeds the threshold. All remaining windows are treated as belonging to the 'unknown' class and are excluded from the output, regardless of the confidence score of that class.

The analyzed audio tracks are:

- 'track_down_0': containing the command 'down',
- 'track_go_0': containing the command 'go',
- 'track_left_0': containing the command 'left'.

Their composition is as follows:

- 'track_down_0': [unknown (0s–1s), unknown (1s–2s), down (2s–3s), unknown (3s–4s), unknown (4s–5s)] – number of commands in the track: 1
- 'track_go_0': [unknown (0s–1s), go (1s–2s), unknown (2s–3s)] – number of commands in the track: 1
- 'track_left_0': [unknown (0s–1s), left (1s–2s), unknown (2s–3s)] – number of commands in the track: 1

The line plots and heatmaps (shown in images 7 8 9 10 11 12) remain identical for cooldown values of 0 and 1, since the predictions on individual windows are unaffected. The difference emerges only in the program's output logs, where the cooldown mechanism suppresses redundant detections.

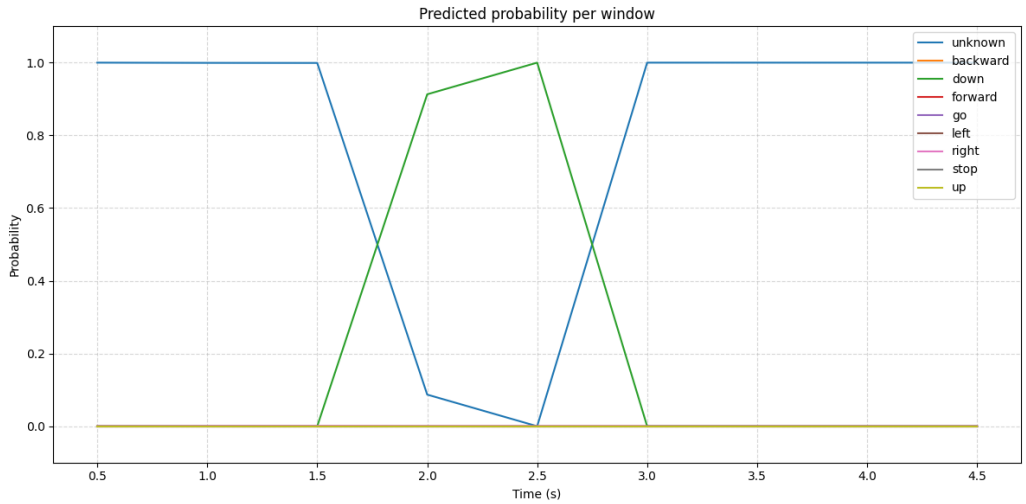

Figure 7: Line graph relating to the classification of 'track_down_0' audio.

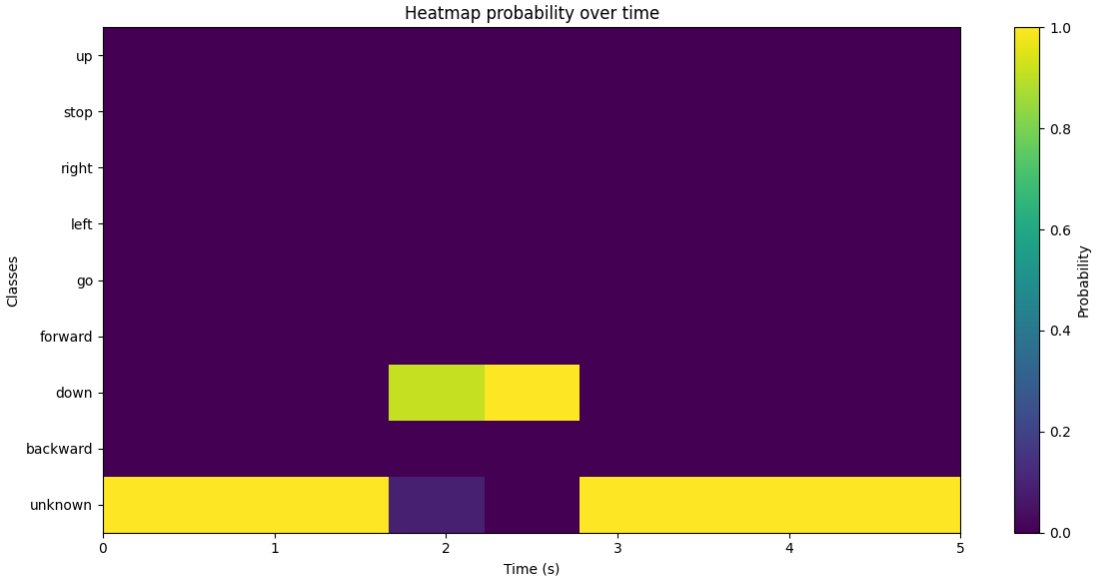

Figure 8: Heatmap relating to the classification of 'track_down_0' audio.

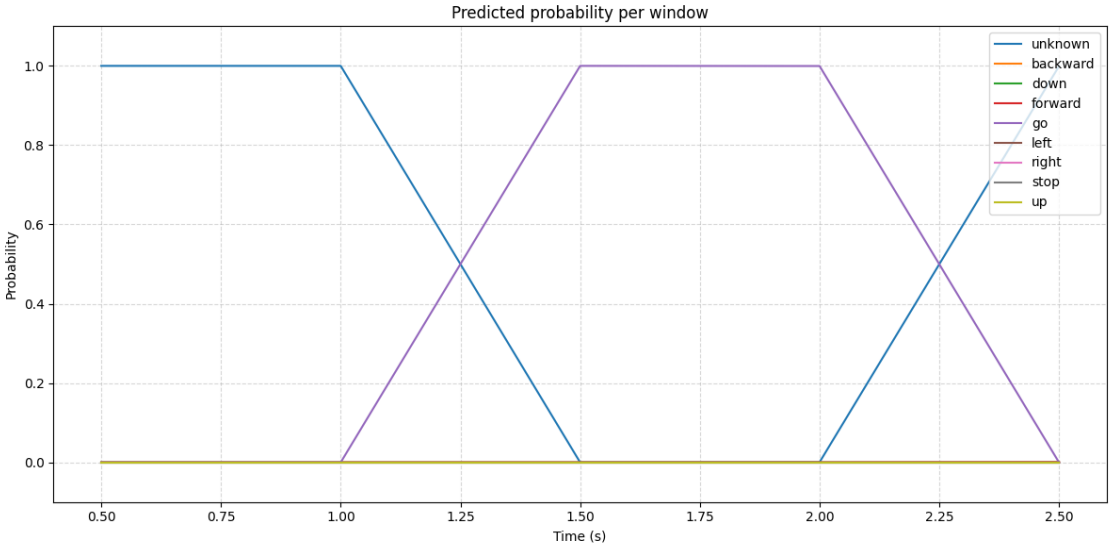

Figure 9: Line graph relating to the classification of 'track_go_0' audio.

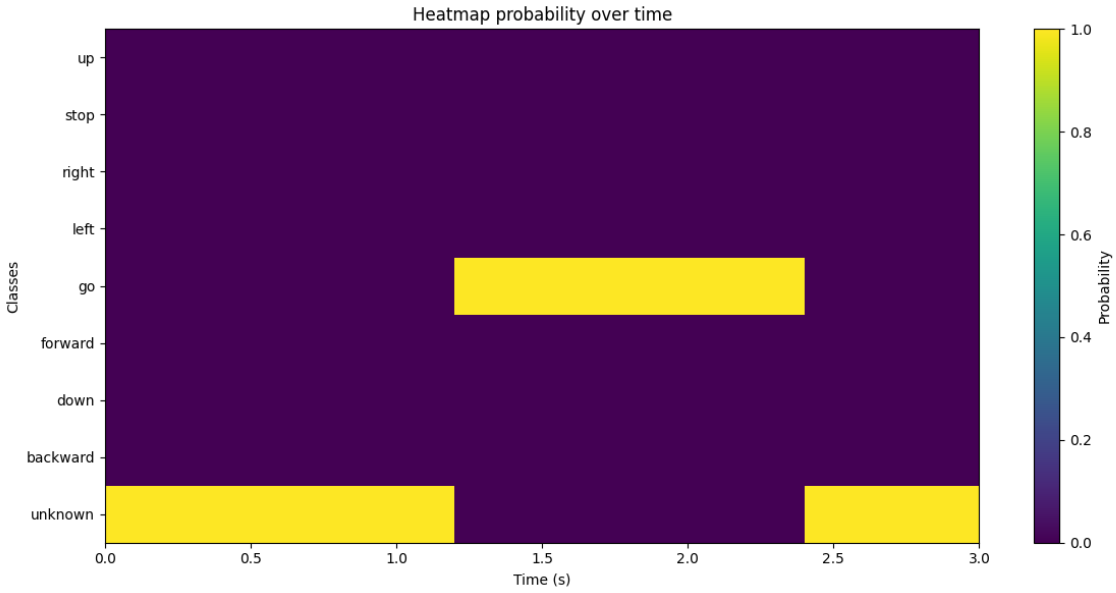

Figure 10: Heatmap relating to the classification of 'track_go_0' audio.

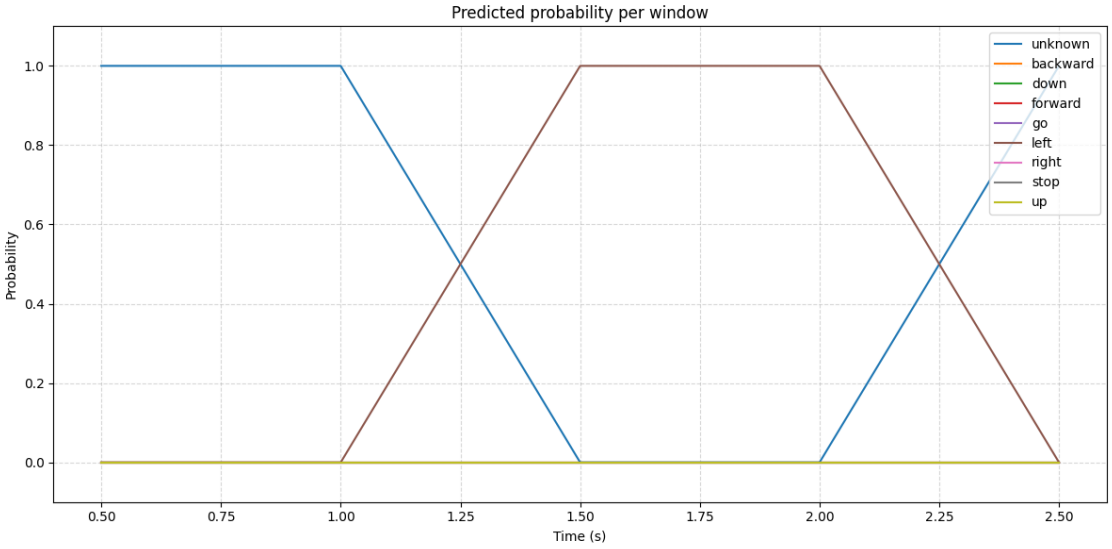

Figure 11: Line graph relating to the classification of 'track_left_0' audio.

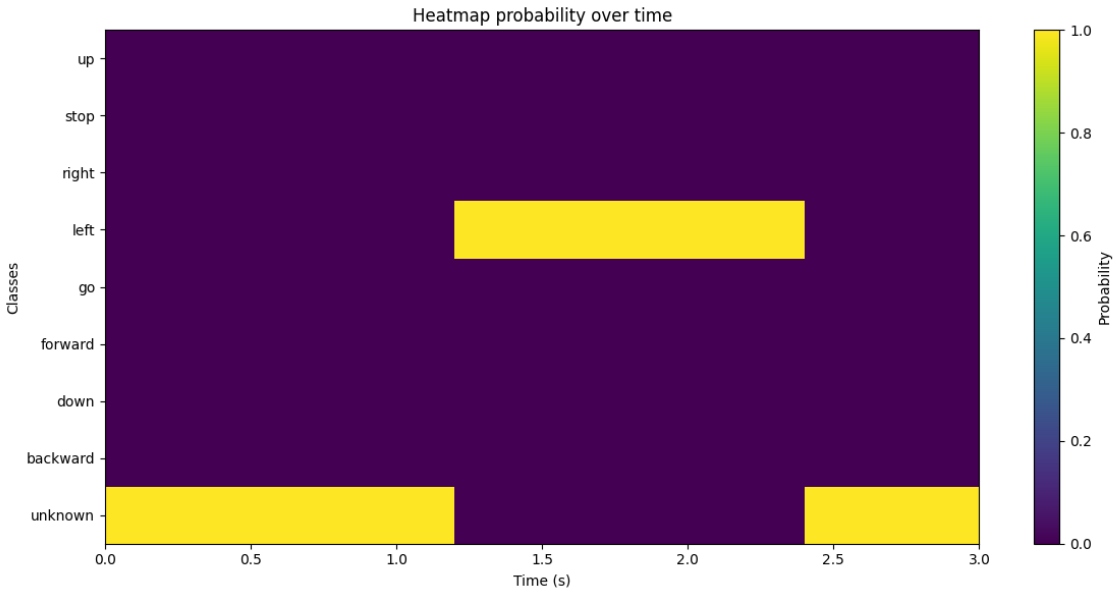

Figure 12: Heatmap relating to the classification of 'track_left_0' audio.

Results with cooldown = 0.

The images 13 14 15 show, in all cases, that the command is detected twice in succession. The maximum confidence is consistently reached in the window containing the target command. However, in two cases, very high-confidence detections also occur in adjacent windows:

- For 'down', the command is also detected in the preceding window with a confidence of

91.3%.

- For 'go', the command is also detected in the following window with a confidence of 99.96%.
- For 'left', the command is again detected in the subsequent window with a confidence of 100%.

```
---- Start predition ----
Audio input-> Name: ../dataset/long_audio/track_left_0.wav , lenght: 3.00 s, windows = 5
Window 2 [1000-2000 ms] -> left (100.00)
Window 3 [1500-2500 ms] -> left (100.00)
---- End predition ----
```

Figure 13: Output relating to the classification of 'track_left_0' audio with cooldown = 0.

```
---- Start predition ----
Audio input-> Name: ../dataset/long_audio/track_go_0.wav , lenght: 3.00 s, windows = 5
Window 2 [1000-2000 ms] -> go (100.00)
Window 3 [1500-2500 ms] -> go (99.96)
---- End predition ----
```

Figure 14: Output relating to the classification of 'track_go_0' audio with cooldown = 0.

```
---- Start predition ----
Audio input-> Name: ../dataset/long_audio/track_down_0.wav , lenght: 5.00 s, windows = 9
Window 3 [1500-2500 ms] -> down (91.30)
Window 4 [2000-3000 ms] -> down (100.00)
---- End predition ----
```

Figure 15: Output relating to the classification of 'track_down_0' audio with cooldown = 0.

Results with cooldown = 1.

With a cooldown of one window, the undesired effect is fully resolved: in all cases, the command is detected exactly once (as shown in 16 17 18). Notably, in the case of 'down', the detection occurs one window earlier than expected, but this does not pose a problem for the objectives of this project. This outcome confirms that a cooldown of one window represents the minimum value required to suppress multiple detections in these scenarios, while higher values would not provide additional benefits in this specific setting.

```
---- Start predition ----
Audio input-> Name: ../dataset/long_audio/track_left_0.wav , lenght: 3.00 s, windows = 5
Window 2 [1000-2000 ms] -> left (100.00)
Skip window.
---- End predition ----
```

Figure 16: Output relating to the classification of 'track_left_0' audio with cooldown = 1.

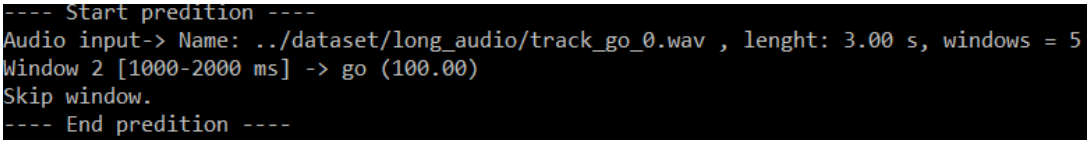

Figure 17: Output relating to the classification of 'track_go_0' audio with cooldown = 1.

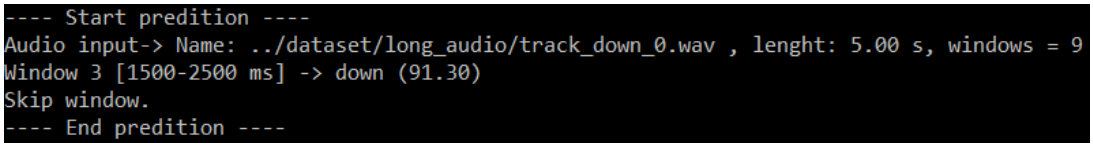

Figure 18: Output relating to the classification of 'track_down_0' audio with cooldown = 1.

These experiments confirm that the hypothesized issue of redundant detections is indeed a concrete problem and that the cooldown technique provides an effective solution. Alternative strategies, such as raising the confidence threshold to suppress adjacent detections, would be inadequate, since in two cases the secondary detections achieved extremely high confidence values (99.96% and 100%). Increasing the threshold to such levels would not only fail to eliminate spurious detections but could also risk missing genuine commands.

In conclusion, the cooldown mechanism, even in its simplest implementation, proves to be both effective and efficient. Having established its effectiveness in isolated commands, the next subsection will extend the analysis to more complex scenarios involving multiple commands in sequence, where the choice between a simple and an advanced cooldown strategy becomes more critical.

7.3.2 Comparison Between Classic and Advanced Cooldown

In this section, we analyze the difference in effectiveness between the two cooldown strategies, evaluated under different parameter values and in the presence of both sequential identical commands and sequential distinct commands.

For these experiments, two 5-second audio tracks containing a sequence of three commands (from second 1 to second 4) with background segments labeled as 'unknown' at the borders were used, together with two 4-second audio tracks containing a sequence of two identical commands (from second 1 to second 3), also with 'unknown' segments at the borders.

Each used track was processed using both the simple cooldown strategy and the advanced one, testing values of the cooldown parameter from 0 to 2 to observe the resulting differences in behavior. The line plots and heatmaps (show in images 19 20 21 22 35 36 37 38) remain identical for all cooldown methodologies and values.

Audio sequences containing two identical consecutive commands.
The analyzed audio tracks with the same commands are:

- 'multiple_same_command_down_0': containing the command 'down'.
- 'multiple_same_command_go_0': containing the command 'go'.

Their composition is as follows:

- 'multiple_same_command_down_0' → [unknown (0s–1s), down (1s–2s), down (2s–3s), unknown (3s–4s)] — total commands: 2
- 'multiple_same_command_go_0' → [unknown (0s–1s), go (1s–2s), go (2s–3s), unknown (3s–4s)] — total commands: 2

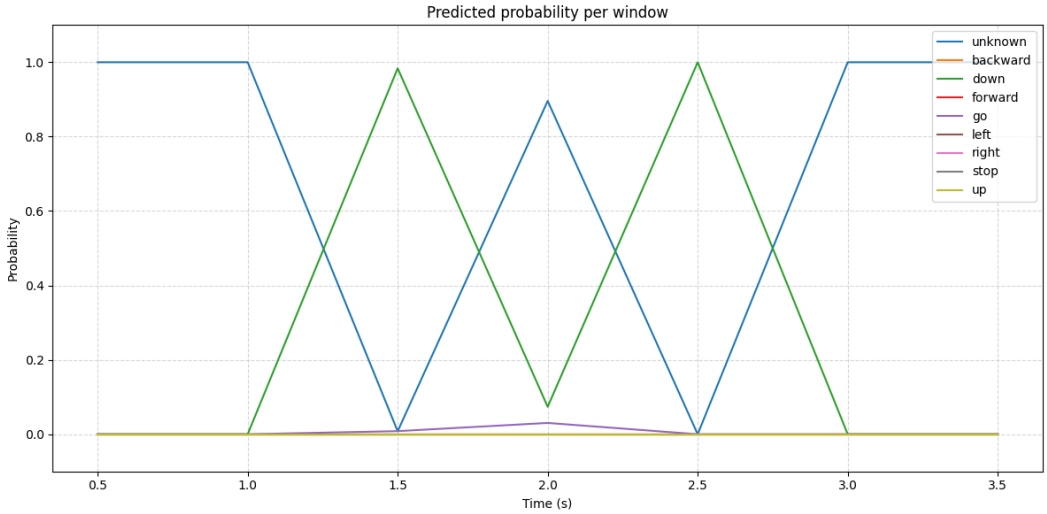

Figure 19: Line graph relating to the classification of 'multiple_same_command_down_0' audio.

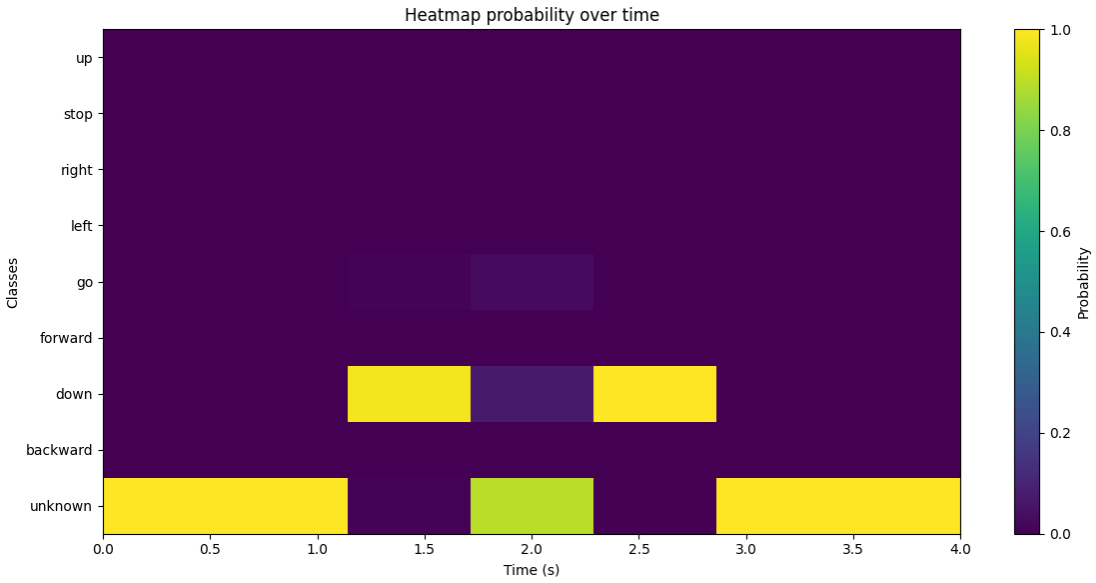

Figure 20: Heatmap relating to the classification of 'multiple_same_command_down_0' audio.

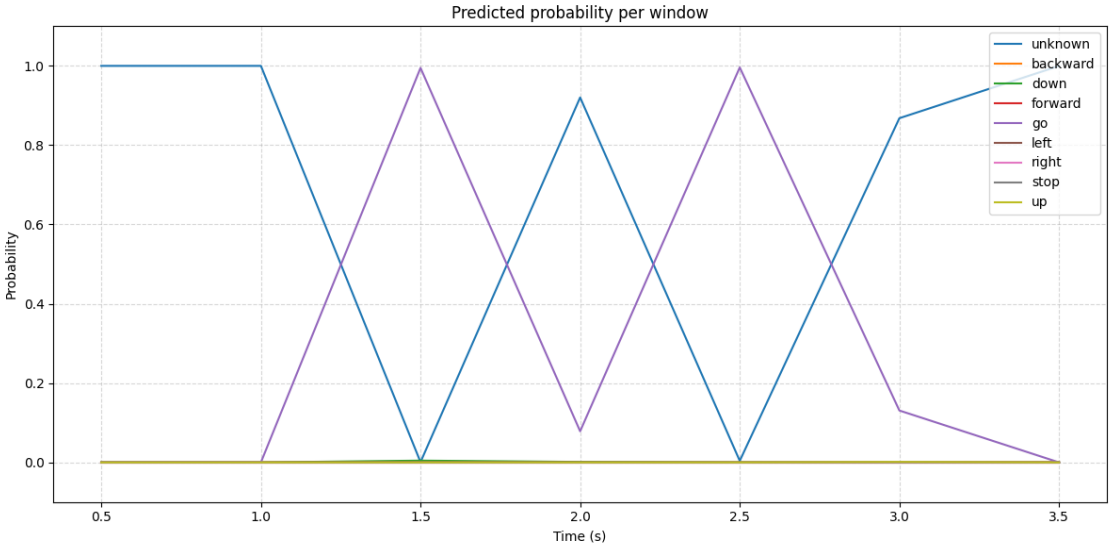

Figure 21: Line graph relating to the classification of 'multiple_same_command_go_0' audio.

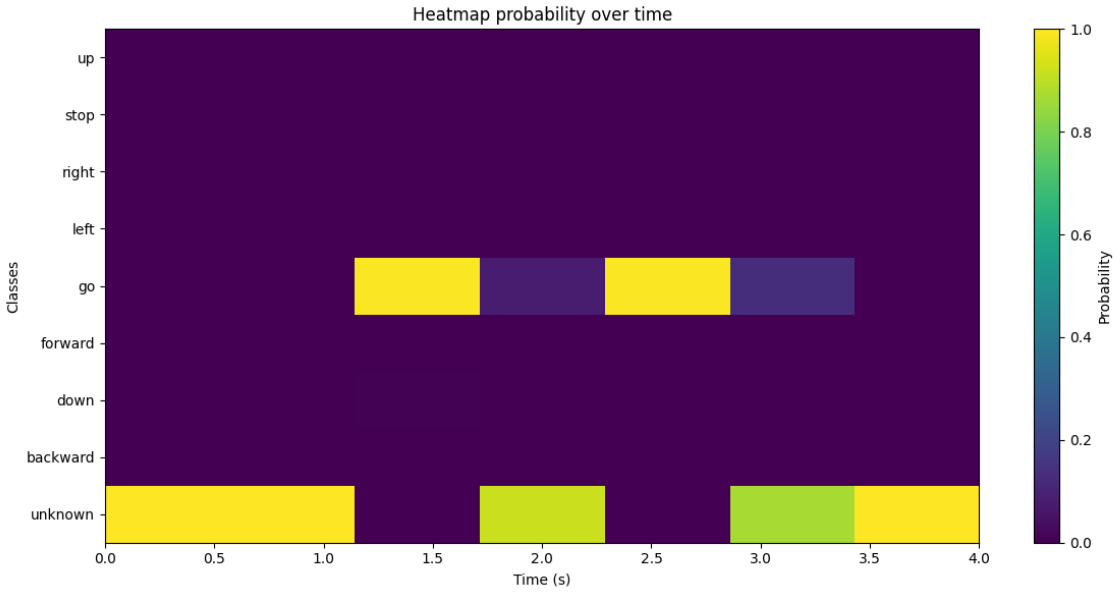

Figure 22: Heatmap relating to the classification of 'multiple_same_command_go_0' audio.

In this scenario, the two approaches exhibit identical performance, as expected, since the detections belong to the same class and the cooldown mechanism behaves equivalently in both implementations. Specifically, with a cooldown of 0 or 1, both commands are correctly recognized, whereas with a cooldown of 2 or higher, the second occurrence of the command is skipped.

The command lines outputs for these cases are shown in these images 23 24 25 26 27 28 29 30 31 32 33 34.

```
---- Start predition ----
Audio input-> Name: ../dataset/long_audio/multiple_same_command_down_0.wav , lenght: 4.00 s, windows = 7
Window 2 [1000-2000 ms] -> down (98.37)
Window 4 [2000-3000 ms] -> down (100.00)
---- End predition ----
```

Figure 23: Output relating to the classification of 'multiple_same_command_down_0' audio with the simple version of cooldown and value = 0.

```
---- Start predition ----
Audio input-> Name: ../dataset/long_audio/multiple_same_command_down_0.wav , lenght: 4.00 s, windows = 7
Window 2 [1000-2000 ms] -> down (98.37)
Skip window.
Window 4 [2000-3000 ms] -> down (100.00)
Skip window.
---- End predition ----
```

Figure 24: Output relating to the classification of 'multiple_same_command_down_0' audio with the simple version of cooldown and value = 1.

```
---- Start predition ----
Audio input-> Name: ../dataset/long_audio/multiple_same_command_down_0.wav , lenght: 4.00 s, windows = 7
Window 2 [1000-2000 ms] -> down (98.37)
Skip window.
Skip window.
---- End predition ----
```

Figure 25: Output relating to the classification of 'multiple_same_command_down_0' audio with the simple version of cooldown and value = 2.

```
---- Start predition ----
Audio input-> Name: ../dataset/long_audio/multiple_same_command_down_0.wav , lenght: 4.00 s, windows = 7
Window 2 [1000-2000 ms] -> down (98.37)
Window 4 [2000-3000 ms] -> down (100.00)
---- End predition ----
```

Figure 26: Output relating to the classification of 'multiple_same_command_down_0' audio with the more complicated version of cooldown and value = 0.

```
---- Start predition ----
Audio input-> Name: ../dataset/long_audio/multiple_same_command_down_0.wav , lenght: 4.00 s, windows = 7
Window 2 [1000-2000 ms] -> down (98.37)
Window 4 [2000-3000 ms] -> down (100.00)
---- End predition ----
```

Figure 27: Output relating to the classification of 'multiple_same_command_down_0' audio with the more complicated version of cooldown and value = 1.

```
---- Start predition ----
Audio input-> Name: ../dataset/long_audio/multiple_same_command_down_0.wav , lenght: 4.00 s, windows = 7
Window 2 [1000-2000 ms] -> down (98.37)
same command already recognized. skip window.
---- End predition ----
```

Figure 28: Output relating to the classification of 'multiple_same_command_down_0' audio with the more complicated version of cooldown and value = 2.

```
---- Start predition ----
Audio input-> Name: ../dataset/long_audio/multiple_same_command_go_0.wav , lenght: 4.00 s, windows = 7
Window 2 [1000-2000 ms] -> go (99.49)
Window 4 [2000-3000 ms] -> go (99.61)
---- End predition ----
```

Figure 29: Output relating to the classification of 'multiple_same_command_go_0' audio with the simple version of cooldown and value = 0.

```
---- Start predition ----
Audio input-> Name: ../dataset/long_audio/multiple_same_command_go_0.wav , lenght: 4.00 s, windows = 7
Window 2 [1000-2000 ms] -> go (99.49)
Skip window.
Window 4 [2000-3000 ms] -> go (99.61)
Skip window.
---- End predition ----
```

Figure 30: Output relating to the classification of 'multiple_same_command_go_0' audio with the simple version of cooldown and value = 1.

```
---- Start predition ----
Audio input-> Name: ../dataset/long_audio/multiple_same_command_go_0.wav , lenght: 4.00 s, windows = 7
Window 2 [1000-2000 ms] -> go (99.49)
Skip window.
Skip window.
---- End predition ----
```

Figure 31: Output relating to the classification of 'multiple_same_command_go_0' audio with the simple version of cooldown and value = 2.

```
---- Start predition ----
Audio input-> Name: ../dataset/long_audio/multiple_same_command_go_0.wav , lenght: 4.00 s, windows = 7
Window 2 [1000-2000 ms] -> go (99.49)
Window 4 [2000-3000 ms] -> go (99.61)
---- End predition ----
```

Figure 32: Output relating to the classification of 'multiple_same_command_go_0' audio with the more complicated version of cooldown and value = 0.

```
---- Start predition ----
Audio input-> Name: ../dataset/long_audio/multiple_same_command_go_0.wav , lenght: 4.00 s, windows = 7
Window 2 [1000-2000 ms] -> go (99.49)
Window 4 [2000-3000 ms] -> go (99.61)
---- End predition ----
```

Figure 33: Output relating to the classification of 'multiple_same_command_go_0' audio with the more complicated version of cooldown and value = 1.

```
---- Start predition ----
Audio input-> Name: ../dataset/long_audio/multiple_same_command_go_0.wav , lenght: 4.00 s, windows = 7
Window 2 [1000-2000 ms] -> go (99.49)
same command already recognized. skip window.
---- End predition ----
```

Figure 34: Output relating to the classification of 'multiple_same_command_go_0' audio with the more complicated version of cooldown and value = 2.

Audio sequences containing three different consecutive commands.

The analyzed audio tracks with different commands are:

- 'multiple_commands_0': containing the commands 'go', 'down', and 'left',
- 'multiple_commands_1': containing the commands 'right', 'down', and 'up',

Their composition is as follows:

- 'multiple_commands_0' → [unknown (0s–1s), go (1s–2s), down (2s–3s), left (3s–4s), unknown (4s–5s)] — total commands: 3
- 'multiple_commands_1' → [unknown (0s–1s), right (1s–2s), down (2s–3s), up (3s–4s), unknown (4s–5s)] — total commands: 3

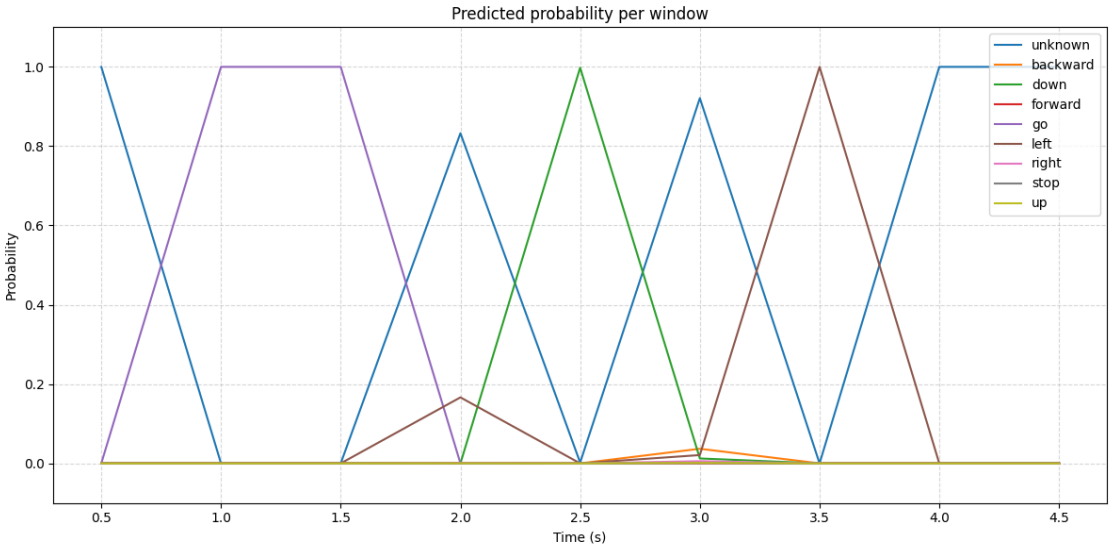

Figure 35: Line graph relating to the classification of 'multiple_commands_0' audio.

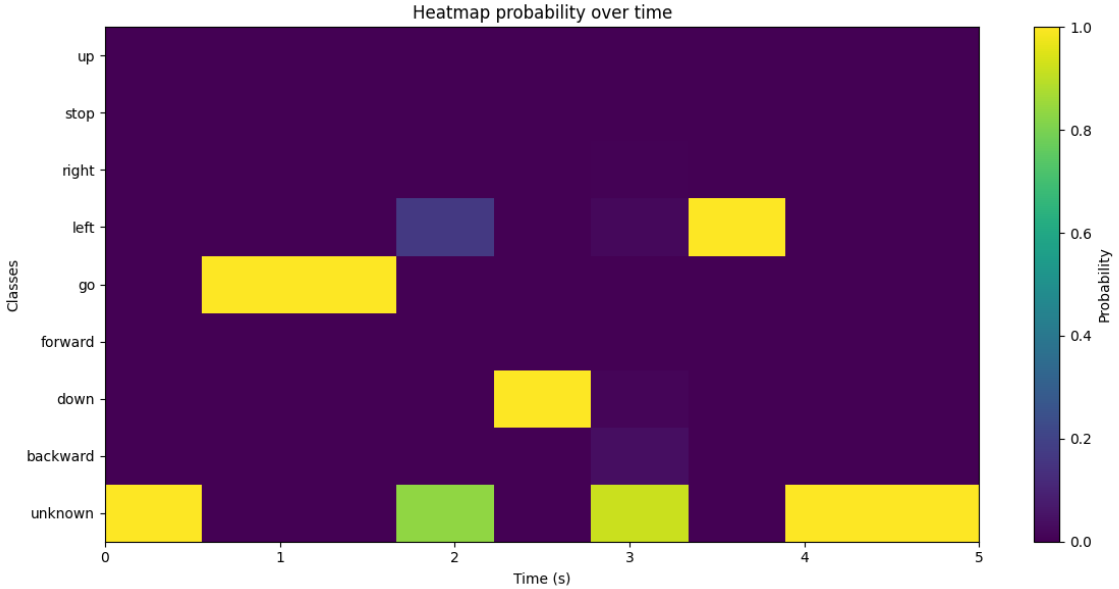

Figure 36: Heatmap relating to the classification of 'multiple_commands_0' audio.

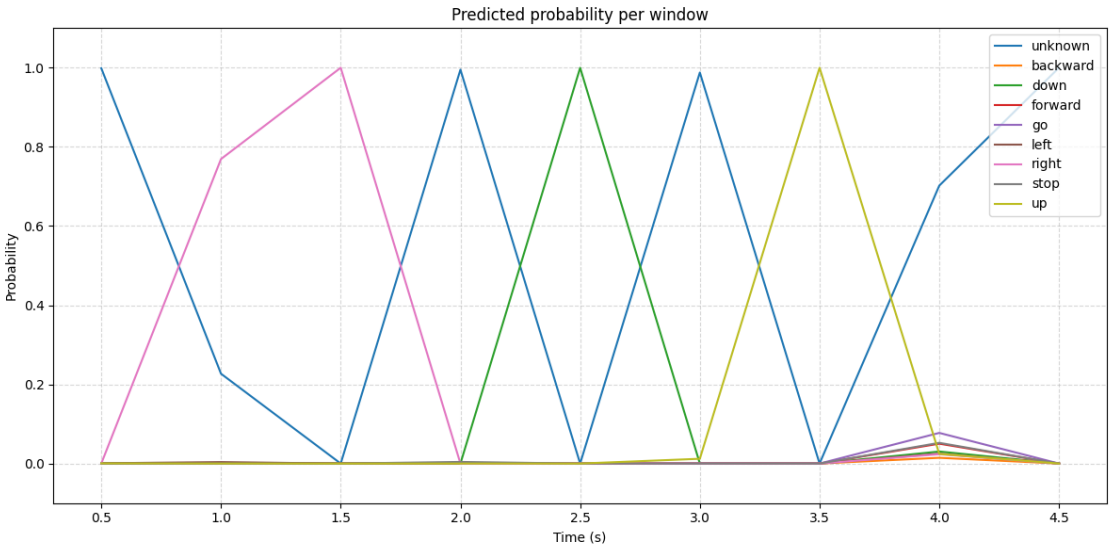

Figure 37: Line graph relating to the classification of 'multiple_commands_1' audio.

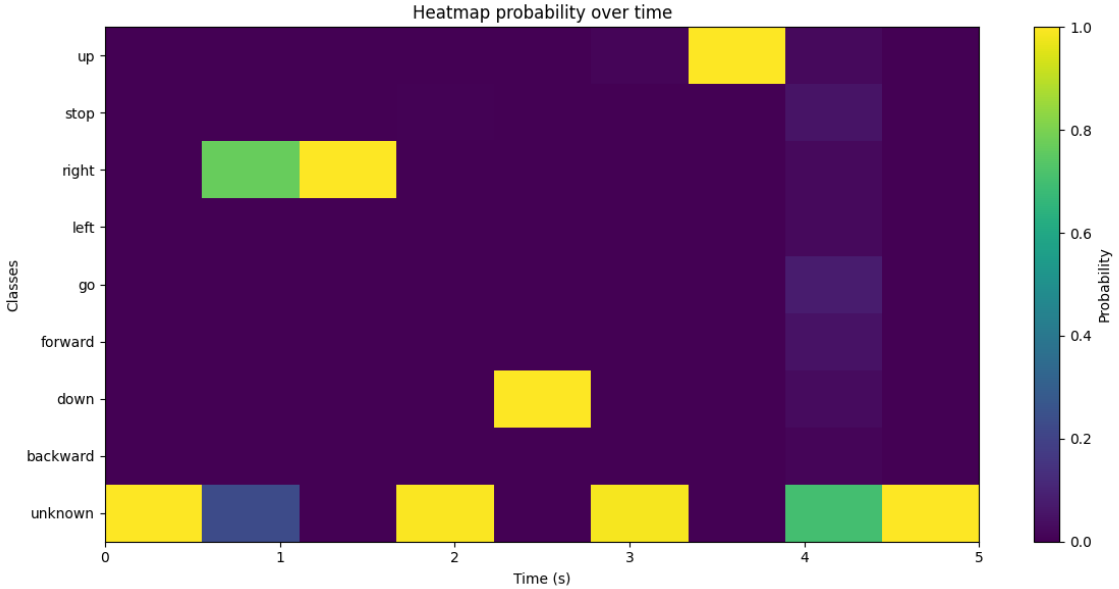

Figure 38: Heatmap relating to the classification of 'multiple_commands_1' audio.

In this scenario, both analyzed audio tracks exhibit very similar behavior. It can be observed that the performance of the two cooldown strategies is equivalent when the cooldown parameter is set to 0 or 1 (images 39 40 42 43 45 46 48 49). However, with a value of 2 (that is, two inactive windows), the simple strategy - which does not discriminate across classes — fails to recognize the final command in both tracks (images 41 47). As expected, this limitation does not occur with the advanced cooldown mechanism (images 44 50). Discrimination based on the last recognized class during the cooldown period allows subsequent commands of different classes to be correctly identified.

In the first audio track ('multiple_commands_0'), an additional phenomenon is observed, namely the duplicate detection of the first command ('go'). This undesired effect is resolved when a cooldown of 1 is applied. Interestingly, this phenomenon did not occur in the earlier experiments with repeated 'go' commands, suggesting that further investigation is required. Such an analysis would help determine whether certain commands are inherently more prone to multiple detections, and whether this is attributable to the commands themselves, to specific pronunciations, or to limitations of the used neural network.

From the current findings, it can be deduced that applying a cooldown of one inactive window is sufficient to eliminate unwanted multiple detections. Moreover, in general, the advanced cooldown mechanism manages sequences of distinct commands more effectively, even with higher cooldown values. This is because, as long as consecutive commands belong to different classes, a previously detected command does not prevent the recognition of subsequent ones.

The command lines outputs for these cases are shown in these images 39 40 41 42 43 44 45 46 47 48 49 50.

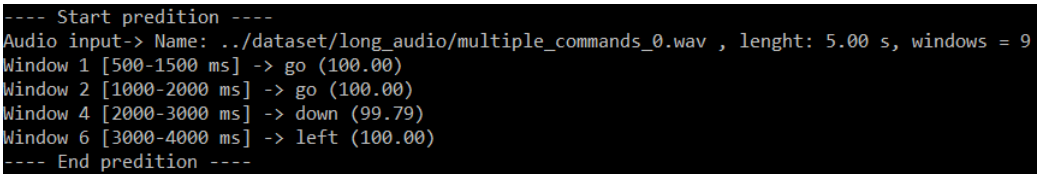

Figure 39: Output relating to the classification of 'multiple_command_0' audio with the simple version of cooldown and value = 0.

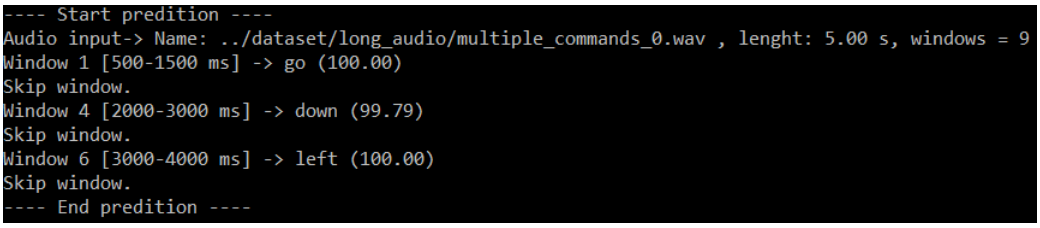

Figure 40: Output relating to the classification of 'multiple_command_0' audio with the simple version of cooldown and value = 1.

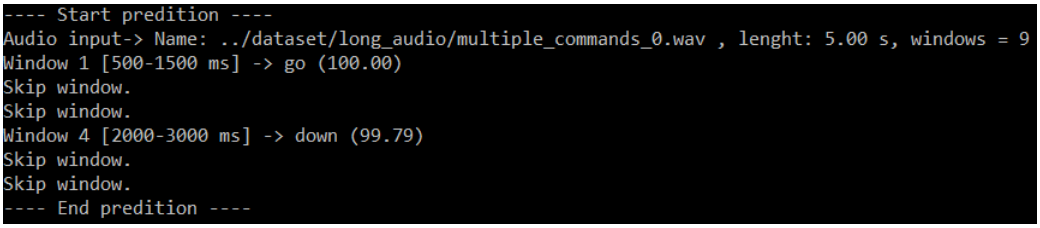

Figure 41: Output relating to the classification of 'multiple_command_0' audio with the simple version of cooldown and value = 2.

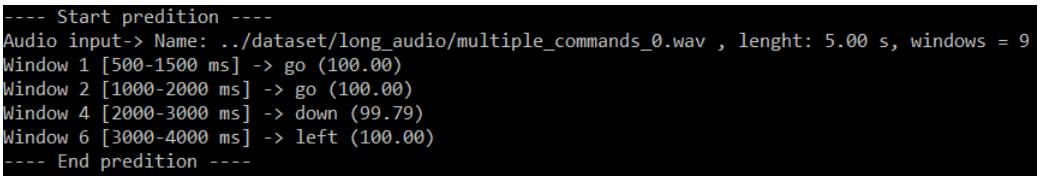

Figure 42: Output relating to the classification of 'multiple_command_0' audio with the more complicated version of cooldown and value = 0.

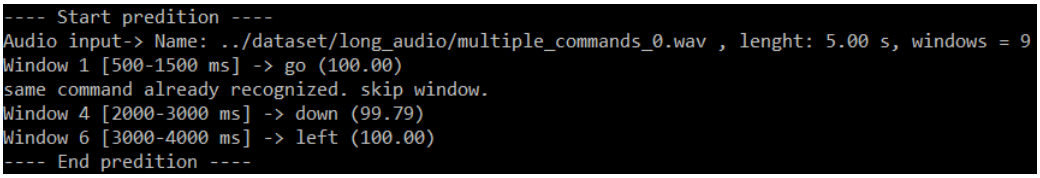

Figure 43: Output relating to the classification of 'multiple_command_0' audio with the more complicated version of cooldown and value = 1.

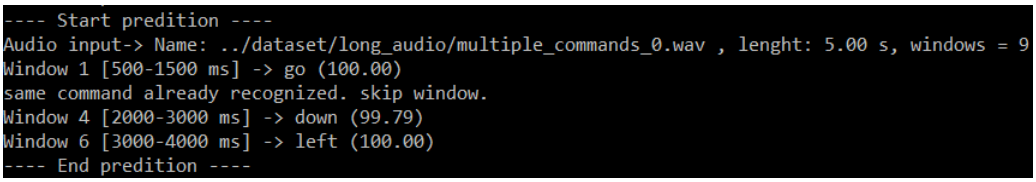

Figure 44: Output relating to the classification of 'multiple_command_0' audio with the more complicated version of cooldown and value = 2.

Figure 45: Output relating to the classification of 'multiple_command_1' audio with the simple version of cooldown and value = 0.

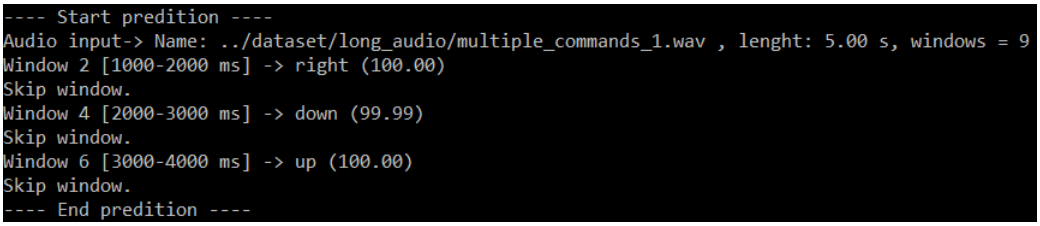

Figure 46: Output relating to the classification of 'multiple_command_1' audio with the simple version of cooldown and value = 1.

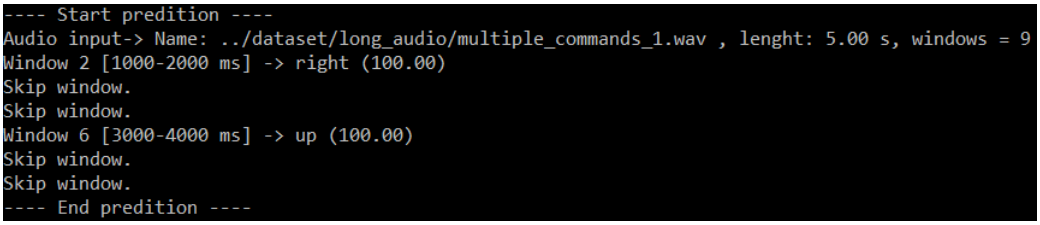

Figure 47: Output relating to the classification of 'multiple_command_1' audio with the simple version of cooldown and value = 2.

Figure 48: Output relating to the classification of 'multiple_command_1' audio with the more complicated version of cooldown and value = 0.

Figure 49: Output relating to the classification of 'multiple_command_1' audio with the more complicated version of cooldown and value = 1.

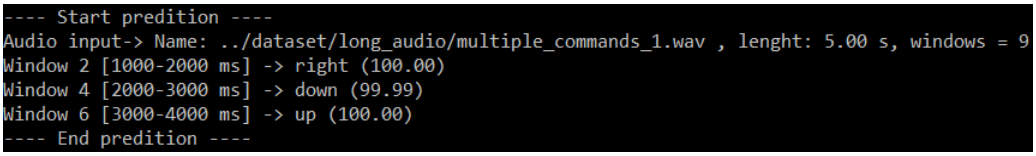

Figure 50: Output relating to the classification of 'multiple_command_1' audio with the more complicated version of cooldown and value = 2.

Given the differences between the cooldown strategies and their parameter values, as explained and demonstrated above, I decided to adopt the more advanced cooldown approach with a value of 1 (corresponding to one inactive window) for this project. This solution appeared to be the most suitable for the project's objectives. It represents an excellent choice for a command recognizer that is capable of detecting commands the correct number of times, recognizing sequences of different commands in succession, and preventing a user from issuing the same command too frequently or in rapid succession, either by mistake or under the false assumption that the previous command was not registered.

7.3.3 Evaluation on Long Empty Audio Tracks

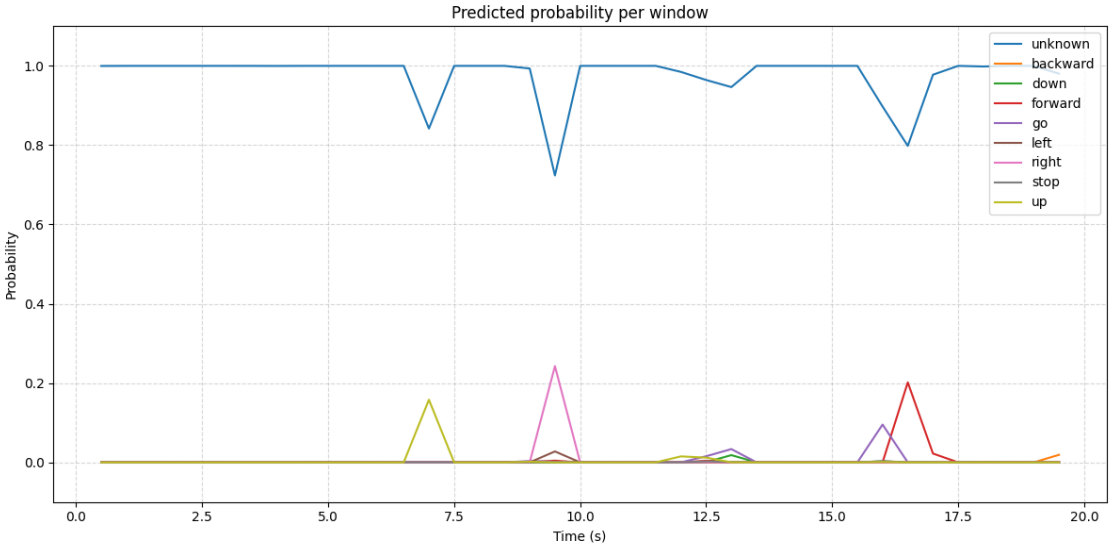

Figure 51: Line graph relating to the classification of 'long_empty_audio_0' audio.

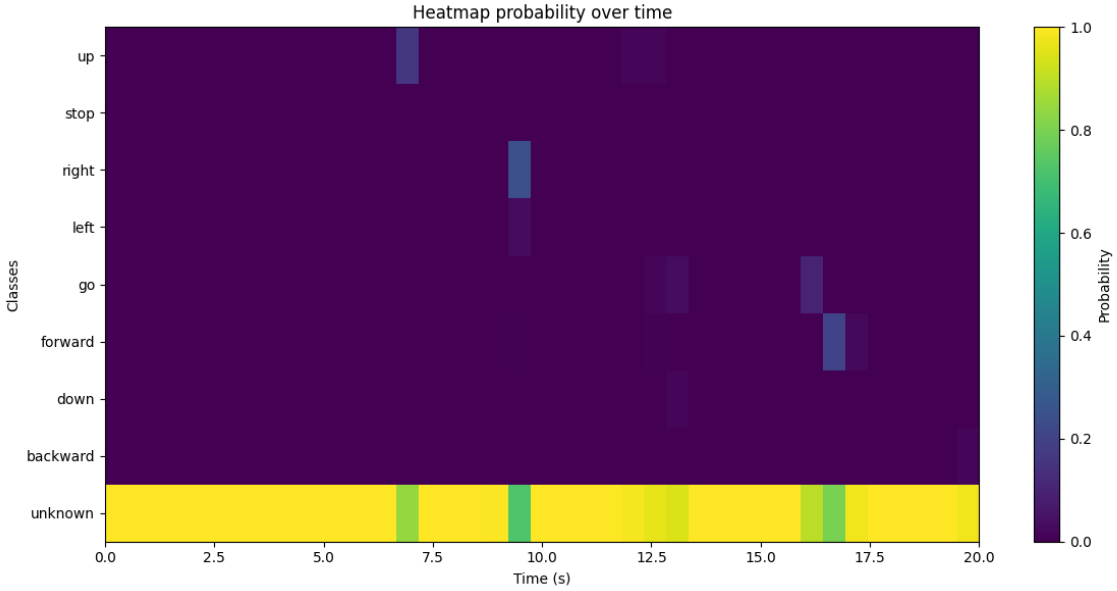

Figure 52: Heatmap relating to the classification of 'long_empty_audio_0' audio.

In this section, we analyze the network's classification performance on longer audio tracks, designed to better approximate real-world scenarios. In this case, the evaluated track (named 'long_empty_audio_0') did not contain any target commands. As shown in Figures 51 and 52, the

network performs consistently well, recognizing the “unknown” class in nearly all windows with very high confidence. Only three short segments exhibit a slight degree of uncertainty, though without producing any spurious command detections.

It is important to recall that the implemented system only recognizes commands when their confidence score exceeds 90%. By contrast, the “unknown” class is not explicitly recognized based on its own confidence level; instead, it is implicitly assigned whenever no command is detected with sufficient reliability. Had the same thresholding logic been applied to the “unknown” class, the three uncertain segments would have produced no classification output at all. To guarantee consistent and interpretable system behavior, the “unknown” handling was therefore implemented as described.

The points of uncertainty coincide with time intervals where the prediction values for multiple commands simultaneously increase, suggesting that such segments may contain acoustic features that are ambiguous or partially overlapping across different classes. Nevertheless, since no command is detected above the threshold, the system produces no output for this track, which is the expected and desired outcome.

This analysis therefore confirms the robustness of the system in realistic scenarios where no commands are present, ensuring that no false positives are generated and that the classifier remains stable even over extended audio durations.

7.3.4 Evaluation on Long Full Audio Tracks

In this scenario, a long audio track containing multiple commands to be recognized was analyzed in order to more realistically simulate a real-world use case. The track under consideration, named ’long_full_audio_0’, has a total duration of 20 seconds and is structured as follows: [unknown (0s - 1s), unknown (1s - 2s), go (2s - 3s), left (3s - 4s), unknown (4s - 5s), unknown (5s - 6s), backward (6s - 7s), unknown (7s - 8s), unknown (8s - 9s), forward (9s - 10s), up (10s - 11s), down (11s - 12s), unknown (12s - 13s), unknown (13s - 14s), right (14s - 15s), unknown (15s - 16s), down (16s - 17s), unknown (17s - 18s), stop (18s - 19s), unknown (19s - 20s)] – num of commands in the audio track: 9

This track includes at least one occurrence of every command used during training, thus providing a comprehensive and representative test case. Moreover, it contains contiguous sequences of different commands, which simulate realistic voice interaction scenarios, such as:

- the sequence ’go’-’left’ between seconds 2–4,
- the sequence ’forward’-’up’-’down’ between seconds 9–12.

In all other cases, the commands are separated by at least one ’unknown’ segment.

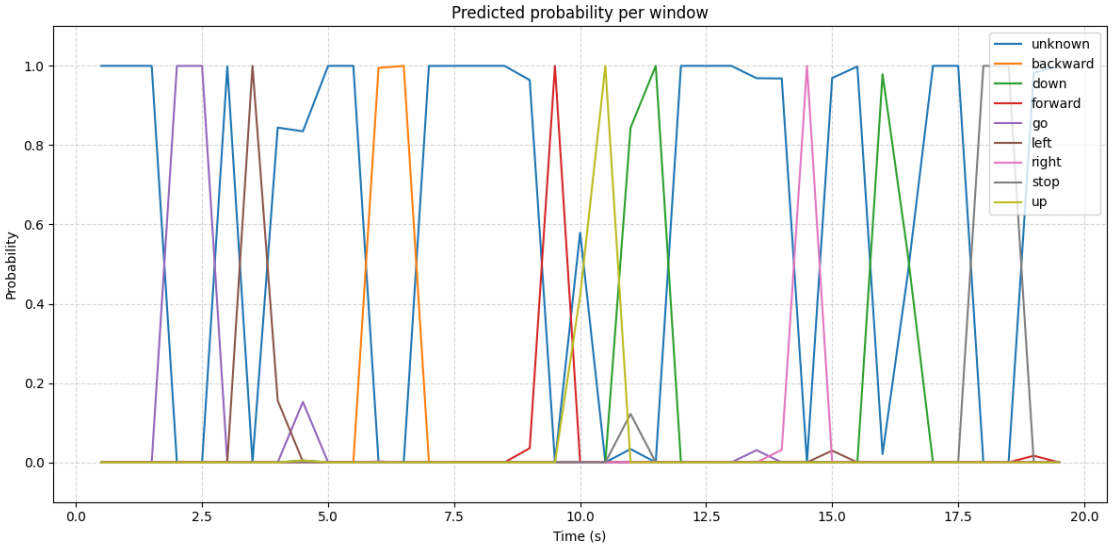

Figure 53: Line graph relating to the classification of 'long_full_audio_0' audio.

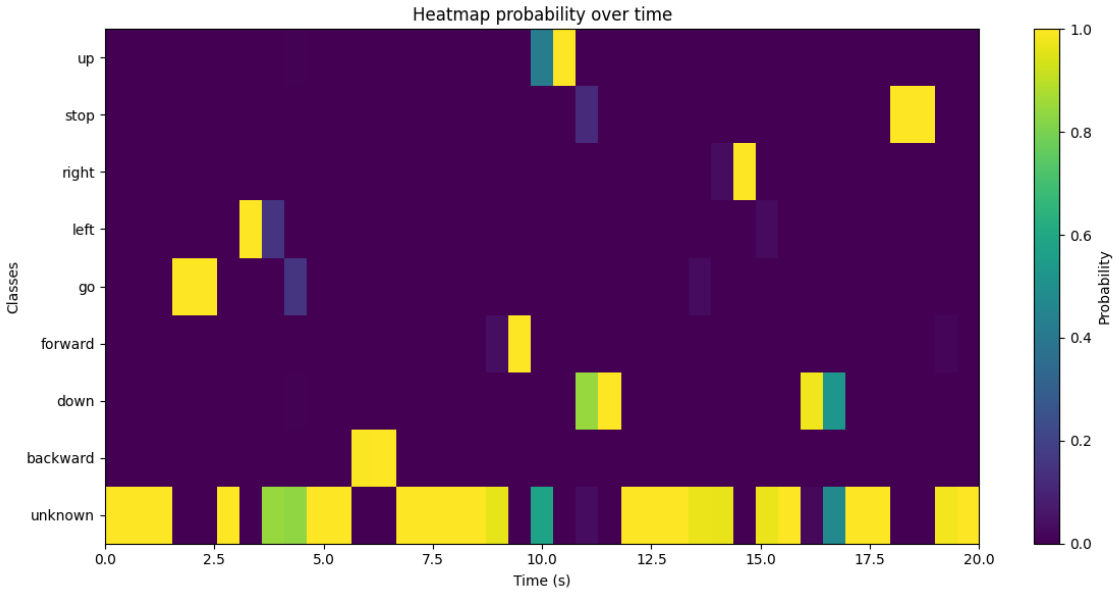

Figure 54: Heatmap relating to the classification of 'long_full_audio_0' audio.

Figures 53 and 54 illustrate the temporal evolution of the predicted probabilities for each class. It can be clearly observed that the 'unknown' class achieves high confidence only when all other classes remain close to zero. Its probability, therefore, follows a trend opposite to that of the command classes. Importantly, no situations were observed in which multiple commands were predicted simultaneously with high confidence, nor cases where both a specific command and 'unknown' were strongly activated at the same time.

In a few instances, more than one class showed nonzero probabilities, but always below 15% and never overlapping with the recognition of an actual command (where probabilities approached 100%). This behavior demonstrates the model's ability to maintain clear decision boundaries and strong discriminative power, even in the presence of complex inputs.

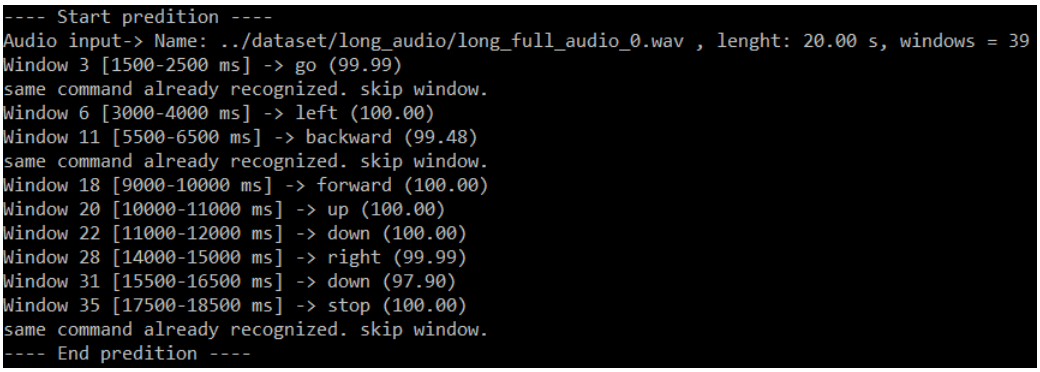

Figure 55: Output relating to the classification of 'long_full_audio_0' audio with the more complicated version of cooldown and value = 1.

The program's output for this track (shown in image 55) further validates the system's effectiveness: all 9 commands were correctly recognized, including contiguous sequences of different commands. In three cases, commands were predicted slightly earlier than their expected temporal window; however, this behavior—already observed in previous tests—does not compromise the system's reliability. In two additional cases, the cooldown mechanism successfully intercepted and prevented duplicate detections, thus proving effective in reducing false positives.

Overall, the classification of this long audio track yielded excellent results. The ability to correctly detect a sequence of heterogeneous commands, without misclassifying them as unknown and without producing spurious multiple recognitions, confirms the robustness of the model in scenarios that closely resemble real-world applications.

7.3.5 Inference Times and Computational Analysis

To evaluate the computational performance of the proposed keyword spotting system, detailed timing statistics were collected for each stage of the processing pipeline: segmentation, feature extraction, neural network inference, and post-processing (recognition and cooldown handling).

For each processing pipeline considered in this work, several statistical indicators were computed to provide a more complete and reliable characterization of system performance.
The mean value offers an estimate of the average computational cost, providing an intuitive measure of central tendency. The minimum and maximum values highlight the best and worst-case scenarios observed, respectively, which are particularly relevant in applications where latency constraints are critical.
To further refine the analysis, percentiles were also calculated, specifically the 50th (median), 95th, and 99th. The median provides a robust central estimate that is less sensitive to extreme values than the mean, while the 95th and 99th percentiles capture the behavior under high-load or rare but possible conditions.

These indicators are crucial for evaluating the stability and predictability of the system, as they reveal whether outliers significantly affect performance or if computational times remain consistently within acceptable ranges.

Together, these statistics allow for both an overall understanding of the efficiency of the system and a precise assessment of its reliability under varying circumstances.

The audio classification program collects the statistics discussed in this section and stores them in a log file named 'audio_classifier_log', located in the results directory. For all the classifications analyzed and presented so far, the corresponding log files can be found in their respective subfolders within results. Specifically, for the data considered in the following analysis, the log file is named v1_w1_info and is located at the path 'results/audio_classifier/classifier_cooldown_v1'.

Since the results obtained across the different test cases are highly consistent, this section focuses on the analysis of two representative long audio tracks: one containing no commands ('long_empty_audio_0') and one containing multiple commands distributed across its duration ('long_full_audio_0'). Both tracks have a total length of 20 seconds and generate 39 overlapping windows of 1 second each with a step of 0.5 seconds.
In the following sections, we analyze the collected statistics for the different stages of the audio classification pipeline.

Segmentation
The segmentation stage corresponds to splitting the input audio into fixed-size windows. This operation was found to be computationally negligible, with mean processing times of approximately 0.49–0.50 ms per window. Even at the 99th percentile, segmentation required less than 1.1 ms, demonstrating its minimal contribution to the overall pipeline latency.

Feature Extraction
Feature extraction (Mel-spectrogram or MFCC computation) represents a more significant computational step. In these tests, I consistently employed Mel-spectrograms and did not make use of MFCCs, even though the latter was implemented. Consequently, the values reported here should be regarded as lower compared to the case with MFCCs, since the computation of MFCCs necessarily requires the prior calculation of Mel-spectrograms. Average processing times were close to 31 ms per window, with maximum values never exceeding 34 ms. This stage, therefore, accounts for roughly one third of the total per-window processing time.

Inference
Neural network inference constitutes the most computationally expensive component of the pipeline. Across both test cases, the average per-window inference time ranged between 51.5 ms and 53 ms. The variance was limited, with the 95th percentile remaining below 56 ms and the maximum not exceeding 58 ms.

Post-processing (Recognition)
Recognition and cooldown handling required negligible time, with mean values well below 0.02 ms per window. Even in the 'long_full_audio_0' case, where multiple commands were detected, the maximum recorded value was only 0.05 ms.

Overall Per-Window and Per-Audio Statistics

Aggregating the different stages, the total per-window computational cost averaged 83–84 ms. This corresponds to 0.42% of real-time processing for each 20-second audio track. In both test cases, the complete processing of a 20-second signal required 3.2 seconds.

Discussion

These results highlight several key aspects:

- Segmentation and post-processing can be considered negligible in terms of computational cost.
- The majority of the time is spent in feature extraction and inference, which together account for more than 99% of the pipeline.
- The system exhibits highly stable inference times, with minimal differences across different input conditions (empty or command-rich tracks).
- Given the current implementation, real-time deployment is feasible even on consumer-grade hardware, as the processing requirements per window remain well below the input duration (1 second).

Overall, the computational analysis demonstrates that the proposed system not only achieves high recognition accuracy but also maintains efficiency and scalability, supporting its applicability in real-world scenarios.

8 Discussion

8.1 Strengths of the Approach

The proposed approach demonstrates several notable strengths:

- The use of overlapping windows and the cooldown mechanism allows the system to effectively mitigate the risk of multiple spurious detections, ensuring a more reliable recognition of commands in realistic scenarios.
- The adoption of Mel-spectrogram features ensures a good balance between computational efficiency and discriminative power, making the model suitable for real-time applications.
- The implementation of two cooldown strategies provides flexibility: the simpler approach favors stability in scenarios where rapid command sequences are undesirable, while the more advanced variant enables robust handling of closely spaced and heterogeneous commands.
- The system achieves high classification accuracy, with performance consistently above 98% on the evaluated dataset, confirming the effectiveness of the adopted network architecture (SirenNet v1) and training strategy.

8.2 Limitations

Despite the positive results, certain limitations are inherent to the current implementation:

- The fixed window size and stride may not optimally capture commands of varying lengths or speech rates, which could reduce recognition robustness in more diverse scenarios.
- The reliance on a fixed confidence threshold also introduces potential sensitivity to cases where background noise or atypical pronunciations affect the prediction confidence.
- The cooldown mechanism mitigates multiple detections, but it can occasionally anticipate or slightly delay the recognition of a command, which may be undesirable in time-critical applications.
- From a computational standpoint, although inference times are acceptable, the overall latency includes feature extraction overhead, which may become a bottleneck on resource-constrained devices.
- The evaluation has been limited to a predefined dataset, and further validation in more uncontrolled, real-world acoustic environments remains necessary.

8.3 Possible Improvements

Several avenues for improvement can be identified, for example:

- Introducing adaptive windowing techniques could improve robustness by dynamically adjusting window size and stride according to the temporal characteristics of the input signal.
- The confidence-based decision rule could be enhanced through calibration techniques or by employing post-processing strategies such as smoothing, which averages the predicted probabilities across neighboring windows to reduce spurious peaks, or majority voting, which assigns the final class based on the most frequently predicted label within a group of consecutive windows, thereby mitigating isolated misclassifications.

- Exploration of alternative or complementary feature extraction techniques, such as MFCCs, to evaluate their impact on both performance and computational efficiency compared to the exclusive use of Mel-spectrograms.
- An important improvement could be to optimize SirenNet 1 for efficiency while maintaining high accuracy. A network that achieves comparable accuracy but with a significantly smaller model size can offer substantial advantages in terms of inference latency and portability on embedded devices.
- Extending the evaluation with diverse speakers, noisy backgrounds, and spontaneous speech patterns would provide a more comprehensive assessment and guide further refinements of the approach.
- Another practical improvement would be to explore a broader range of hyperparameters during training, such as learning rate schedules, batch sizes, or regularization strengths. Fine-tuning these parameters can significantly affect both convergence speed and final accuracy, potentially allowing the current model to achieve higher performance without altering its architecture.

9 Conclusion and Future Work

9.1 Applications in Real-world Scenarios

KWS systems have a wide range of applications in real-world contexts where reliable voice recognition is essential, such as voice-controlled home automation or assistive technologies.
In the specific case of this project, the developed system is intended for deployment in media playback devices, such as smart TVs or MP3 players, as well as in embedded systems like FPGAs, where lightweight and efficient models are particularly advantageous.

9.2 Directions for Future Work

Based on the results obtained in this study, several promising directions for future work can be identified:

- Incorporating data augmentation techniques into the training pipeline. By artificially increasing the variability of the training dataset (e.g., through time-shifting, adding background noise, or varying pitch and speed), the model could become more robust to real-world acoustic conditions. This would improve generalization, particularly in noisy or unpredictable environments, making the system more reliable in practical deployments.

- Adoption of more advanced neural architectures, such as Transformers instead of CNNs. Transformers have been shown to outperform CNNs in various speech recognition tasks thanks to their ability to model long-range dependencies within the input signal and their higher degree of parallelization during inference. This could lead to further improvements in both accuracy and scalability.

- Deployment on dedicated hardware platforms such as FPGAs, to optimize performance and energy efficiency in real embedded scenarios.

- Experimentation on specialized microcontrollers, such as the 'Arduino Nano 33 BLE Sense', which integrates a microphone and Bluetooth connectivity and supports neural networks through Arduino ML and TensorFlow Lite for Microcontrollers. In this scenario, the on-board microphone could be used to acquire real-time data, processed directly on the device to perform KWS, while the recognized commands could be transmitted via Bluetooth to the target system in real time.