Tampere University

Tuomo Tuunanen

REAL-TIME SOUND EVENT DETECTION WITH PYTHON

Faculty of Information Technology and Communication Sciences
Master of Science Thesis
October 2020

ABSTRACT

Tuomo Tuunanen: Real-time Sound Event Detection With Python
Master of Science Thesis
Tampere University
Information Technology
October 2020

Python is a popular programming language for rapid research prototyping in various research fields, owing it to the massive repository of well-maintained 3rd party packages, built-in capabilities of the language and strong community. This work investigates the feasibility of Python for the task of performing sound event detection (SED) in real-time, which is important in demonstrating project research results to any interested parties or utilise it for practical purposes such as acoustic health care monitoring, e.g. in attempts to reduce the transmission of the COVID-19 disease.

The relevant background theory for detecting sound events based on a pre-determined sound recordings is first provided, which is followed by introduction to the basic of concepts that enable performing the same in real-time. Then, Python real-time system designs based on two related approaches are proposed and their feasibility is also evaluated with the help of corresponding reference system implementations. The results acquired with the implementations strongly suggest that Python is indeed very feasible for performing real-time SED, even when using a sophisticated model that possess 3.7M total parameters.

Keywords: real-time, aed, sed, machine listening, python

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

TIIVISTELMÄ

Tuomo Tuunanen: Äänitapahtumien Reaaliaikainen Tunnistus Pythonilla
Diplomityö
Tampereen yliopisto
Tietotekniikka
Lokakuu 2020

Tutkimusprojektien prototyyppien kehittämisessä Python on suosittu valinta ohjelmointikieleksi ja tätä tilannetta ovat edistäneet osaltaan ainakin seuraavat tekijät: laaja valikoima kolmannen osapuolen paketteja, ohjelmointikielen kattavat ominaisuudet ja laaja Python-yhteisön tuki. Tässä opinnäytetyössä selvitetään Pythonin soveltuvuutta reaaliaikaisessa äänitapahtumien tunnistuksessa, jota voidaan soveltaa tutkimustulosten havainnollistamisen lisäksi myös tärkeissä käytännön sovelluskohteissa, kuten akustisessa terveyden valvonnassa esim. hillitsemään COVID-19 -taudin leviämistä.

Työssä aluksi esitellään oleellinen taustateoria äänten luokittelemiseksi valmiiksi taltioidun äänisignaalin perusteella ja näitä apuna käyttäen esitellään Pythonilla reaaliaikaprosessoinnin mahdollistavat käsitteet. Sitten esitellään kaksi toisiinsa liittyvää Python -ratkaisuprototyyppiä, joiden perusteella toteutettuja järjestelmiä käytetään arvioimaan solveltuvuutta reaaliaikaprosessointiin. Toteutuksien avulla saatujen tulosten perusteella Python soveltuu mainiosti reaaliaikaiseen äänitapahtumien tunnistukseen jopa silloin, kun järjestelmä käyttää suhteellisen monimutkaista 3.7M parametria sisältävää neuroverkkomallia.

Avainsanat: reaaliaikainen, äänitapahtumien tunnistus

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

PREFACE

This thesis is based on the work done as a research assistant in the Audio Research Group of Tampere University Hervanta campus. I would like to thank Prof. Tuomas Virtanen for all the support, the provided opportunities to contribute to many challenging topics over the years, and for examining this thesis. Also, very grateful for the insightful thesis supervision by the researcher Toni Heittola. I would also like to give thanks to the rest of the research group, in particular the researchers Pasi Pertilä, Mikko Parviainen, Zhao Shuyang and Gaurav Naithani, for all the times we collaborated on the same projects.

I am eternally indebted to my parents for their support and encouragement through various stages of my life. Above all, I am immeasurably grateful to my fiancée Michelle, for inspiring me to become a better and more diligent person, and bringing great joy in my life every day.

Helsinki, 11th October 2020

Tuomo Tuunanen

CONTENTS

1 Introduction 1
- 1.1 Problem Statement 2
- 1.2 Content Overview 5

2 Sound Classification 6
- 2.1 Supervised Learning 6
- 2.2 Feature Extraction 8
  - 2.2.1 Frame-blocking and Windowing 9
  - 2.2.2 Mel-frequency Cepstral Coefficients 10
  - 2.2.3 Dynamic MFCC Features 13
- 2.3 Classification Methods 15
  - 2.3.1 Gaussian Mixture Models 15
  - 2.3.2 Deep Neural Networks 17

3 Real-time Processing 22
- 3.1 Audio Input 23
- 3.2 Audio and Feature Buffering 25
- 3.3 Concurrency and Parallelism 27
  - 3.3.1 Program and Process 28
  - 3.3.2 Threading and Global Interpreter Lock 28
  - 3.3.3 Process-based Parallelism 30
  - 3.3.4 Coroutines with asyncio 31

4 System Description 33
- 4.1 System Design 33
  - 4.1.1 Design Considerations 33
  - 4.1.2 Basic System Components 35
  - 4.1.3 Process-based Multiprocessing System 41
  - 4.1.4 Thread-based System 44
- 4.2 Graphical User Interface 46
  - 4.2.1 Qt Application Essentials 46
  - 4.2.2 Visualizations 46

5 Evaluation 48
- 5.1 Method 48
- 5.2 Pre-trained Model 50
- 5.3 Evaluation Scheme with YAMNet 52
- 5.4 Setup 54
- 5.5 Results 54
  - 5.5.1 Real-time System Output Validity Evaluation 54

5.5.2 Estimating Inference Speed 56
6 Conclusion 57
References 58

LIST OF FIGURES

1.1 The difference of sound events compared to the fixed-length segments outputted by the implemented system. Approximated sound events can be reconstructed using temporal information associated with the fixed-length segments. 4

2.1 Off-line classification scheme based on supervised learning 8
2.2 An audio signal $s(n)$ represented in a and analysis frames corresponding to signal $s(n)$ are illustrated in b. 11
2.3 The procedure where an arbitrary analysis frame $a(n)$ is transformed into corresponding MFCCs. The process is repeated for all analysis frames $a_j(n), j = 1, 2, \ldots$ derived from the input audio signal $s(n)$. 12
2.4 Conversion of MFCCs into Δ coefficients. The elements hilighted in red illustrate how a single MFCC at the center is interpolated with $K = 2$ surrounding coefficients into its corresponding Δ coefficient. Note that in this illustration, the displayed coefficients are enough for converting only the centermost or previous MFCC vectors $\boldsymbol{c}_{t+k}, k \leq 0$. That is, the grayed out elements represent coefficients that do not exist yet. 14
2.5 Diagram illustrating how deep learning is a type of representation learning: more abstract features are learned in layers one after another, and each layer is based on the previous simpler one. The diagram is derived from a similar one in [36, p. 10]. 18
2.6 The structure of an artificial neuron. This single *unit* of execution itself comprises a minimal neural network that estimates the output of logistic regression. The diagram representation style has been derived from a similar one in [37]. 18
2.7 Multilayer feed-forward neural network with two hidden layers. 20

3.1 Real-time classification scheme. The training stage is still identical to the off-line system, but the approach to SED has been altered in usage stage. . 23
3.2 Diagram illustrating a Python application using python-sounddevice and how it communicates with audio hardware through low-level APIs. The diagram is derived from a similar one in 24
3.3 Buffering hierarchy of the real-time SED system. Updating the frame-block triggers an extraction of MFCC feature vector and it will be pushed into the static buffer. That also triggers the ΔMFCC extraction and consequently a concatenated feature vector will be pushed into the feature buffer. 25

3.4 The concept of circular buffering is illustrated in panels a) and b). In the first iteration of panel a), data is written to slots A, B and C and the write pointer is incremented by three. In the next iteration of panel b), slots A and B are read and slots D, E and F are written; read pointer is incremented by two and write pointer by three. The slots highlighted between the pointers marks an area of active data and the white slots are free for writing. 26
3.5 Panel A illustrates the GIL-limitation of a threaded Python program. The C program in panel B possesses the same threading architecture except that it is not limited by GIL; fortunately, comparable concurrency performance in Python can be achieved with the architecture of panel C. 29
3.6 A Python example of Producer-consumer pattern with child processes in place of threads. 31

4.1 Class diagram showing the prequisite classes required for both of the real-time SED system design approaches. 36
4.2 The discontinuity of a single frame-block between adjacent fixed-length segments. In this example the overlap is 50%. 39
4.3 Sequence diagram that describes the way in which basic system components communicate with each other. 40
4.4 A free-form diagram depicting the hierarchy of data in the process-based real-time SED system approach among the main and child processes. . . . 42
4.5 A free-form diagram depicting the hierarchy of data in the thread-based real-time SED system approach among the main process and its worker thread. 44

5.1 The evaluation scheme of a real-time SED system against its counterpart offline system. Real-time usage stage is executed first and it is then followed by the offline usage stage. After obtaining prediction results from both stages, they can be validated using a pre-determined evaluation procedure. 49
5.2 The simplified evaluation scheme of a real-time SED system against its counterpart offline system demonstrating YAMNet as the sound classifier. . 52
5.3 Barplot representations of the samples of Table 5.3 55

LIST OF TABLES

5.1 YAMNet Training Parameters. 51
5.2 Evaluation System Audio Streaming Parameters. 53
5.3 Results of the real-time evaluation system output validity tests. 54

LIST OF SYMBOLS AND ABBREVIATIONS

ASR	Automatic speech recognition
CPU	Central processing unit
DCT	Discrete cosine transform
DFT	Discrete Fourier transform
DNN	Deep neural networks
GIL	Global interpreter lock
GMM	Gaussian mixture models
GPU	Graphics processing unit
GUI	Graphical user interface
LPCC	Linear prediction cepstral coefficients
MFCC	Mel-frequency cepstral coefficients
NPU	Neural processing unit
OS	Operating system
PCA	Principal component analysis
RC	Reflection coefficients
SED	Sound event detection
STFT	Short-time Fourier transform
N_{dev}	The number of development set examples
N_{test}	The number of test examples
N_{train}	The number of training examples
C	The total number of sound classes
q	The number of folds to be used when splitting the training data for cross-validation purposes
$\boldsymbol{S}_{dev}$	Development dataset containing a number of N_{dev} example pairs
$\boldsymbol{S}$	A countable set of N_{train} observations called the training data
$\boldsymbol{S}_{test}$	Test dataset containing a number of N_{test} example pairs
$\boldsymbol{S}_{train}$	Training dataset containing a number of N_{train} example pairs
$\boldsymbol{Y}$	A countable set of N_{train} output examples called the target values

$\mathbf{x}_n$	nth observation or feature vector
$(\mathbf{x}_n, y_n)$	nth input-output example pair
y_n	nth discrete-valued output example or target value
L	The total number of hidden layers
l	The layer index number spanning all layers, including the input, hidden layers and the output layer
N_l	The number of neurons belonging to the l layer
$\mathbf{a}^{[l]}$	A vector containing the activation outputs for the lth layer neurons
$\mathbf{b}^{[l]}$	A vector of bias values for the lth layer neurons
$\mathbf{w}_r^{[l]}$	A vector containing the weights of the rth neuron located at the lth layer
$\mathbf{z}^{[l]}$	A vector containing the linear combiner outputs for the lth layer neurons
K	Adjacency radius involved in the computation of delta coefficients
$c_{t,i}$	ith cepstral coefficient of tth frame
$\Delta c_{t,i}$	ith delta coefficient of tth frame
$\Delta \boldsymbol{c}_t$	tth frame index of the delta coefficients feature vector
μ	Normalisation factor involved in the computation of delta coefficients
$\boldsymbol{c}_t$	tth frame index of the cepstral coefficient feature vector
π_g	gth mixing coefficient in a mixture of Gaussians
$\mathcal{N}(\mathbf{x}\|\boldsymbol{\mu}, \boldsymbol{\Sigma})$	d-dimensional Multivariate Gaussian probability density function with feature $\mathbf{x} \in \mathbb{R}^d$, mean $\boldsymbol{\mu} \in \mathbb{R}^d$ and covariance matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{d \times d}$
G	The number of Gaussian mixture densities
$\mathbb{Z}_{\geq 0}$	The set of non-negative integers
$\mathbb{Z}_{>0}$	The set of strictly positive integers
$\mathbb{R}$	The set of real numbers
$\mathbb{R}^d$	Real vector space of dimension d
$a(n)$	An arbitrary analysis frame of length S
$s(n)$	An arbitrary input audio signal
S	Frame length in amount of samples
H	Frame hop size in amount of samples

J	The number of frames signal $s(n)$ is divided into
$w(n)$	A window function of length S
N_{block}	The number of samples in an audio input block
N_{buf}	The number of samples in a fixed-length segment buffer
$N_{channels}$	The number of audio channels used by a real-time or offline SED system implementation
$N_{classes}$	The number of classes utilised by a given model
N_{long_buf}	The number of samples held by the long audio buffer
N_{mel}	The number of mel bands in a spectrogram
$N_{patches}$	The number of patches an audio spectrogram is split into
N_{seg}	The number of samples in a fixed-length audio segment
N_{stft}	The number of stft frames in a spectrogram
z	A positive integer number multiple of the audio input stream block size N_{block}, which expresses both the number of audio blocks to collect and the number of fixed-length segments predicted by an evaluation system

1 INTRODUCTION

Considerable effort has been put into the analysis of speech signals, primarily to solve the problem of automatic speech recognition (ASR). The related field of sound event detection, which is the focus of this work, is attracting increasing amounts research interest and has benefited from the substantial amount of ASR research conducted during the past decades. *Sound event* is characterised as a text-labelled *segment* of audio corresponding to any activity or event of interest [1]. The acoustic sources producing sound events can be virtually anything, e.g. humans, animals or specific ambient noises. These events can be detected in the digital domain by utilising signal processing and machine learning algorithms in order to generalise the behaviour of related sound events. Importantly, new observed events can be categorised with the help of the resulting generalisations – that is, performing SED on previously unobserved acoustic source signals. SED is being researched in application areas such as automatic audio tagging [2], audio surveillance [3], health care monitoring [4] and identifying audio context of surroundings [5]. Furthermore, there also exists companies that have adopted SED as part of their business model, serving the needs of consumers and other businesses.

Real-time SED systems are needed for demonstrations, proof-of-concepts and practical applications that potentially benefit from live interaction with the surrounding sound sources – or in other words, with the *sound scene*. One might want to acoustically monitor occurring events of interest during some time interval or perform some actions based their presence, e.g. detecting anomalous breathing sounds or coughing could be useful contextual information in health care monitoring applications related to the ongoing COVID-19 pandemic. Another interesting possibility is to automatically tag photos and videos shared to the social media based on the detected sound events. [6, p. 4]

This work describes the requirements and procedure for implementing a SED system that is capable of processing monophonic sound events in real-time. The type of real-time processing involved is considered to be *soft real-time*, where occasionally missing a task deadline is allowed, although still undesirable [7, p. 35]. Soft real-time processing is utilised in consumer grade hardware equipped with general-purpose operating systems (OS). Those are, e.g. many popular Linux distributions, MacOS, Windows or certain embedded systems where soft real-time is still a reasonable choice over fully deterministic and time independent *hard real-time* systems. In research communities, SED systems are typically designed for obtaining and validating research results, in which case processing audio from files is already sufficient. Performing SED in real-time becomes

suitable, when system prototype or demo is designed for illustration purposes or for applicable industrial or commercial scenarios. Additionally, it could be also useful to extend the system to collect more data for future machine learning experiments.

Real-time SED system obtains input from an audio stream instead of a file. This change into *stream processing* data flow on a desktop OS leads to utilising *concurrent* execution model of program code, instead of executing instructions *sequentially* one after the other. In computing, concurrency refers to a system performing several independent tasks in parallel [8, p. 2]. Concurrent programs have their own unique challenges: code is executed simultaneously in multiple parts of the running program, and data is accessed simultaneously for reading and writing, increasing the complexity and error-proneness of a program. This work addresses these challenges using very standard approaches and describes how to apply them to design a real-time SED system. [9, Ch. 20]

The real-time SED system in the thesis project has been implemented in Python programming language and tested with regular consumer grade computer hardware that runs general purpose OS. Python is widely acknowledged as one of the standard prototyping and development tools in signal processing and machine learning alongside with MATLAB and R. In this work, choosing Python instead of e.g. MATLAB or R for more conventional software engineering scenarios, is more reasonable, because Python is general-purpose and high-level programming language, while other research programming tools may have more restrictive drawbacks or compromises that reduce the maintainability and implementability of the target system.

Alternatively, high performance low-level language, e.g. C language, could have been chosen over Python, but this choice can be counterproductive on fast-paced research projects spanning a moderately short period of time. Furthermore, studies show that developing with a high-level language contributes to productivity: they are simply more expressive and make some of the basic programming tasks straightforward in comparison to low-level languages [10, pp. 62–63]. Importantly, Python sufficiently satisfies the requirements for executing program code concurrently, although there are performance restrictions. Similar to its alternatives, many data science and machine learning research tools are available to Python, which have become very popular among researchers [11, 12, 13]. Finally, this thesis also aims to show that extending a Python-based research codebase into a system that illustrates those research results through a soft real-time data flow, can be a straightforward and productive development process.

1.1 Problem Statement

Python offers a large amount of well-supported packages and tools for scientists and engineers that enable them to conduct daily research experiments. Some experiments can turn out to be very complex and laborious to both plan and set up. Furthermore, a suitable way of conducting the experiments with Python may not be immediately clear, especially when many approaches to address individual parts of the problem can be

pursued.

This work pursues a solution to one problem specifically, namely discovering a straightforward way to design and implement a reliable real-time SED system with Python. Such a system can be launched on a computer that is attached to a microphone, and then the signal from that microphone can be processed and analysed in real-time with Python, in order to responsively detect any sound events of interest in the near vicinity of the microphone.

Implementing a real-time SED system in Python is very attractive due to the fact that the research code the real-time system is based on is also frequently implemented in Python. Any Python SED systems that allow detecting sound events based on a predetermined audio clip or a signal, which this work refers to as *off-line* SED systems, are used as a baseline for deriving the counterpart real-time SED system. Succeeding in transforming the off-line system to its real-time counterpart with Python is expected to be advantageous at least in the following practical ways:

- only Python tools and packages are used, so no need to learn additional programming languages;
- consequently, an increase in productivity is expected due to less time being spent in learning new tools or converting algorithms.

The primary objective of this work is to ascertain whether Python scales up to the task of performing real-time SED, and in the case if it does, is the behaviour of the implemented system compatible with the offline counterpart. Also, an additional target is to provide one or more suitable general designs that allow replicating or extending this work based on, e.g. other offline SED systems besides the ones involved in this work.

The Python system of this work comprises two separate stages: *training* and *classification*. In the context of this work, classification implies inferring the categories of provided audio segments with a machine learning algorithm. A machine learning system that performs classification is called a *classifier*. It is trained with datasets of annotated *training data*, which will result in a trained *model*. Then, *target class* can be predicted for all input audio segments using this trained model. Training stage is repeated primarily when the model needs to be updated, e.g. with new data or when either the model parameters or the actual classifier, is altered. Classification is performed either off-line (sequentially executing program instructions) or in real-time (concurrent execution model) and this work focuses on the latter.

Initially, classification technique called Gaussian mixture models (GMMs) was utilised in combination with mel-frequency cepstral coefficients (MFCC) for feature extraction. As the development progressed, dynamic delta features were also extracted from MFCCs, improving audio class separability. Later, after GMMs were established as a standard baseline for the system, decision to use more advanced classification technique was made: off-line and real-time stages of the system were modified by replacing GMMs with deep neural network (DNN) to further improved classification accuracy. At the end of the

project, a publicly available DNN model was also utilised in evaluation of the real-time system designs based on the aforementioned GMM and DNN experiments.

An Important Simplification

There is an important simplification related to the way both the off-line and real-time systems perform sound event detection in this work. SED systems are assumed to produce subsequent fixed-length segments based on the input of a single-channel audio signal. However, the fixed-length segments do not yet constitute as detected sound events, because those segments are not guaranteed to individually match the *onset* and *offset* times of intended sound events [6, pp. 13–16]. Rather, the fixed-length segments can be used to reconstruct full output sound events as shown in Figure 1.1.

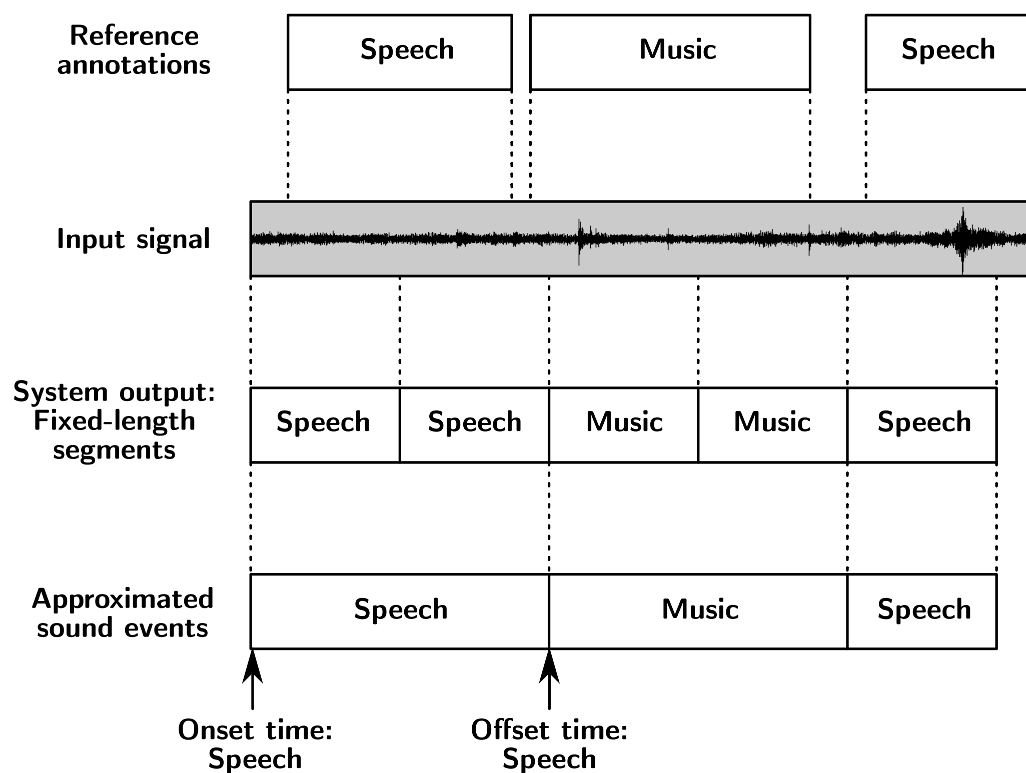


Figure 1.1. *The difference of sound events compared to the fixed-length segments outputted by the implemented system. Approximated sound events can be reconstructed using temporal information associated with the fixed-length segments.*

Onset and offset times of each output sound event are approximated based on appropriate fixed-length segment boundaries. As a result, the outputted sound events have a time resolution determined by the fixed segment length that can be adjusted to suit the desired level of accuracy. It should be noted that an algorithm for reconstructing sound events based on fixed-length segments, which the implemented SED system has outputted, is not part of this work. Implementing such an algorithm is left as future work instead.

1.2 Content Overview

The work first covers the theory for designing a sound event detection system with sequential flow (off-line) in Chapter 2. Then, Chapter 3 builds on top of the previous one, describing the requirements for real-time system (on-line). The real-time counterpart includes concepts not required in the off-line scenario, such as audio input processing, buffering and executing tasks in parallel. The approach changes from theoretical to practical in Chapter 4, where Python-specific design and implementation details and solutions are described. In Chapter 5, methods for evaluating the real-time system performance are explained and benchmarked results are also reviewed. Chapter 6 concludes the thesis and introduces possible future work.

2 SOUND CLASSIFICATION

In sound classification various machine learning techniques are applied on an audio signal processing problem. For given audio segments, learning algorithm is used to predict their categories from a finite set of related textual labels. In machine learning, these distinct categories are commonly referred as *classes*. This chapter introduces the techniques relevant to performing sound classification, and then Chapter 3 further extends those concepts to real-time audio processing, providing the necessary background that allows implementing a practical SED system with Python.

A learning algorithm infers the best fit for an acoustic model by optimization. For that purpose, the majority of practical machine learning systems rely on labelled data, which in the case of this work is the audio segments. This makes the sound classification approach to be considered *supervised learning*.

Ultimately, it is hoped that learning algorithms can accurately fit a model from a large set of unlabelled data with an *unsupervised learning* method, using complex reasoning capabilities. Although this self-organised learning approach is expected to become important in the long term, it is still far away from producing the best results. Moreover, combining supervised learning with other techniques has also produced good results, and it might become important in the future. [14]

When reading this chapter, a very important distinction to keep in mind is the difference between sound classification and *detection* of sound events. A sound classifier does categorise textual labels within a given input signal or audio recording, but it does not reveal the temporal position of those occurred labels. That is where detection comes in, as already illustrated by the reference annotations and approximated sound events in Figure 1.1. Sound classification is rather one of the required components that allow defining an occurred sound event. [6, p. 5]

2.1 Supervised Learning

In supervised learning, also known as *learning with a teacher*, a learning algorithm utilises a large collection of observed input-output examples in order to readjust the parameters of a model to match desired responses, based on the given examples [15, p. 63]. The collection of all input examples is defined as a countable set of N_{train} observations known as *training data*

$$\boldsymbol{S} = \{\mathbf{x}_n \mid n = 1, \ldots, N_{train}\}, \tag{2.1}$$

where the individual real-valued observations $\mathbf{x}_n$, also known as *features*, are denoted by vector

$$\mathbf{x}_n = \begin{bmatrix} x_{n,1} \\ \vdots \\ x_{n,d} \end{bmatrix} \in \mathbb{R}^d. \quad (2.2)$$

When training a sound classifier, a set of feature vectors $\mathbf{x}_n$ are extracted from each audio segment, where the segments correspond to a particular sound event. The given audio segments are divided into statistically stationary frames and feature vectors are extracted from each frame. These frames are then transformed into feature space $\mathbb{R}^d$ by *hand-crafting* or with *feature learning* [15, 16]. This work encompasses hand-crafted features and any abstractions learned with DNNs from them. Since the examples are input-output pairs, the training data $\mathbf{S}$ is combined together with a corresponding set of output examples known as *target values*

$$\boldsymbol{Y} = \{y_n \mid n = 1, \ldots, N_{train}\}. \quad (2.3)$$

The elements y_n of equation 2.3 are discrete values representing the true category of corresponding features $\mathbf{x}_n$. That is, equations 2.1 -2.3 are related such that the training data can be rewritten as a set of ordered pairs

$$\boldsymbol{S}_{train} = \{(\mathbf{x}_n, y_n) \mid n = 1, \ldots, N_{train}\}, \quad (2.4)$$

which is called *training dataset*.

The parameters of a model are tuned with training examples $(\mathbf{x}_n, y_n)$ in order to reduce sum of costs [16] or maximise a likelihood function [17], depending on the method. Then, with the trained model, classifier categorises unforeseen examples $(\widehat{\mathbf{x}}_n, \widehat{y}_n)$ belonging to a separate *test dataset*

$$\boldsymbol{S}_{test} = \{(\widehat{\mathbf{x}}_n, \widehat{y}_n) \mid n = 1, \ldots, N_{test}\}. \quad (2.5)$$

It is also common practice to separate or *hold out* a small portion of the training dataset $\boldsymbol{S}_{train}$ for a *development dataset* $\boldsymbol{S}_{dev}$ containing N_{dev} examples – the dataset $\boldsymbol{S}_{dev}$ is used for selecting the best-performing model. Note that the distributions these datasets need to match, so $\boldsymbol{S}_{train}$ is randomised before splitting in a process of *stratification*. One more common practice involves *cross-validating* the training data by resampling it q times. The idea is that in each q iterations, different portions of training and development dataset are drawn from individual *folds*.

The above-described supervised learning process is illustrated in Figure 2.1. The two stages, training and usage, are separate. Training stage is performed first in order to acquire models that can be used for classification of sounds. In usage stage, an acoustic

signal acquired from an audio file is processed into segment of features, and classifier uses those features to produce sound category labels as prediction results for each segment.

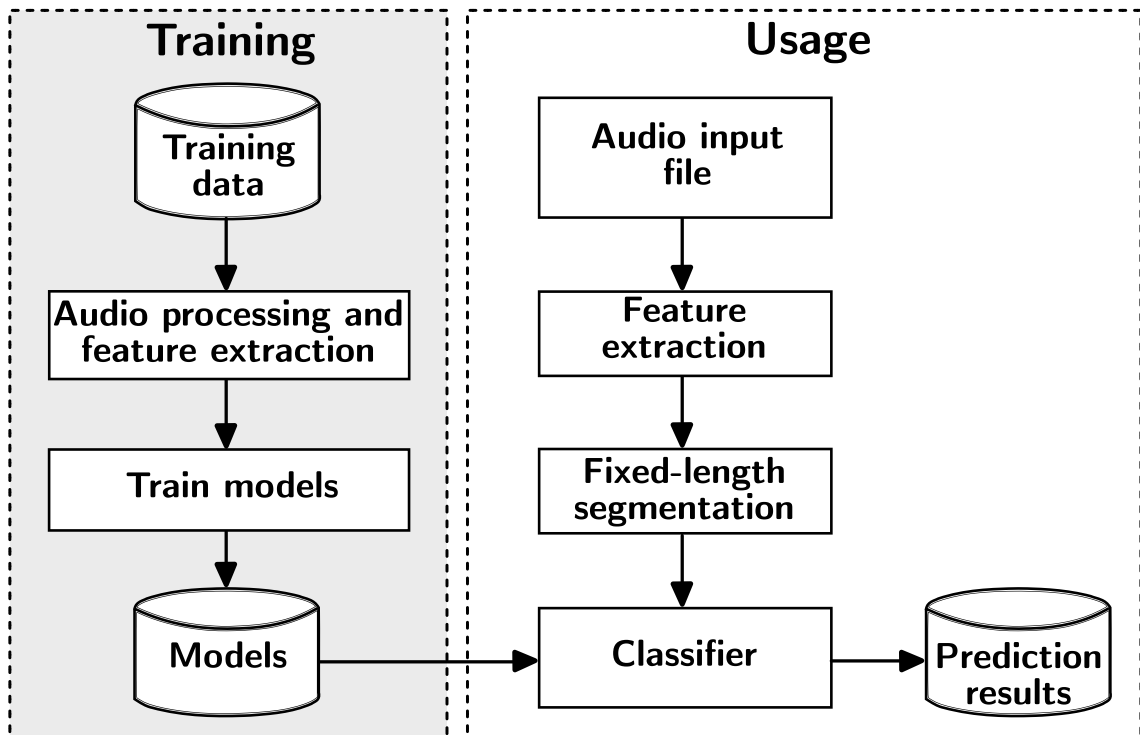


Figure 2.1. *Off-line classification scheme based on supervised learning*

Performance can be evaluated with the prediction results. The training stage is repeated if the models did not generalise well, e.g. due to *over-fitting* by poor *model selection* — or if some potential model training approach was experimented with, consequently having to modify and retrain the model.

2.2 Feature Extraction

One of the most important goals of feature extraction is improving the separability between different categories of input data in a machine learning problem. The original input variables are transformed into a new variable space called *feature space*, where the classes of interest are expected to become more separable than in the original input space [17, p. 2]. The motivation for this expectation is found in *Cover's theorem* [18], which in qualitative terms states that non-linearly transformed features cast into a high-dimensional space are more prone of becoming linearly separable than in the original low-dimensional space [15, Sec. 5.2]. Techniques of *feature transformation* can further improve the class separability of high-dimensional features; the data is mapped into a space that discards redundancies, making the new representation more effective and dimensionally reduced [19, p. 424]. As a fortunate consequence, computational complexity is also decreased along with reduced dimensionality, which is desirable due to the time-sensitive nature of real-time classification systems.

Section 2.1 described that a set of feature vectors is extracted for each audio segment. In order to simplify classification, the length of these segments is fixed to one second. The reasoning is the following: classifier will be able to identify prominent characteristics that sufficiently describe the given sound event even though some inaccuracies exist when segment boundaries mismatch with the real ones. The fixed segment length also has the consequence that an equal amount of feature vectors is always extracted from each audio segment.

This work utilises MFCCs in feature extraction, which were also extended with temporal cepstral derivatives. First, the procedure known as *frame-blocking* divides given audio segment into short *windowed* frames known as *analysis frames*. Then, a fixed quantity of MFCCs are extracted from each analysis frame, discarding the zeroth mean (or DC) coefficient as redundant. Furthermore, the time derivatives extracted from MFCCs provide information about temporal changes in the features. Detailed explanations and motivations for these subsequent steps are given in the two following subsections.

2.2.1 Frame-blocking and Windowing

The goal of frame-blocking and windowing is to ensure that sufficiently accurate spectral estimate is obtained from statistically stationary segments of an audio signal. Suppose that there exists an arbitrary audio signal $s(n)$ as in Figure 2.2a. Frame-blocking procedure divides signal $s(n)$ into frames of S samples, where S is also known as the quantity called *frame length*. Adjacent frames are separated from each other by H samples and this divergence or shift between frames is known as *hop size*. When $H \leq S$, the frames overlap and correlate with each other, depending on the size of hop H. Respectively, frames do not overlap when $H > S$ and results in increasing amount of data loss, as the size of H is increased while S remains constant. [20, pp. 113–114]

After obtaining frame-blocks, estimating the short-time spectra would be already possible with *Discrete Fourier Transform* (DFT) from each frame-block of signal $s(n)$. However, DFT interprets the finite frame-blocks to be periodic, although in reality they are not [21, p. 688]: discontinuities in frame-block borders cause undesired spectral overshoot and undershoot called Gibbs-phenomenon [22, p. 288]. Fortunately, tapering the samples of frame-blocks towards zero near the bordering discontinuities is efficient in minimising the magnitude of Gibbs-phenomenon [20, p. 114]. Tapering with a window function may cause data loss, so overlapping the adjacent frame-blocks ensures that most information originating from signal $s(n)$ is conveyed to the resulting analysis frames after windowing, which is generally true with overlap between 50 and 75%. [21, p. 691].

Besides minimising the Gibbs-phenomenon, there is another reason for applying a window function. Without separately applied window function, frame-block is still considered to be rectangularly windowed, where samples of the rectangle are tapered zero outside the frame boundaries. Rectangular window shape is problematic, because for each frequency component of a signal frame-block, spurious peaks are introduced. These peaks

may mask the spectrum with side lobes or could end up concealing the true ones. The phenomenon is known as *spectral leakage* and is countered by choosing a window, which reduces the size of side lobes. However, all window choices are compromises, because reducing side lobe effect increases the main lobe width. In audio processing, the *Hamming* or *Hanning* windows are typical choices for minimising the effects of both Gibbs-phenomenon and spectral leakage, while maintaining a good compromise between side lobe heights and main lobe widths. The frame-blocking scheme manually implemented in this work utilises Hamming window with 50% overlap between frames, hop size of 20 ms and frame length of 40 ms. This choice for overlap is a very suitable, because the sum of overlapping regions of Hamming and Hanning windows adds up to one. Also, overlap of 40% is also experimented with in Chapter 5 due to the requirements of a pre-trained model. [21, p. 689-690] [19, p. 232]

The concept of analysis frames is illustrated in Figure 2.2b, which represents four frame-blocks being tapered with the Hamming window function into the analysis frames. Analysis frame and the window can be defined as follows. Suppose that there exists J frame-blocks of $s(n)$, then an individual jth block is denoted by

$$x_j(n) = s(Hj + n), \tag{2.6}$$

where

$$0 \leq n \leq S - 1 \tag{2.7}$$

and

$$0 \leq j \leq J - 1. \tag{2.8}$$

Multiplying the samples of $x_j(n)$ with the ones of window function $w(n)$ results in the jth analysis frame

$$a_j(n) = x_j(n)w(n), \tag{2.9}$$

where

$$w(n) = 0.54 - 0.46 \times cos(\frac{2\pi n}{S - 1}) \tag{2.10}$$

and

$$0 \leq n \leq S - 1. \tag{2.11}$$

Equation 2.10 is known as the Hamming window function, which is commonly utilised in audio processing.

2.2.2 Mel-frequency Cepstral Coefficients

MFCCs were originally developed for ASR [23] and have become very widely utilised features since their introduction; the previously popular approaches based on Linear Pre-

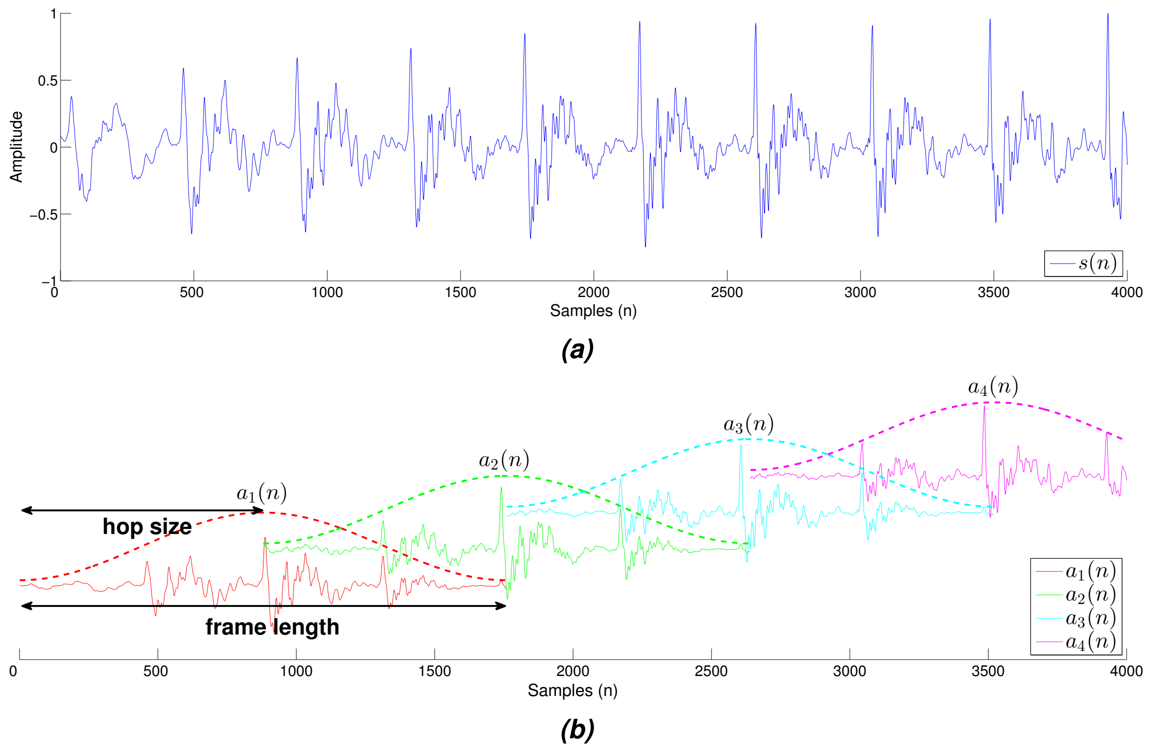


Figure 2.2. *An audio signal* $s(n)$ *represented in a and analysis frames corresponding to signal* $s(n)$ *are illustrated in b.*

diction Cepstral Coefficients (LPCC) and Reflection Coefficients (RC) proved to be more vulnerable to noisy speech [24] and spectrally less accurate [23] compared to filter bank based MFCCs. In general, SED systems should strive to utilise features that discriminate well between different types of sounds and MFCC-based features are typically a well-performing choice [24, 25, 26]. Many processing steps in MFCCs are influenced by the observed and assumed behavior of the human auditory system and these assumptions will be introduced along with the formal representation next.

The thesis project employed MFCC extraction pipeline illustrated in Figure 2.3, which converts a single analysis frame into MFCCs. The process gets repeated until MFCCs have been extracted from all analysis frames corresponding to input signal $s(n)$. In speech processing, a pre-processing step known as the *pre-emphasis* is frequently applied on the input signal. The step spectrally emphasises high frequencies, which is favorable in speech recognition [20, p. 113], but it might decrease the discrimination of features for sounds other than speech. Therefore, this work omits pre-emphasis and obtaining the DFT *power spectrum* of an analysis frame is the first step in extracting MFCCs, as shown in Figure 2.3. The major reason MFCCs utilise DFT in particular is that the human auditory system performs a similar spectral analysis for sounds. [27, p. 250][28, p. 194].

The DFT power spectrum is next modified to further model the known behaviour of the auditory system: multiple narrowband sounds with closely related frequencies of similar levels are interpreted being a single auditory event. That is, for an arbitrarily selected frequency bin in the power spectrum, the closely neighbouring bins become less impor-

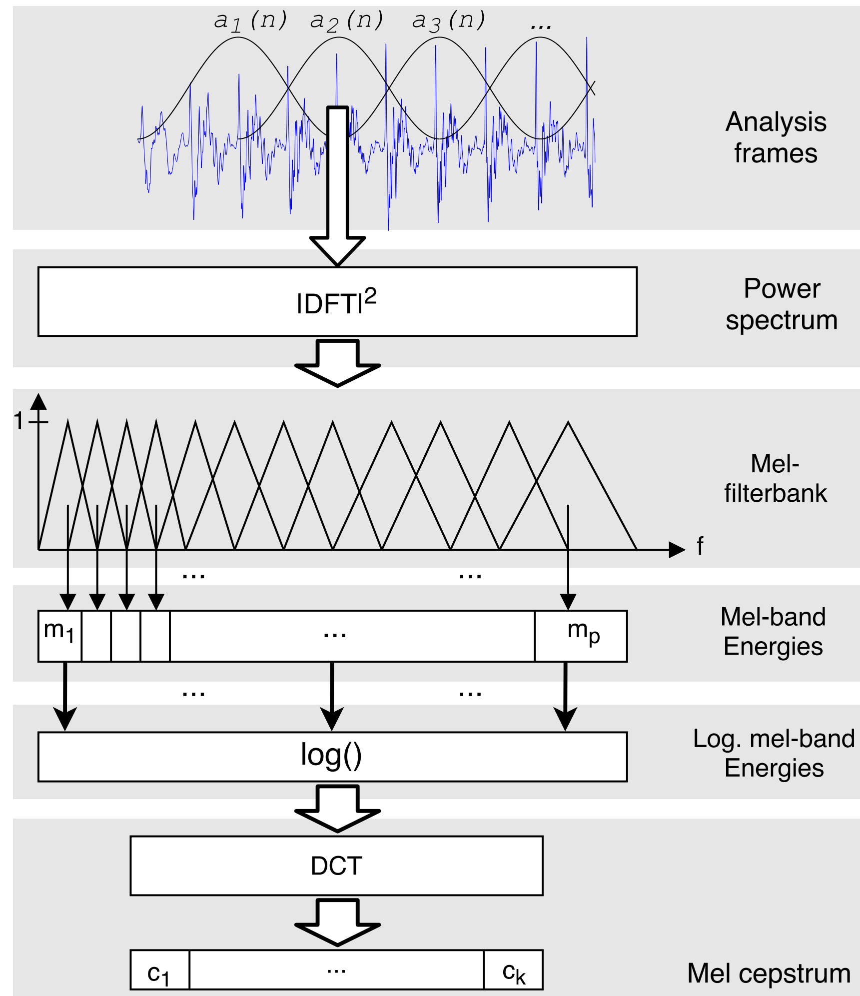


Figure 2.3. *The procedure where an arbitrary analysis frame $a(n)$ is transformed into corresponding MFCCs. The process is repeated for all analysis frames $a_j(n), j = 1, 2, \ldots$ derived from the input audio signal $s(n)$.*

tant compared to the referencing center bin [27, pp. 163–164]. In practice, a triangular window function, which in this context is often referred as filter, is a good choice for de-emphasising the importance of bins surrounding a selected reference frequency. The triangularly weighted frequency bins are not needed individually, so they will be summed to correspond the energy of a frequency band in the power spectrum. Now, the whole frequency range needs to be converted into band energies with a group of triangular filters; this requires defining the band width and center frequency (the peak of a triangle) for each filter. Humans have been observed to perceive sound pitch roughly in the non-linear *mel-scale* instead of the measured linear one [27, pp. 174–175]. Therefore, the center frequencies of the triangular filters are first defined uniformly in mel-scale, which results in mel-spaced center frequencies after inverting back to linear frequency scale. This work utilised alternative equations in mel-scale frequency warping, compared to the ones common in literature [27, 28, 29]. However, the warping is almost identical in both

approaches. Accordingly, conversion from linear to mel-frequency scale is defined as

$$M(f) = 1125\ ln(1 + \frac{f}{700}) \tag{2.12}$$

and conversely equation 2.12 can be inverted back to the linear frequency-scale with

$$M^{-1}(m) = 700\ \exp(\frac{m}{1125}) - 1. \tag{2.13}$$

Defining the band width of each mel-scale triangular filter is straighforward: a band starts from the center frequency of the previous one and stops at the next center frequency, resulting in overlapped filters as illustrated the *mel-scale filter bank* of Figure 2.3 — exceptions to this are the stop and start points of filters adjacent to frequency scale boundaries. The band energies obtained with the mel-scale filter bank from the power spectrum are referred as *mel-band energies* and the auditory model can still be enhanced even from this state: besides the pitch of a sound, loudness also behaves non-linearly [28, p. 195] and this is modelled in MFCCs simply by computing the natural logarithm for each mel-band energy coefficient.

The final processing in MFCC pipeline addresses the high correlation between adjacent *logarithmic mel-band energy* coefficients. These correlations exists, because the mel-scale triangular filters overlap with each other. In signal processing, the Discrete Cosine Transform (DCT), can be used for decorrelation and also to discard insignificant transform components, which is very comparable with the well-known Principal Component Analysis (PCA) transform [21, p. 136]. The logarithmic mel-band energies are transformed with the DCT and only the lowest order transform components corresponding to significant changes are kept; the higher order DCT components correspond to quicker changes in the input, making them irrelevant in representing the most descriptive characteristics of the log-energies. Typically, the DCT is computed from log-energies correspondinging to an amount of 24 to 40 triangular filters and the first 13 or 14 DCT coefficients are used to form the mel cepstrum of an analysis frame [19, p. 315]. This work uses the well-known 13th-order MFCC feature that is common to many ASR and SED applications. It is formed using the first 13 cepstrum coefficients acquired with 40 triangular mel filter from log-energy outputs.

2.2.3 Dynamic MFCC Features

Measuring the change in cepstral coefficient over time is a common method for improving their feature separability and it is achieved by computing their time derivatives [19, p. 424]. These type of temporal audio features are known as *dynamic features*, while MFCCs are being called *static* features. In this work, the MFCC derivatives are approximated with an orthogonal polynomial fit over a window of length $2K + 1$, where K represents an adjacency radius for frames surrounding the middlemost one in that window [20, p. 117]. Figure 2.4 illustrates the mechanism of this radial window. The approximated 1st-order

derivatives are sufficient for achieving most of the benefits in improving the baseline of static MFCC features — in general, derivatives of 3rd-order or greater cease to provide any meaningful improvements in terms of class separability [19, p. 423-424].

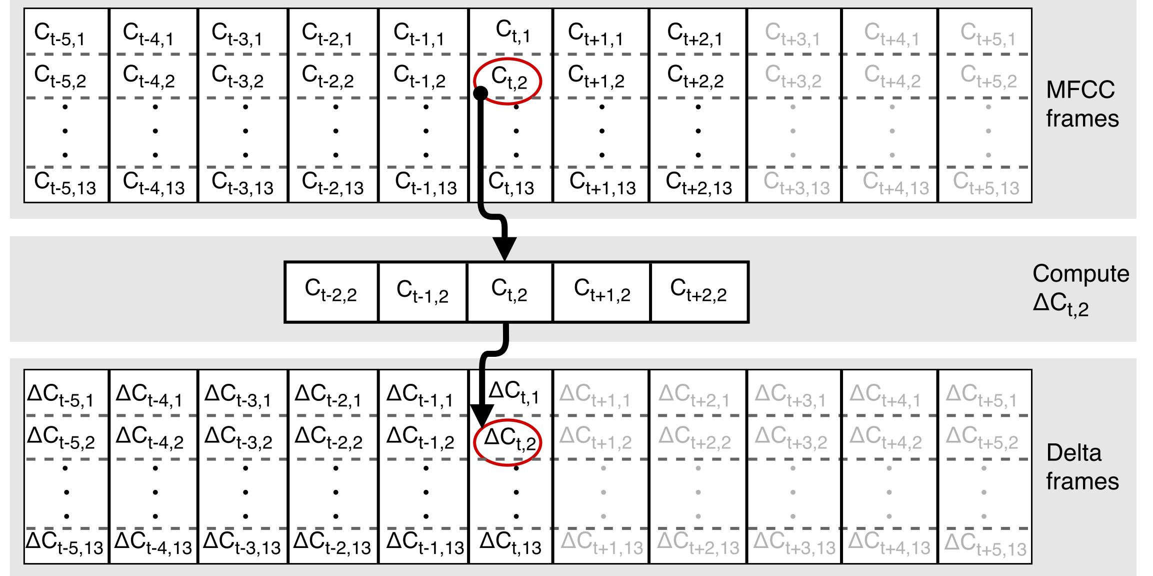


Figure 2.4. *Conversion of MFCCs into Δ coefficients. The elements hilighted in red illustrate how a single MFCC at the center is interpolated with $K = 2$ surrounding coefficients into its corresponding Δ coefficient. Note that in this illustration, the displayed coefficients are enough for converting only the centermost or previous MFCC vectors $\boldsymbol{c}_{t+k}, k \leq 0$. That is, the grayed out elements represent coefficients that do not exist yet.*

The dynamic features are defined as follows. Suppose that there exists J 13th-order MFCC vectors

$$\boldsymbol{c}_t = \begin{bmatrix} c_{t,1} \\ c_{t,2} \\ \vdots \\ c_{t,13} \end{bmatrix}, 1 \leq t \leq J, \tag{2.14}$$

which have been extracted from J adjacent analysis frames, t is a discrete time index, and $c_{t,1}, \ldots, c_{t,13}$ are the individual cepstral coefficients. Then, each MFCC vector $\boldsymbol{c}_t$ is concatenated with the corresponding 1st order time derivatives $\Delta \boldsymbol{c}_t$ known as *delta coefficients* in order to form *dynamic* MFCC-Delta feature vector

$$\boldsymbol{x}_t = \begin{bmatrix} \boldsymbol{c}_t \\ \Delta \boldsymbol{c}_t \end{bmatrix}, K+1 \leq t \leq J, \tag{2.15}$$

where MFCCs associated with frames from $K+1$ to the Jth one are converted into dynamic features. Indexing of $\boldsymbol{x}_t$ begins from $K+1$, because of the required $2K+1$ window length, when estimating the first-order derivatives.

This work utilises a formulas defined in [20] for approximation of temporal cepstral deriva-

tives. The formulas will be applied to the elements of each vector

$$\Delta \boldsymbol{c}_t = \begin{bmatrix} \Delta c_{t,1} \\ \Delta c_{t,2} \\ \vdots \\ \Delta c_{t,13} \end{bmatrix}, \tag{2.16}$$

needed by equation 2.15. The aforementioned approximation formulas are defined as follows:

$$\Delta c_{t,j} = \mu \sum_{k=-K}^{K} k c_{t+k,j} \tag{2.17}$$

and

$$\mu = \frac{1}{\sum\limits_{k=-K}^{K} k^2}, \tag{2.18}$$

where μ is a normalisation factor and $j = 1, \ldots, 13$. This work utilised $K = 4$ as the delta window radius.

2.3 Classification Methods

Two classification methods will be discussed in this section: Gaussian mixture models and deep neural networks. The state-of-the-art in SED is set by classification methods based on deep neural networks [26, 30], but the use of GMMs in combination with standard features such as MFCCs and their first-order deltas is commonplace in the preceding works [5, 31, 32]. Despite being surpassed by the novel methods, however in this work, GMMs are still very useful and well-tested method for setting a classification performance baseline – utilising them is also straighforward.

2.3.1 Gaussian Mixture Models

When modelling a real-world data set, machine learning algorithms based on *mixture modelling* do not solely rely on the modelling capacity of a single distribution, but instead take the linear combination of multiple distributions, in order to achieve better fit for the data. Gaussian mixture models are based on this mixture approach and can potentially overcome the limitations of a single distribution by being able to approximate almost any continuous density. [17, pp. 110–111]

Consider the multivariate d-dimensional Gaussian distribution

$$\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{d/2}} \frac{1}{|\boldsymbol{\Sigma}|^{1/2}} e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}} \boldsymbol{\Sigma}^{-1} (\mathbf{x}-\boldsymbol{\mu})} \tag{2.19}$$

with an observation (or feature) vector $\mathbf{x} \in \mathbb{R}^d$, mean $\boldsymbol{\mu} \in \mathbb{R}^d$ and covariance matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{d\times d}$. Equation 2.19 is the basic distribution used when defining a *mixture of Gaussians*, which takes the form [17, p. 111]

$$p(\mathbf{x}) = \sum_{g=1}^{G} \pi_g \mathcal{N}\left(\mathbf{x}|\boldsymbol{\mu}_g, \boldsymbol{\Sigma}_g\right). \tag{2.20}$$

Notice that equation 2.20 consists of G Gaussian densities defined by equation 2.19 as individual mixture *components*, each with an individual mean $\boldsymbol{\mu}_g$ and covariance $\boldsymbol{\Sigma}_g$. The G Gaussian densities $\mathcal{N}\left(\mathbf{x}|\boldsymbol{\mu}_g, \boldsymbol{\Sigma}_g\right)$ are multiplied with *mixing coefficients* π_g that have the following constraints [17, p. 111]

$$\sum_{g=1}^{G} \pi_g = 1 \tag{2.21}$$

and

$$0 \leqslant \pi_g \leqslant 1. \tag{2.22}$$

In this work, the set of parameters $\{\pi_g, \boldsymbol{\mu}_g, \boldsymbol{\Sigma}_g\}, g = 1, \ldots, G$ in equation 2.20 are estimated such that they represent a single sound event category – multiple categories require estimating their own set of parameters accordingly. The estimation is based on the fit to a given training data, which will be defined next for a single sound event category. Given a subset $\boldsymbol{S}_c$ of i.i.d. training data $\boldsymbol{S}$ that corresponds to equation 2.1, $\boldsymbol{S}_c$ is defined as

$$\boldsymbol{S}_c = \{\mathbf{x}_{c,m} \mid m = 1, \ldots, M_c\} \in \boldsymbol{S}, \tag{2.23}$$

where the M_c observations $\mathbf{x}_{c,m} \in \mathbb{R}^d$ belong to a sound category c. The parameters π_k, $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}_k$ are still unknown and need to be fit into the training data subset $\boldsymbol{S}_c$. The most popular way to fit those parameters to match the distribution of the observations $\mathbf{x}_{c,m}$ is to estimate the maximum of log likelihood function

$$\ln p(\mathbf{X}_c|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{m=1}^{M_c} \ln \left\{ \sum_{g=1}^{G} \pi_g \mathcal{N}\left(\mathbf{x}_m|\boldsymbol{\mu}_g, \boldsymbol{\Sigma}_g\right) \right\} \tag{2.24}$$

using the *expectation-maximization* (EM) algorithm [33]. In equation 2.24, $\mathbf{X}_c$ is a $M_c \times d$ matrix representation of the observations belonging to subset $\boldsymbol{S}_c$, which is given by

$$\mathbf{X}_c = \begin{bmatrix} \mathbf{x}_{c,1}^{\mathrm{T}} \\ \vdots \\ \mathbf{x}_{c,m}^{\mathrm{T}} \end{bmatrix}. \tag{2.25}$$

It can be proved that the value of log likelihood function in equation 2.24 is guaranteed to increase as EM algorithm progresses through each iteration, and subsequently better fit

for the parameters is found. Eventually the changes in the likelihood function value after each iteration become very small, and at that point the training process for that model is finished, although it is possible that EM algorithm has converged to a local maximum instead of a global one. [17, pp. 437–438]

The steps for training a single model for an arbitrary sound event category have now been summarised. This scheme of estimating the parameters with EM algorithm simply needs to be repeated multiple times in order to obtain a model for each sound event category, based on the training data $\boldsymbol{S}_c, c = 1, \ldots, C$.

Finally, the likelihood of belonging to categories $c = 1, \ldots, C$ for a new segment $\widehat{\mathbf{X}} \in \mathbb{R}^{M_{new} \times d}$ and has M_{new} observations, can be obtained based on

$$p\left(\widehat{\mathbf{X}} \mid c\right) = \prod_{m=1}^{M_{new}} p\left(\mathbf{x}_m \mid c\right) \tag{2.26}$$

and assuming that the observations within the segment $\widehat{\mathbf{X}}$ are independent of each other. The most likely category for the segment $\widehat{\mathbf{X}}$ can be selected based on the highest likelihood among $p\left(\widehat{\mathbf{X}} \mid 1\right), \ldots, p\left(\widehat{\mathbf{X}} \mid C\right)$.

2.3.2 Deep Neural Networks

In the past, GMM-based methods set the state-of-the art in SED-related classification tasks, but they have been surpassed by methods based on deep learning [34][35, p. 83]. There are many suitable deep learning model architectures that could be used for SED-related machine learning tasks, but the scope of this work only covers DNNs. Investigating other deep learning models in real-time SED context is left as future work.

DNN is a *supervised deep learning* algorithm that has been developed based on pre-existing research on *feed-forward neural network*, and it can be used to solve both regression and classification problems. In order to make use of this algorithm, there first exists a training stage, in which optimal model parameters are learned from a labelled set of data. These parametes are being iteratively optimised during the training process with a *gradient descent* based algorithm. The training stage is followed up by the usage stage, in which prediction results are produced based on new set of unlabelled examples that were not shown during model training – e.g., the results could be classification labels outputted by the SED system.

The above scheme as such, cannot yet be called supervised deep learning. For this purpose, the concept of *representation learning* is also involved. In practice, feature extraction step is simply extended in a way that more abstract features are generated successively from simpler ones. As a result, a layered hierarchy of representations is learned – the idea is illustrated in Figure 2.5. [36, p. 5]

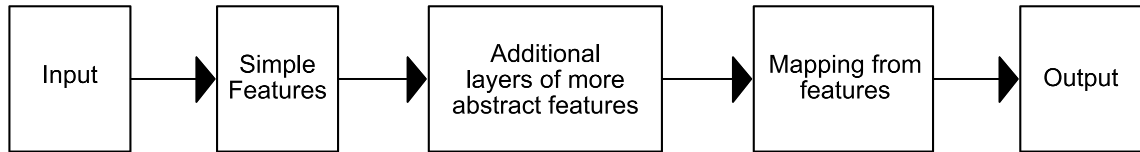


Figure 2.5. *Diagram illustrating how deep learning is a type of representation learning: more abstract features are learned in layers one after another, and each layer is based on the previous simpler one. The diagram is derived from a similar one in [36, p. 10].*

In the case of DNNs, this layered hierarchy is achieved by modelling a feed-forward neural network that possesses one or more *hidden* layers, where features belonging to deeper layers are hoped to have learned more abstract features than the ones in the preceding layers. Now that DNN has been discussed on a general level from the perspective of deep learning, the actual details can be defined next.

Model of a Neuron

The basic *unit* of computation in feed-forward neural network, is the model of *an artificial neuron* shown in Figure 2.6. This single neuron by itself constitutes a minimal neural network. It takes features $x_1, x_2, \ldots, x_d$ as input and transforms them into an outputted *activation value* $a \in \mathbb{R}$ via a series of computations. Those computations can be divided into the following two steps: first, calculating the linear combiner output $z \in \mathbb{R}$, and then the activation output value $a \in \mathbb{R}$. The inputs are first multiplied with their *synaptic* weights $w_1, w_2, \ldots, w_d \in \mathbb{R}$ correspondingly, then those synapses are summed together along with the bias term $b \in \mathbb{R}$, and finally the amplitude of the resulting sum is limited with an *activation function* φ. [15, p. 10]

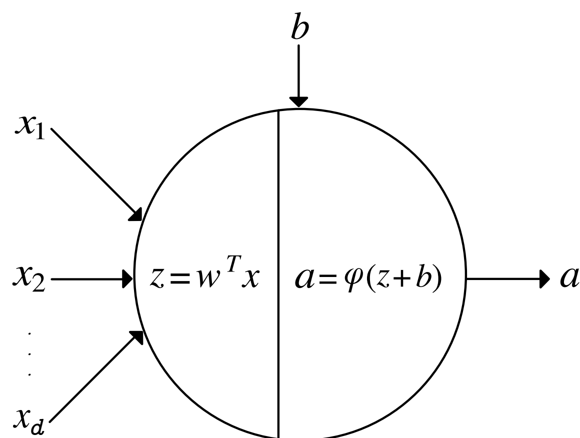


Figure 2.6. *The structure of an artificial neuron. This single unit of execution itself comprises a minimal neural network that estimates the output of logistic regression. The diagram representation style has been derived from a similar one in [37].*

The computations performed by an arbitrary neuron r can be expressed with the following

equations:

$$z_r = \sum_{j=1}^{m} w_{rj} x_j = \mathbf{w}_r^T \mathbf{x}, \tag{2.27}$$

$$a_r = \varphi\left(z_r + b_r\right), \tag{2.28}$$

where

$$\mathbf{w}_r = \begin{bmatrix} w_{r,1} \\ \vdots \\ w_{r,m} \end{bmatrix} \in \mathbb{R}^m \tag{2.29}$$

is vector of weights corresponding to d inputs and

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix} \in \mathbb{R}^d. \tag{2.30}$$

is a vector of d features corresponding to the neuron inputs $x_1, x_2, \ldots, x_d$. The signal z_k of equation (2.27) constitutes a *linear combiner* output, which simply computes the weighted sum of the inputs $x_1, x_2, \ldots, x_d$ by applying the synaptic weights $w_1, w_2, \ldots, w_d$ to each input signal individually. The linear combiner output z_r is further processed into the activation output a_r of an arbitrary neuron r, by applying an *activation function* φ as defined in equation (2.28); this function serves the purpose of squashing the amplitude of the linear combiner output z_r to permissible limits, which is typically bound to the interval $[0, 1]$ or $[-1, 1]$. Notice also the bias term b_r of equation (2.28), which simply has the effect of either increasing or decreasing the net input of the activation function φ. [15, p. 11]

Multilayer Feed-forward Neural Network

A single neuron itself is not capable of modelling input-output mappings involving complex hyperplane decision boundaries. Instead, many neurons should be used to form a neural network architecture that enables the network to behave as a *universal approximator*. Feed-forward neural network is one of the applicable architectures, but it should also fulfill requirements of the *universal approximation theorem*: in short, the network is required to possess one or more hidden layers [38]. Hidden layer simply represents a stack of one or more neurons that have been placed in between the *input* and *output* layers of the network, which is illustrated in Figure 2.7. [17, p. 230]

Let us discuss the roles and flow of data from layer to layer. The input layer of a neural network is a bit special compared to other layers, because the nodes represent the input values $x_1, x_2, \ldots, x_d$ instead of neuron activations. Despite this, the input layer units can still be viewed as activation values passed on to the first hidden layer $l = 1$. Furthermore, it is also common to denote them as units $a_1, a_2, \ldots, a_d$ instead of $x_1, x_2, \ldots, x_d$ in order to achieve uniform notation for the activations of all layers. Similar to Figure 2.7, if more

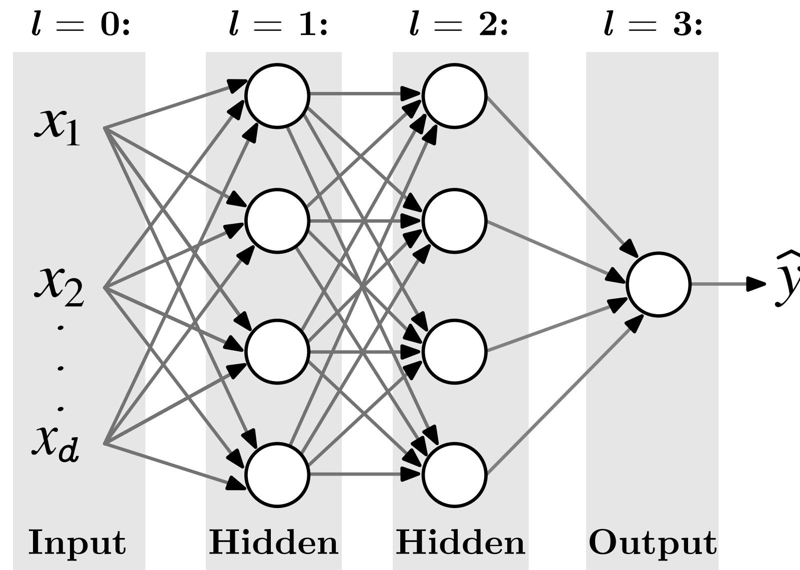


Figure 2.7. *Multilayer feed-forward neural network with two hidden layers.*

hidden layers exist after the first one, the activations of the first hidden layer $l = 1$ would be passed on to the second hidden layer $l = 2$, and finally those in turn would be passed on to the output layer. The outputted value $\hat{y}$ of the whole network is acquired simply by computing the activation value of the output layer unit. At this point, note that nothing limits utilising many neurons in the output layer – a single unit is used for simplicity at the moment.

The above describes the flow of information from layer to layer until the output has been reached, but there exists one more consideration: neurons are *fully connected*, which means that the activation output value of any unit from the previous layer $l - 1$ is associated with all the units in the current layer l. Visually, this simply shows up as full amount of connections (arrows) between neurons, e.g. in Figure 2.7. Note that in feed-forward neural networks, the neuron connections are directed only forward, from previous layer to the next, but not vice versa.

The concepts introduced so far now allow representing the output activations for a layer of neurons. In practice, the equations 2.27 –2.28 simply need to be extended such that the computations can be flexibly repeated for many neurons and layers. First, assume a feedforward neural network with L hidden layers corresponding to indices $1, \dots, L$. The input layer is denoted by layer $l = 0$ and output layer by $l = L + 1$. An arbitrarily chosen layer $l \in \{0, 1, \dots, L + 1\}$ from the network contains N_l neurons. Now, equation 2.29 that represents the weights of a single neuron, can be extended to become weight matrix

$$\mathbf{W}^{[l]} = \begin{bmatrix} (\mathbf{w}_1^{[l]})^T \\ (\mathbf{w}_2^{[l]})^T \\ \vdots \\ (\mathbf{w}_{N_l}^{[l]})^T \end{bmatrix}. \tag{2.31}$$

The rows of matrix $\mathbf{W}^{[l]}$ correspond to the synaptic weights of the individual neurons located within layer $l \in \{1, 2, \ldots, L+1\}$. Correspondingly, based on equation 2.27 and equation 2.31, a vector containing the amount of N_l linear combiner outputs

$$\mathbf{z}^{[l]} = \begin{bmatrix} z_1^{[l]} \\ z_2^{[l]} \\ \vdots \\ z_{N_l}^{[l]} \end{bmatrix} = \begin{bmatrix} (\mathbf{w}_1^{[l]})^T \mathbf{a}^{[l-1]} \\ (\mathbf{w}_2^{[l]})^T \mathbf{a}^{[l-1]} \\ \vdots \\ (\mathbf{w}_{N_l}^{[l]})^T \mathbf{a}^{[l-1]} \end{bmatrix} = \mathbf{W}^{[l]} \mathbf{a}^{[l-1]} \tag{2.32}$$

can also be acquired. Finally, this allows computing the lth layer activation output vector

$$\mathbf{a}^{[l]} = \boldsymbol{\varphi}\left(\mathbf{z}^{[l]} + \mathbf{b}^{[l]}\right) = \boldsymbol{\varphi}\left(\begin{bmatrix} z_1^{[l]} + b_1 \\ z_2^{[l]} + b_2 \\ \vdots \\ z_{N_l}^{[l]} + b_{N_l} \end{bmatrix}\right) = \begin{bmatrix} \varphi\left(z_1^{[l]} + b_1\right) \\ \varphi\left(z_2^{[l]} + b_2\right) \\ \vdots \\ \varphi\left(z_{N_l}^{[l]} + b_{N_l}\right) \end{bmatrix} = \begin{bmatrix} a_1^{[l]} \\ a_2^{[l]} \\ \vdots \\ a_{N_l}^{[l]} \end{bmatrix}, \tag{2.33}$$

where

$$\mathbf{b}^{[l]} = \begin{bmatrix} b_1^{[l]} \\ b_2^{[l]} \\ \vdots \\ b_{N_l}^{[l]} \end{bmatrix} \tag{2.34}$$

is the vector of bias values for layer $l \in \{1, 2, \ldots, L+1\}$.

The details up until this point of calculating the estimated network output value comprise a *forward propagation* step, and are the most relevant neural network details in this work. This is due to equations 2.27 –2.34 directly influencing the inference speed of the DNN-based SED system, and thus forming a good target for possible speed optimisations in order to produce sound event labels as rapidly as possible. In the end, there was no need to optimise the computational speed of these equations, because the inference speed of popular deep learning frameworks such as TensorFlow and PyTorch turned out to be good enough. The details of training a model will be omitted as they are not directly required for implementing a real-time SED system.

3 REAL-TIME PROCESSING

This Chapter introduces the key components and concepts that allow designing a real-time SED system later in Chapter 4. Some of the concepts discussed so far fit into the real-time paradigm without modifications. However, successful transformation of an offline SED system into its real-time counterpart does also require introducing some new concepts yet to be discussed, and also altering pre-existing ones.

Compared the offline audio audio input processing, audio is being collected non-stop and simultaneously instead of separately reading pre-recorded clips. Temporally, both short and long input data is handled by the real-time input processing scheme [39], namely the following:

1. a block (or chunk) of audio data that is currently being processed;
2. the audio block expected to be received after handling the current one;
3. and possibly the all the input samples collected since the real-time audio processing session was initiated.

Regarding the third point in the listing above, It is beneficial to cache the recorded input signal for later use, since the whole system behaviour can be verified with it – this is crucially important in the evaluation experiments of Chapter 5. The first two listed points rather suggest that a real-time audio stream processing is required due to the nature of the processing. The usage stage in Figure 3.1, which is based on the off-line counterpart of Figure 2.1, implements such an audio stream processing scheme.

Compared to its off-line counterpart, the real-time scheme has been transformed to always extract successive individual feature vectors based on the latest available analysis frame. In practice, audio input stream is polled in multiples of hop size and features are extracted after every frame-block update. The feature extraction pipeline for extracting static and dynamic feature vectors does not require modifications, meaning that the procedure derived in Section 2.2 is still utilised also in the real-time SED system. However, this chapter shows that the way of reading audio input and storing the extracted feature vectors has to be changed because of the real-time flow of data. Furthermore, the execution model needs to be changed from sequential to concurrent: while new feature vectors are accumulated into the incomplete audio segment, the classifier simultaneously computes new prediction results based on the previous segments.

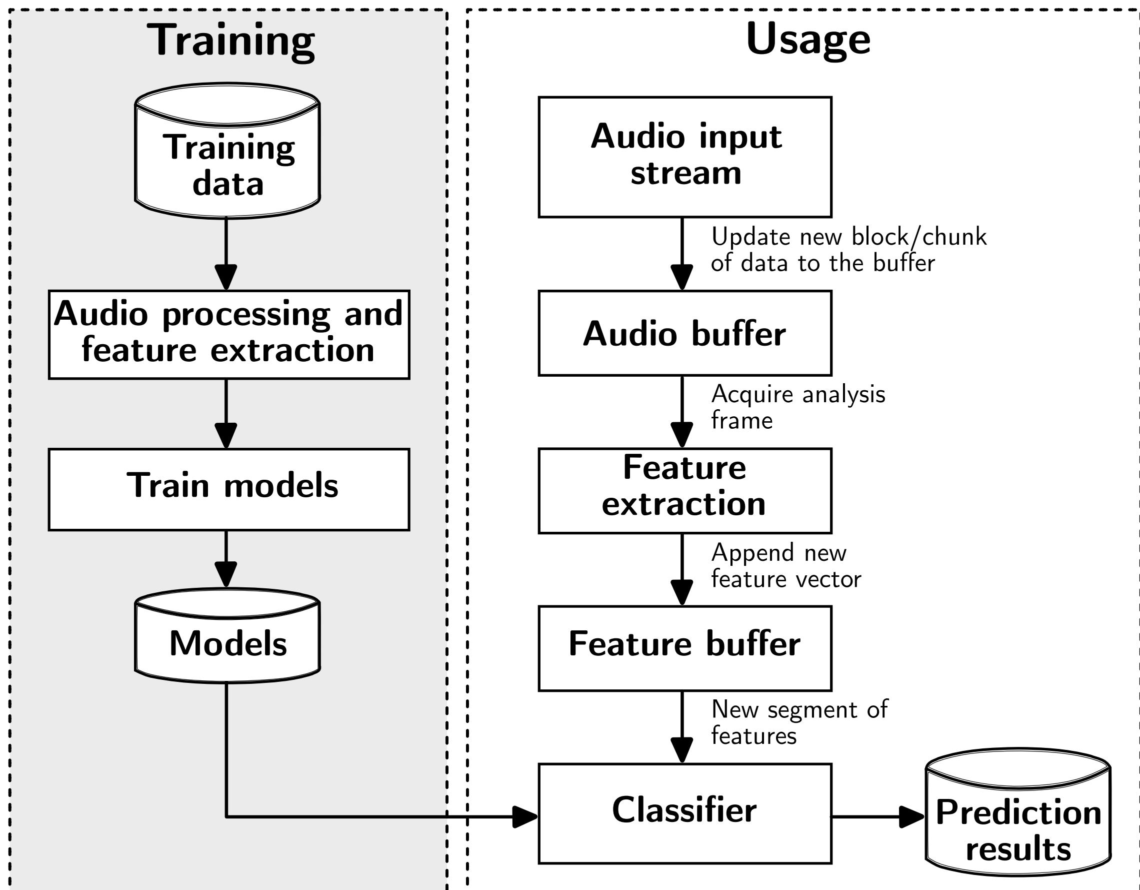


Figure 3.1. *Real-time classification scheme. The training stage is still identical to the off-line system, but the approach to SED has been altered in usage stage.*

3.1 Audio Input

Processing a real-time audio stream is at the essence of this work and there exists a variety of common tools for handling these streams. Unfortunately, not all of these tools are multi-platform or even provide Python bindings. Among all free software choices, a real-time audio API called PortAudio, has distinguished itself as a reliable choice that supports a number of host languages, including Python. PortAudio provides support for many system-dependent APIs, which are conveniently used through a single mutual API.

At the time of writing, the following packages are popular choices for using PortAudio with Python: `pyaudio` and `python-sounddevice`. Initially, this work made use of `pyaudio`, but later a choice was made to explicitly use `python-sounddevice` instead, as it offers a very user-friendly way of reading and writing blocks of audio directly with `numpy` arrays. The relationship between system-dependent APIs, PortAudio, `python-sounddevice` and the user-developed Python application is illustrated in Figure 3.2. It shows that PortAudio encapsulates away the platform-dependent APIs, which directly communicate with the audio hardware, making it also unnecessary for a programmer to get familiar with any of the individual platform-dependent audio APIs. Simply using the same PortAudio API is enough across all supported platforms and the low-level audio hardware interaction is left

to the system-dependent APIs instead. [40]

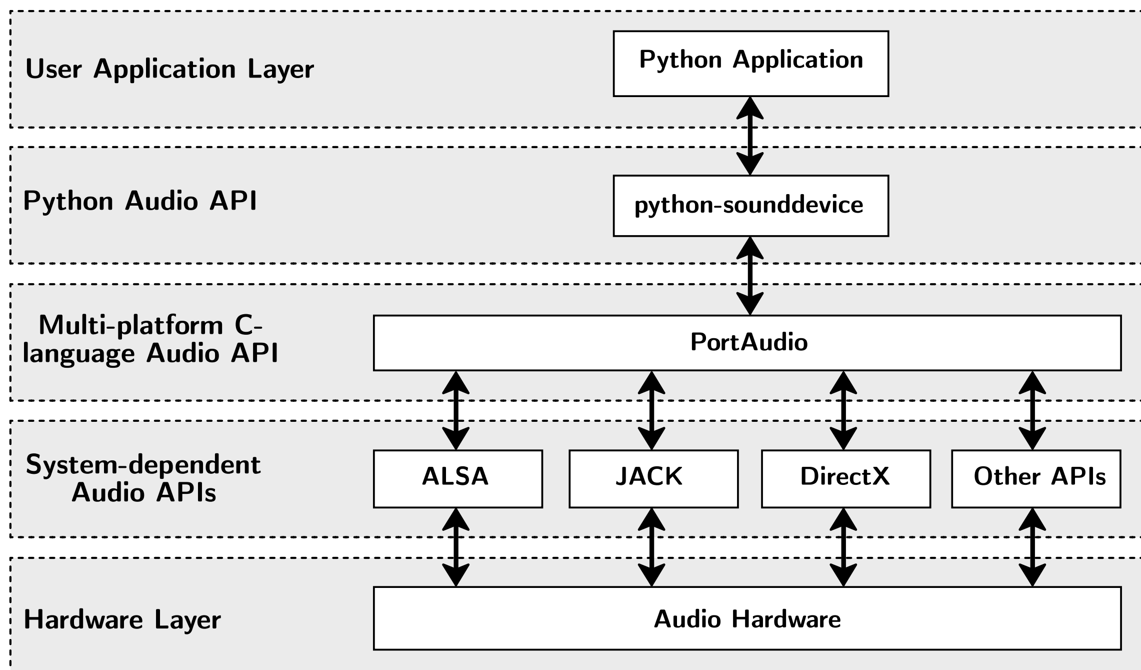


Figure 3.2. *Diagram illustrating a Python application using python-sounddevice and how it communicates with audio hardware through low-level APIs. The diagram is derived from a similar one in [40].*

Processing audio input with `python-sounddevice` is very simple. PortAudio is first initialised and input stream is then opened separately. This open stream can be polled for a desired amount of latest audio samples as they become available.There exists two operating modes for audio processing in PortAudio: blocking and callback modes. Both of these modes were experimented with, where *blocking-mode* turned out to be very straightforward and simple to use, while callback mode is better for achieving low audio latency. This work focuses on the use of *callback-mode*, as it allows exploring the full potential of real-time audio processing with Python. The blocking mode has merely been designed to used for simple applications and pedagogical purposes. [40]

The callback mode allows executing a user-defined function that consumes an audio block as a response to a request from an active `python-sounddevice` stream. The user-supplied callback is required to have a specific function signature:

```
def callback(indata: ndarray,
             frames: int
             time:   CData,
             status: CallbackFlags) -> None
```

The first callback function argument `indata` is the audio buffer containing the latest block of input signal. It is a multidimensional `ndarray` from the `numpy` package with the shape `(num_rows, num_channels)`, where `num_rows` matches the number of audio samples for the buffer data and `num_columns` matches the number of audio channels in the active input stream; assume single-channel data in this work, because spatial data manipulation

is not performed as a pre-processing step. The second callback argument `frames` tells the length of the `python-sounddevice` input buffer. The third argument `time` is a `CData` structure from Python CFFI package that indicates the analog-to-digital conversion timing for the first sample captured in the `indata` input buffer array and the time the callback function had been invoked. Finally, the fifth argument `status` tells important information about the input buffer with a set of flags: whether the buffer has been inserted successfully to `indata` or the possibility of it being dropped to resolve *underflow* and *overflow* situations.

3.2 Audio and Feature Buffering

This section describes the ways data is managed and updated in the real-time processing pipeline of Figure 3.1. First, it is necessary to understand in detail how the usage stage has diverged in the real-time system from its off-line counterpart. In frame-blocking, the length of adjacent audio segments is still fixed to one second like in Chapter 2, meaning that each collected segment possesses an equal amount of J analysis frames and feature vectors. The system keeps in memory only the latest frame-block consisting of S audio samples and updates it periodically with the hop size of H new ones, where the new samples are polled from the audio input stream. A single feature vector is extracted after every frame-block update from an analysis frame and the segment can be classified every time J new feature vectors have been accumulated. Based on these observations, the real-time SED system should keep in memory S audio samples and J features vectors. Furthermore, continuous updates are required as new audio samples and features vectors are acquired. Storing the amount of $2K+1$ static MFCCs is also required in order to obtain the dynamic ones as illustrated in Figure 3.3.

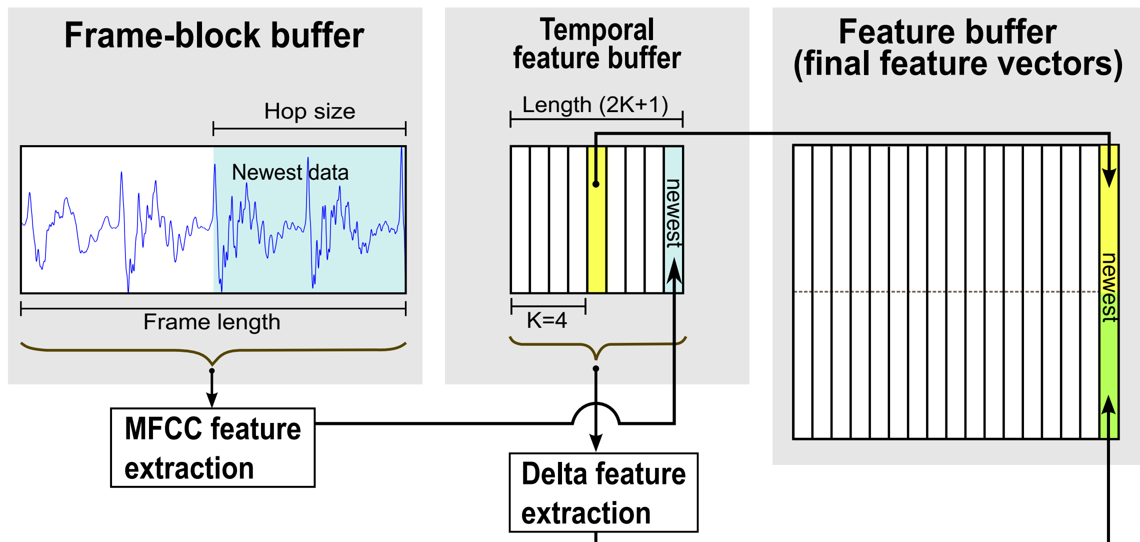


Figure 3.3. *Buffering hierarchy of the real-time SED system. Updating the frame-block triggers an extraction of MFCC feature vector and it will be pushed into the static buffer. That also triggers the ΔMFCC extraction and consequently a concatenated feature vector will be pushed into the feature buffer.*

The technique known as *circular buffering* is a good way to manage the stored audio samples and feature vectors in the situation described above [39, p. 507]. Circular buffer is represented as a fixed-length array, where updating with one or more new data samples causes an equal amount of the oldest ones getting discarded [21, p. 766]. Visually, it is common to represent circular buffer as a circle (see Figure 3.4), where both ends of the array are adjacent with each other. In the illustrated buffers, the slots labelled with capital letters represent fixed-size blocks of data, which also possess an index number for marking the data positions in the array. Reading and writing operations in a circular buffer are governed by write and read pointers, where their current index positions indicate a starting location for accessing a chosen amount of adjacent data slots for either reading or writing. When the blocks of data are written to the assigned slots, write pointer is moved to the index of a slot positioned immediately after the written ones, and correspondingly the same applies for read pointer with the exception that slots are simply read instead of written.

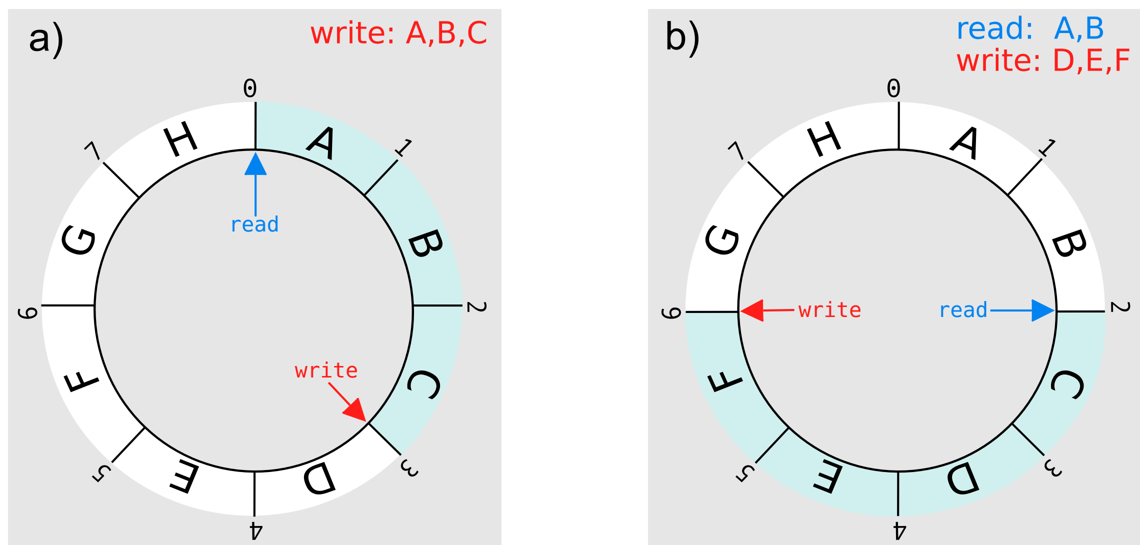


Figure 3.4. *The concept of circular buffering is illustrated in panels a) and b). In the first iteration of panel a), data is written to slots A, B and C and the write pointer is incremented by three. In the next iteration of panel b), slots A and B are read and slots D, E and F are written; read pointer is incremented by two and write pointer by three. The slots highlighted between the pointers marks an area of active data and the white slots are free for writing.*

Circular buffers can be utilised for real-time SED purposes by considering the data slots between read and write pointers as meaningful, while the slots located outside can be freely overwritten with new data (see the areas highlighted in blue and white in Figure 3.4). The first scenario, where buffering is required involves managing the audio samples of the latest frame-block like in Figure 3.3. The data slots in a circular buffer are made to represent the latest audio samples received through the audio input stream. This kind of frame-block buffer possesses the capacity for frame length S amount of slots, where the individual slots represent audio samples; new data is written to the buffer strictly in multiples of frame hop H. Initially the buffer needs to be filled with S latest audio samples,

which can be done e.g. by first setting the write pointer to the zeroth index, and then writing to its slot and the following $H-1$ slots, after which write pointer is again located at the zeroth index due to travelling a full circle. For each upcoming frame-block update, H new audio samples needs to be written into the buffer, but there is not yet enough successive free slots available for writing. For this purpose, read pointer is moved forward H slots, which stages the H oldest samples in the buffer to be overwritten with new data. This update mechanism follows what was stated earlier in this section regarding circular buffers: when certain amount of new samples are added, an equal amount of the oldest ones will be discarded.

Two additional circular buffers, which are known as the *temporal feature buffer* and *feature buffer* (see Figure 3.3), are required for containing the latest static MFCC and ΔMFCC feature vectors. Buffering of feature vectors differs from frame-block buffering in two ways: each data slot represents a feature vector in both feature buffers; and new feature vectors are updated to their corresponding feature buffers one at a time, where the oldest vector in both cases will be overwritten by the new one. The temporal feature buffer contains $2K+1$ amount of MFCC vectors, which are utilised for extracting a new delta feature vector every time the buffer is updated. The extracted delta feature vector is concatenated into a final feature vector with the copy of its corresponding MFCC vector, where the obtained final feature vector is then updated into the feature buffer. The feature buffer contains the amount of final feature vectors that can be extracted from a fixed-length one second audio segment in order to classify based on the features. However, classification is performed only when the feature buffer contains completely new final feature vectors.

In Python, circular buffering can be utilised in two ways: the `deque` datatype in Python built-in `collections` package and the `numpy.roll` algorithm in the `numpy` package. The `deque` datatype can be utilised as a circular buffer, when the maximum array length is given during initialisation of a `deque` instance. In this work, `numpy.roll` was utilised for convenience with linear algebra and the real-time SED system still functioned properly with the performance of `numpy.roll` circular buffering.

3.3 Concurrency and Parallelism

Typically, Python scripts written for academic research purposes consist of language constructs approachable to both entry and veteran level programmers alike. E.g. the offline SED system of Figure 2.1 is straightforward to implement with functions, classes, modules, easy to use built-in and 3rd party packages; furthermore, on such a basic program, the written Python instructions can be executed *sequentially*, making the execution behaviour predictable enough. However, in order to design and implement the online SED system while sufficiently also fulfilling soft real-time requirements, solid understanding of more advanced execution paradigms, concurrency and *parallel*, becomes necessary.

Parallel programming involves decomposing the computations of an algorithm into several subtasks, which will be distributed to some parallel capable hardware like Central

Processing Units (CPUs), Graphics Processing Units (GPUs) or Neural Processing Units (NPUs), assuming that the algorithm is inherently parallelisable. The expectation is that the algorithm executed in parallel finishes a task faster overall compared to its sequential counterpart, making better use of the available hardware. For various applications and purposes, there exists parallel optimised Python packages, which allow developers to even avoid implementing parallel algorithms and focus instead on other software engineering or research tasks, saving valuable project time. It is exactly the case in this work, e.g. notably in classification and feature extraction tasks of Figs. 2.1 and 3.1, in which Python packages such as `numpy`, `scikit-learn` or `TensorFlow` transparently perform parallel optimised operations besides various other non-parallel optimisations present in them. [41, pp. 3–4]

Since optimizing machine learning tasks for processing speed is beyond the scope of this work and the focus is instead on concurrency problems, and also the correct ways of applying them to designing a real-time SED system, parallel programming will not be further explored in this work. This particular paradigm becomes more important after the system has been implemented and specific parts of it must be tuned for intended computational speed. The necessities for understanding concurrency in the context of Python will be introduced in the following subsections.

3.3.1 Program and Process

When OS user launches a program, a *process* is created, which is an instance of that program in execution. Each instance is unique, so launching the same program multiple times would result in equally many processes running that program code individually. Compared to a process, program is not being executed at all and merely comprises the machine instructions stored on a application binary file – in the case of interpreted languages such as Python, program comprises merely the source code.

A process can also be thought as an entity that groups together wide variety of related resources as a unique entry in the OS *process table*. Process resources are required for various important purposes, e.g. to inspect or change process state or manage allocated memory. Significantly, each process needs to contain resources to enable *multiprogramming*, which simply means that OS rapidly switches between the processes, only a single process being active for each available CPU core at any given time. Finally, processes cannot directly access the memory space of each other, but requires using protocols designed for such situations, e.g. with an Interprocess Communication (IPC) mechanism or using *shared memory* between the involved processes. [7, pp. 85–95]

3.3.2 Threading and Global Interpreter Lock

Sometimes a single Python applications needs to oversee and perform multiple activities simultaneously. Similar to many other programming languages, Python also supports

multithreading – more frequently referred to as *threading* – that allows simultaneous execution of code in multiple locations of the running program. In essence, thread is just like a process, but it is created, managed and overseen by its parent process. Each process is considered to be at least *single-threaded*, but more threads can be launched and disengaged during execution, depending on the needs and circumstances. Compared to creating a process, threads do not need to allocate nearly as many resources as processes do, so in a sense they are also called *lightweight processes*. Using substantial amount of processes to closely work on a task would consume a lot of OS resources. For this reason, threads are meant to be the better choice by design for cooperating on a task inside a process. [7, pp. 97–106]

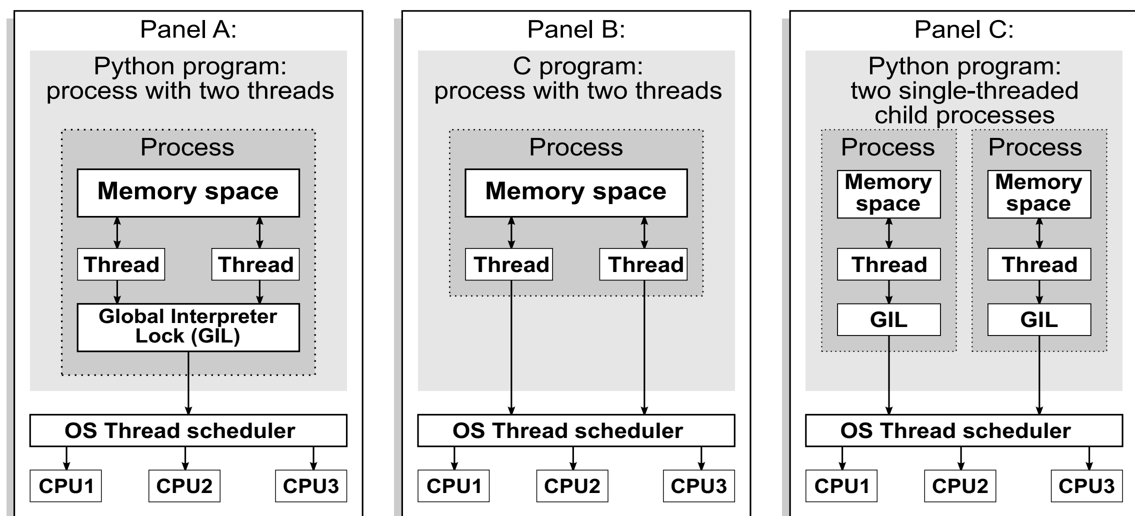


Figure 3.5. *Panel A illustrates the GIL-limitation of a threaded Python program. The C program in panel B possesses the same threading architecture except that it is not limited by GIL; fortunately, comparable concurrency performance in Python can be achieved with the architecture of panel C.*

Unfortunately, owing to certain design choices made in the standard Python interpreter, Python threads is not as flexible compared to their counterpart implementations in many other programming languages. More accurately, due to a notorious thread synchronization mechanism known as the Global Interpreter Lock (GIL), standard Python is primarily suitable for I/O-bound tasks and concurrent execution speed (based on threads) will suffer in the presence of CPU-bound tasks. Figure 3.5 illustrates in which manner the threads of a running Python program are restricted, when compared against those of an equivalent program implemented in C-language. Comparing panels A and B, it can be seen that the only difference is the lack of GIL component in the latter. In the running C program, threads are simultaneously scheduled by the OS, which is true multiprocessing. However, threads running in the Python program are restricted by GIL to merely *cooperate* with each other, which means that in the worst-case scenario only one of them is active simultaneously. Fortunately, GIL is released for I/O operations, so Python threads can be very useful as long as CPU-bound processing inside threads is avoided. There exists workarounds to solve the performance dilemma of panel A in Figure 3.5 and a very straightforward one will be presented in the next section.

3.3.3 Process-based Parallelism

As briefly mentioned in Section 3.3.2, processes are quite resource-heavy compared to threads. However, in Python programming, they do have the advantage of being truly parallel and thus being suitable for CPU-bound tasks, unlike threads. Therefore, processes can be useful and this section aims to show exactly how. In Figure 3.5, panel C illustrates an alternative to Python threads in order to achieve similar degree of concurrency as in panel B. The Python main process of panel C achieves it by spawning a pair of single-threaded child processes under its control, instead of threads, like in panel A. In Python, `multiprocessing` is the package that enables Python programmers to implement *process-based parallelism* to their programs – the parallelism in the definition refers to concurrency, basically. If the problem solution requires spawning only a limited amount of Python processes for each CPU-bound task, this approach is reasonable. E.g. on a networking or Web-based program, it is not uncommon to spawn massive number of threads to handle all the I/O workload – replacing all those threads with processes would be disastrous. [9, Ch. 20]

Unlike in the case of threads, memory space is separate between processes, so the spawned child processes in panel C cannot closely work on a task the same way threads do. Threads can easily share and access mutual resources and data inside the same running program, but attempting the same feat with processes requires indirectly passing data between them by using any supported IPC mechanisms or allocating a shared memory area between the involved processes. The downside is that more complexity and overhead is introduced compared to thread-based communication. The shared memory approach is typically not the first option programmers start to experiment with, as it requires involving *synchronization primitives* such as *mutex* or *semaphore* for serialising access to the shared memory. Attempting it can also be very error-prone, so first testing the performance of a message passing mechanism is safer [9, p. 415] in the beginning, even though more overhead has to be accepted. [9, Ch. 20]

An easy and safe methodology that sidesteps the need to directly rely on synchronization primitives is straightforwardly illustrated with the *producer-consumer* pattern, which has been well-known to be utilised in a wide variety of programming problems. The pattern consists of an entity that continuously *produces* data in some designated manner and inserts it into a queue. Simultaneously, another entity is monitoring the same queue and constantly *consumes* data from it for some pre-determined purpose. Thus, the queue has been shared among the producer and consumer with synchronized access to its data.

This pattern can be applied to the process-based parallelism scheme in panel C of Figure 3.5, when one process is assigned as the producer and another one as the consumer. Figure 3.6 illustrates these two concepts fused together. The queue shared between the two processes is safely able to pass data from producer to consumer. In Python, `multiprocessing.Queue` would be the typical choice for the queue object as it synchronizes the access between processes that *put* items in the queue and *get* items from the

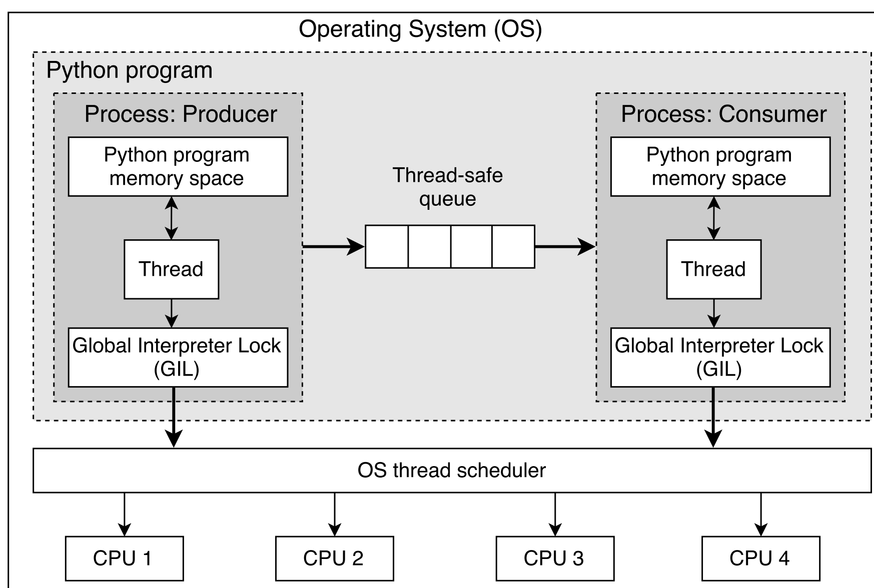


Figure 3.6. *A Python example of Producer-consumer pattern with child processes in place of threads.*

queue. The queue introduces some copy overhead as it has been implemented using another Python object called `multiprocessing.Pipe` that serialises the data put into the queue into a safely allocated area of the main memory. [9, Ch. 20]

In this work, `multiprocessing.Queue` scaled without any issues when experimenting with the producer-consumer pattern and implementing the system. In summary, the process-based producer-consumer pattern scales well, is easy and safe to use and allows circumventing the performance issues of Python threads as long as one keeps the spawned processes single-threaded. [9, Ch. 20]

3.3.4 Coroutines with asyncio

Asyncio package , which enables the use of coroutines in Python, has been a standard part of the language since Python 3.4.1 release. At the time of writing, coroutines are still considered relatively new advanced concept in Python and even the syntax has still evolved throughout the releases. Python coroutines enable something known as *asynchronous programming*, which allows implementing concurrency without launching new threads – this is possible due to executing the coroutines inside a single thread of execution. Threads are frequently characterised as being lightweight processes. Correspondingly, it is then appropriate to characterise `asyncio` coroutines as being lightweight threads – that is, even lighter than lightweight processes. This comparison is intuitive also, because a single process oversees multiple threads of executions, and correspondingly a single thread oversees multiple coroutines.

Essentially, coroutines are simply tasks that are being scheduled one-by-one based on some policy within an active event loop. They are and should be also very short-lived, since only a single thread within a single process is scheduling them – one could cause

the program to freeze by scheduling a long-lived coroutine, because the other routines would not be able to progress meanwhile. In short, coroutines achieve illusion of concurrency for very lightweight (I/O-bound) tasks and they are implemented as Python definitions with a special grammar that meet the syntactic requirements of the `asyncio` package. Coroutines provide an alternative way to effect concurrency besides threading, and they possess a different set of advantages and limitations compared to threads.

4 SYSTEM DESCRIPTION

This chapter describes real-time system designs that have been derived based on this work. The intention is to summarise interesting findings, observations and the solution of a rapidly developed system prototype belonging to a research project. Descriptions for two different system prototypes will be given and both of them are successful approaches, each of them possessing certain advantages and disadvantages over the other.

The basics introduced in the previous chapters enable describing the system prototypes straightforwardly, simply bringing together everything discussed so far, consequently allowing to describe the final systems. The system prototypes perform the following activities: processing blocks of audio from an input stream, and produce low-delay live prediction results displayed on a Graphical User Interface (GUI), based on that input. The GUI could be extended for user interactivity, so conceptually it is not limited to merely viewing results. Essentially, the prototypes implement the usage stage of Figure 3.1, extending it with a GUI that displays live prediction results and discards the step for model training.

4.1 System Design

A variety of important design choices have to be made, when implementing a reliable real-time SED system that simultaneously attends to many different tasks. E.g., the system could rapidly produce new prediction results, handle user input events, query databases or store analysed blocks input audio signal. This work approaches the major design challenges by applying relevant Python design patterns to applicable scenarios and taking into consideration the various advantages and limitations of Python. First, the essential design considerations are discussed in Section 4.1.1, which is followed by Section 4.1.2 that defines the basic system components used by the two designs. Finally, two plausible system designs will be described individually in Sections 4.1.3 and 4.1.4.

4.1.1 Design Considerations

The most important design considerations influencing real-time SED system design will be provided next. The considerations were discovered both during initial experiments and while designing and implementing the system prototypes. Their purpose is to guide the process of designing a real-time SED system such that intended performance and reliability is achieved more likely and implementation process becomes more straightfor-

ward (namely, to avoid unnecessary over-engineering). Research projects benefit from successfully applying the aforementioned targets, because they can be very quick-paced and short-lived, and a system may need to be implemented in a very short amount of time.

I/O bound vs. CPU bound

The greatest difference between the two system prototypes of this work is the way each of them implements concurrency. The standard Python implementation limits thread-based concurrency in the ways described in Section 3.3 and well-known ways to counter those limitations are utilised in both system prototype. The thread-based system prototype mainly needs to ensure that any spawned threads will not slow down due to the Python GIL issue, which is triggered by executing too many CPU-bound threads in the same program or process. Therefore, the first design consideration is that we will avoid spawning multiple CPU-bound threads under the same process.

Correspondingly, the process-based multiprocessing prototype also addresses the GIL issue, although with a different methodology detailed in Section 3.3. It replaces threads with processes in CPU-bound tasks, completely circumventing GIL and allowing these tasks to proceed independent of other threads and without slowing down those threads unintentionally. Therefore, the consideration for the process-based multiprocessing prototype is simply to replace CPU-bound tasks with processes that only have a single thread, and these processes are assumed to be directly managed by the main process of the program.

Asynchronous I/O

Besides CPU-bound processing, a specific design consideration for I/O-bound processing also exists. It needs to be decided whether to execute a task asynchronously or simply process it synchronously – the latter being the typical approach. In this work, some asynchronous programming experiments were made using `asyncio`, which is the standard Python built-in package for asynchronous programming. However, the I/O tasks of the real-time SED system prototypes were chosen to be implemented with synchronous programming instead. This is due to unforeseen difficulties, when attempting to implement very basic behaviour with `asyncio` on Windows platform. E.g., a normally simple matter of preventing program hang-up during quitting active `asyncio` event loop turned out to be challenging, even though this was never a problem while experimenting on Linux or MacOS platforms. API and OS differences were traced to be at the core of this issue. Fortunately, synchronously handling the I/O scales sufficiently well in this work and was quick and straightforward to implement, allowing to save valuable project time. In summary, the design consideration here is to favour synchronous instead of asynchronous processing.

Hidden Threads

There is a certain thread-related situation that occurs unbeknownst to the programmer, and may cause problems if ignored. Unless known for sure, e.g. by inspecting the documentation or reading the source code, a Python object instance may subtly launch Python worker threads. In this work, it was not clear until verifying from the documentation that the `Queue` object instances supported by `multiprocessing` and `threading` packages spawn these unobvious worker threads. Furthermore, inspecting the implementation of `python-sounddevice` revealed that it also spawns a worker thread in the callback mode, which is a key component in the system prototypes. The potential danger in these described circumstances is that the previously introduced design considerations of this section will be violated too much, consequently causing the speed of concurrently executed code in the system to degrade. In summary, the consideration here is simply defined as an attempt to verify whether any of the required Python packages subtly spawn Python worker threads, and then make a conclusion of their performance impact in a given circumstance.

Other Considerations

An interesting multiprocessing issue was produced with `python-sounddevice` while developing the prototypes. Any attempts to launch an audio stream that has been initialised in the execution area of a child (or worker) process, caused the SED system to crash every time. Fortunately, the solution (and the design consideration here) is very simple: `python-sounddevice` stream objects should be initialised and executed in the main process of the Python application. The reason for this issue was not investigated in this work due to lack of time, so an explanation for the phenomenon is omitted.

The most of the considerations so far attempt to mitigate the ill effects of very particular performance bottlenecks. In addition to the previous considerations, this work simply applied the basic software engineering practice of *requirements gathering* in order to reveal both functional and non-functional requirements. Omitting to do this step sufficiently well could have varying degree of negative consequences. E.g., ignoring a non-functional consideration such as remote controlling the real-time SED system might be lead to designing the system incorrectly and unable to fit into the client-server design pattern (or any other suitable ones).

4.1.2 Basic System Components

Multiple assumptions are required regarding the way the system designs abstract data. Regardless of the chosen design approach, this work utilises the Python classes shown in Figure 4.1 as the basic building blocks for a minimally functional real-time SED system.

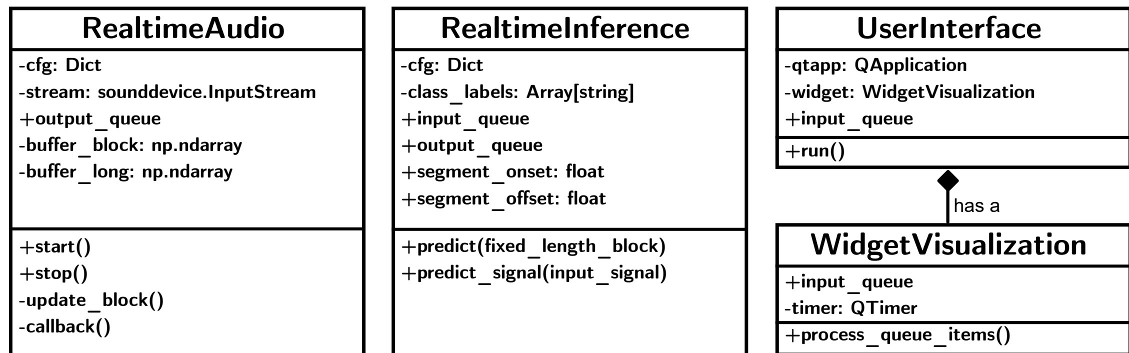


Figure 4.1. *Class diagram showing the prequisite classes required for both of the real-time SED system design approaches.*

Class Description: RealtimeAudio

The class `RealtimeAudio` allows starting and stopping a real-time audio stream, and while the stream is active, passing on incoming (and further processed) input audio blocks via repeated invocations of the `callback()` method to another object instance. In this work, that other instance in question (the recipient) is `RealtimeInference`. Passing on the data from an object to another is facilitated with an IPC mechanism, by making the attributes `RealtimeAudio.output_queue` and `RealtimeInference.input_queue` refer to the same IPC queue instance. Also, either `multiprocessing.Queue` or `threading.Queue` objects will be used for IPC purposes depending on the design approach.

Class Description: RealtimeInference

The instance of `RealtimeInference` class allows predicting an output label and score based on a fixed-length segment that has been provided as an input for its `predict` method. `RealtimeInference` has intermediately received fixed-length segment by invoking `input_queue.get()` call. The class specification does not assume in particular that which model or features should be utilised, except that audio frame-blocking scheme with some arbitrary overlap is being utilised. E.g., machine learning models may indeed require implementing a separate feature extraction scheme, but sometimes the feature extraction has been embedded directly into the model, then only requiring the raw audio signal or a fixed-length segment block as an input.

`RealtimeInference` class also has an output queue, which is meant to be utilised in the following way: invoke `input_queue.put()` to send prediction results along with useful metadata like segment onset and offset times that will be utilised by the instance of `UserInterface` class. Another thing to note is that `RealtimeInference` also must allow predicting in offline from a long input array corresponding to all processed fixed-length segments. The class achieves this with the `predict_signal()` method that produces scores and labels for all corresponding segments at once. These should be then compared to the real-time labels and scores after the real-time session has concluded, e.g. with some applicable classification metric.

Class Description: UserInterface and WidgetVisualization

The classes `UserInterface` and `WidgetVisualization` are even more minimal compared to the others described so far. In this work, the user interface simply displays the latest prediction label and score as soon as possible. However, additional activities and interaction with the running real-time SED system are more common on real use cases. The minimal user interface simply demonstrates that running a GUI that displays results, alongside the real-time SED system, is indeed feasible.

The instance of `UserInterface` communicates with the instance of `RealtimeInference` via a `Queue` instance from the `multiprocessing` or `threading` packages. The attributes `RealtimeInference.output_queue` and `UserInterface.input_queue` refer to the same IPC queue instance, allowing `UserInterface` instance to `input_queue.get()` the objects that have been `input_queue.put()` by `RealtimeInference` instance.

The `UserInterface` class is structured in the following manner. It possesses the attributes related to launching and running a Qt application, a *widget* that shows the latest prediction result and score in some manner, e.g. updating a textual label or a graphical plot. A timer object repeatedly calls `process_queue_items()` to check if `input_queue` has new items, and update the textual label or plot in case there are new results.

RealtimeAudio and RealtimeInference Usage Example

It is useful to illustrate the intended way to use `RealtimeAudio` and `RealtimeInference` classes, so a practical example is provided here. The following code block shows how to configure the two aforementioned classes with `multiprocessing` queues, and predicting once based on a random input array processed through the queue:

```
import multiprocessing as mp
import numpy as np
import time
from audio import RealtimeAudio
from model import RealtimeInference, init_audio_parameters

if __name__ == "__main__":
    cfg = model.init_audio_parameters()
    onset_time = time.time()
    queue_blocks = mp.Queue()
    rta = RealtimeAudio(cfg)
    rti = RealtimeInference(cfg)
    rta.output_queue = queue_blocks
    rti.input_queue = queue_blocks
    rnd_input = np.random.rand(1, cfg.fixed_length_block_samples)
    rta.output_queue.put((rnd_input, onset_time))
    copy_of_rnd_input, copy_of_onset_time = rta.input_queue.get()
    label, score, top_i = rti.predict(copy_of_rnd_input)
    pred_duration = time.time() - copy_of_onset_time
```

The above code block does the following:

1. compute an synthetic segment onset time by invoking `time.time()`
2. instantiate real-time audio and inference to `rta` and `rti` variables;
3. pass on queue reference to the appropriate attributes of `rta` and `rti` instances;
4. put random `numpy` array into the queue;
5. fetch a copy of the same array from the queue;
6. predict using the copied random input array;
7. compute the time duration required for producing an output prediction. It is useful to collect all segment-wise durations and later utilise them for comparing the inference speeds of different real-time SED system implementations.

If the above code would be converted into its real-time counterpart, some steps will be different. First, `rta.start()` would be invoked in order to activate the audio stream. Consequently, the active audio stream would begin repeatedly triggering `rta.callback` method, and the callback implementation in turn would process and put each new input audio block into the queue via `rta.output_queue.put(block)`. Furthermore, `rti` instance would be executed in a different thread or child process compared to `rta`.

Audio Stream Callback Details

The callback method associated with the launched `sounddevice.InputStream` instance is a very central entity belonging to the `RealtimeAudio` class. The block size of the audio input stream is configured to supply input data `numpy` arrays of length N_{block} for the stream-invoked callback function as an function input argument – note that the input data variable is named `indata` by convention. In this work, assume that the audio block length N_{block} is equal to the fixed-length segment length N_{seg} and the currently invoked callback performs the following processing steps:

1. Compute the segment onset time at the beginning of callback execution by invoking `time.time()` function call;
2. resolve a frame-blocking discontinuity between adjacent fixed-length segments (in practice, between adjacent `indata` arrays). This issue emerges, when overlapping frame-blocks are being eventually extracted from the `indata` array, and the samples of a certain frame-block exist between two adjacent fixed-length segments. That is, $H - S$ audio samples exist in the previously invoked `indata`, and the remaining S samples exist in the current one. This problem is illustrated in Figure 4.2. Ignoring this issue may lead to mismatches between real-time and offline SED prediction results depending on how the frame-blocking scheme supplied with the inferencing model has been implemented;
3. concatenate $H - S$ last samples from the `indata` array of the previous callback invocation into the current one, and discard $H - S$ samples from the end of the currently invoked callback `indata` array. This new temporally delayed array is designated as the current fixed-length segment that will be intermediately sent to the

`RealtimeInference` instance for prediction purposes;

4. the audio input block `indata` should be rolled into the long audio buffer `buffer_long` array of the `RealtimeAudio` class instance. The buffer array will be populated with all the fixed-length blocks intended to be collected during the real-time SED session (the buffer essentially holds the full input signal). The contents of the buffer can be given as an input to the `predict_signal` method of the `RealtimeInference` class instance, in order to evaluate the real-time results against these acquired offline counterparts;
5. pack the segment onset time, and the newly acquired current fixed-length segment into a Python tuple, and then intermediately send that tuple with `output_queue.put` to the `RealtimeInference` instance that will predict with the data and also update its `onset_time` attribute.

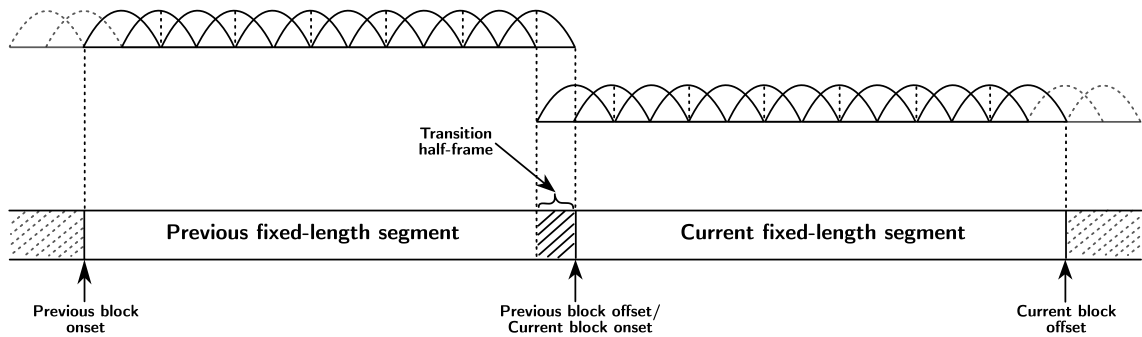


Figure 4.2. *The discontinuity of a single frame-block between adjacent fixed-length segments. In this example the overlap is 50%.*

In this work, the third step in the above list has been implemented as the `update_block` class method of `RealtimeAudio` class, which manipulates the contents of `buffer_block` array that is a member attribute of the `RealtimeAudio` class. The length of this buffer is two times the fixed-length segment length, corresponding to the shape $\mathbb{R}^{2 \cdot N_{seg} \times N_{channels}}$, where $N_{channels} = 1$. It is implemented as follows:

```
def update_block(self, indata):
    fixed_segment_len = self.cfg.fixed_length_block_samples
    self.buffer_block = np.roll(self.buffer_block, -fixed_segment_len,
                                axis=0)
    self.buffer_block[-fixed_segment_len:, :] = indata
    start_pos = -fixed_segment_len - self.cfg.block_pad_samples
    end_pos = -self.cfg.block_pad_samples
    block = self.buffer_block[start_pos:end_pos]
    return block
```

The above algorithm first rolls the contents of `buffer_block` such that the latest fixed-length segment data is shifted backwards by N_{seg} samples. Then the contents of `indata` is written to its previous position at the end of the array. Then, the start and end position of the new fixed-length segment is indexed such that the beginning and end position indices are shifted left by $H - S$ samples. This new data is held by the `block` variable, which the method returns.

Communication Between the Basic Component Instances

Now that each basic component has been described individually, their usage can be analysed also collectively. Figure 4.3 shows a sequence diagram, where the instances of the basic components communicate with each other. All of the steps (numbered 1–11) correspond to everything discussed so far in this section.

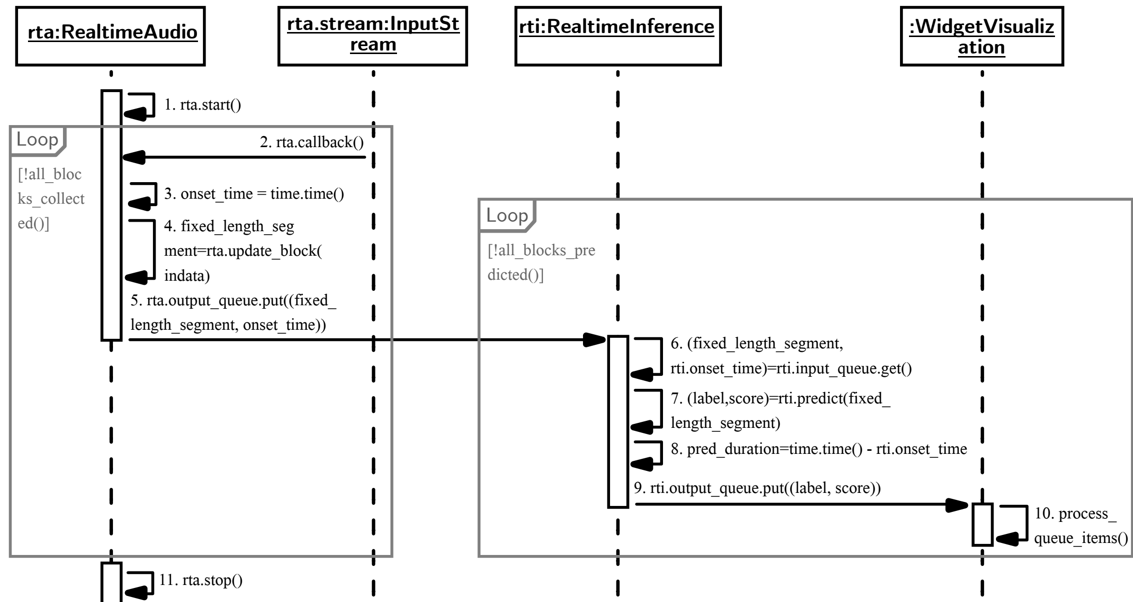


Figure 4.3. *Sequence diagram that describes the way in which basic system components communicate with each other.*

The first step activates the audio input stream with `start()` in order to allow the audio stream to invoke `rta.callback()` infinitely (step 2) until audio stream is stopped in step 11. The infinitely repeated callback invocations lead to steps 2–5 becoming a loop. The loop stop condition `all_blocks_collected()` returns `True`, when intended amount of fixed-length segments have been inputted by the active system, and `False` otherwise. In this work, `all_blocks_collected()` is assumed to be an attribute method of `RealtimeAudio` class, which checks the currently processed count of fixed-length segments.

A similar loop stopping mechanism is required also for `RealtimeInference` class, which it implements as the method known as `all_blocks_predicted()`. The greatest benefit is that the instance of `RealtimeInference` can be halted independently of `RealtimeAudio`, when it is being repeatedly invoked by a loop. The other loop on the right half of Figure 4.3 corresponds to `all_blocks_predicted()`, and causes the steps 6–10 to be repeated until the `all_blocks_predicted()` returns `True`. Step 6 actually blocks the execution until step 5 has concluded, not allowing transition from step 6 to steps 7–10 any earlier. The reason is that `input_queue.get()` calls block until a Python object is `put()` into the same queue.

The `process_queue_items()` method of the `WidgetVisualization` class instance in step 10 is being periodically triggered by a `QTimer` instance that is also a member of the same

class. The implementation of `process_queue_items()` method invokes `input_queue.empty()` in order to determine whether the `input_queue` is empty. If it is not empty, and `put()` in step 9 has been completed, `label` and `score` can be read with `get()` from the `input_queue` and `WidgetVisualization` will be invoked to update new results (e.g. into a plot or some other type of graphics canvas).

Summary of System Parameters

Since there exists several important system parameters that affect the intended behaviour of the real-time system designs, the most relevant ones of them are listed here separately for quick reference:

- Sampling rate: matches with the supplied model
- Number of audio channels ($N_{channels}$): 1
- Audio stream block size (N_{block}): set to the length of the fixed-length segment N_{seg}
- The dimensions of the audio block buffer `buffer_block` (N_{buf}): $\mathbb{R}^{2 \cdot N_{seg} \times N_{channels}}$
- Number of samples in a fixed-length segment: matches with the supplied model
- Frame-length (S): matches with the supplied model
- Hop size (H): matches with the supplied model
- Feature extraction scheme: chosen based on the supplied model, and omitted if embedded directly into the model
- Model input dimensions: $\mathbb{R}^{1 \times N_{seg}}$
- Model output dimensions: $\mathbb{R}^{1 \times N_{classes}}$
- Number of classes ($N_{classes}$): matches with the supplied model

The model input dimensions involve a simplification: it is assumed that feature extraction has been embedded into the model. That is, the model takes raw audio instead of features as an input.

4.1.3 Process-based Multiprocessing System

The first system design to be introduced in this work is the approach based on process-based multiprocessing, and its core idea is primarily motivated by the theory of Section 3.3.3. The approach is also mostly in accordance with the considerations of Section 4.1.1, although some minor compromises do exist. Figure 4.4 illustrates the individual components discussed in Section 4.1.2 as a part of the whole system. This free-form diagram does not describe object states or their transitions like the sequence diagram in Figure 4.3, but rather shows that to which Python processes the object instances should be distributed to.

Similar to the producer-consumer pattern of Section 3.3.3, the system runs two Python processes labelled tasks A and B in Figure 4.4. Task A is the entry process of the system,

and task B has been launched by task A. Furthermore, when task A quits, task B will also quit. The following instances of the basic real-time SED system objects of Section 4.1.2 are declared by task A:

- the classes `RealtimeAudio` and `UserInterface`;
- `multiprocessing.Queue` instances corresponding to the variables `queue_blocks` and `queue_predictions`.

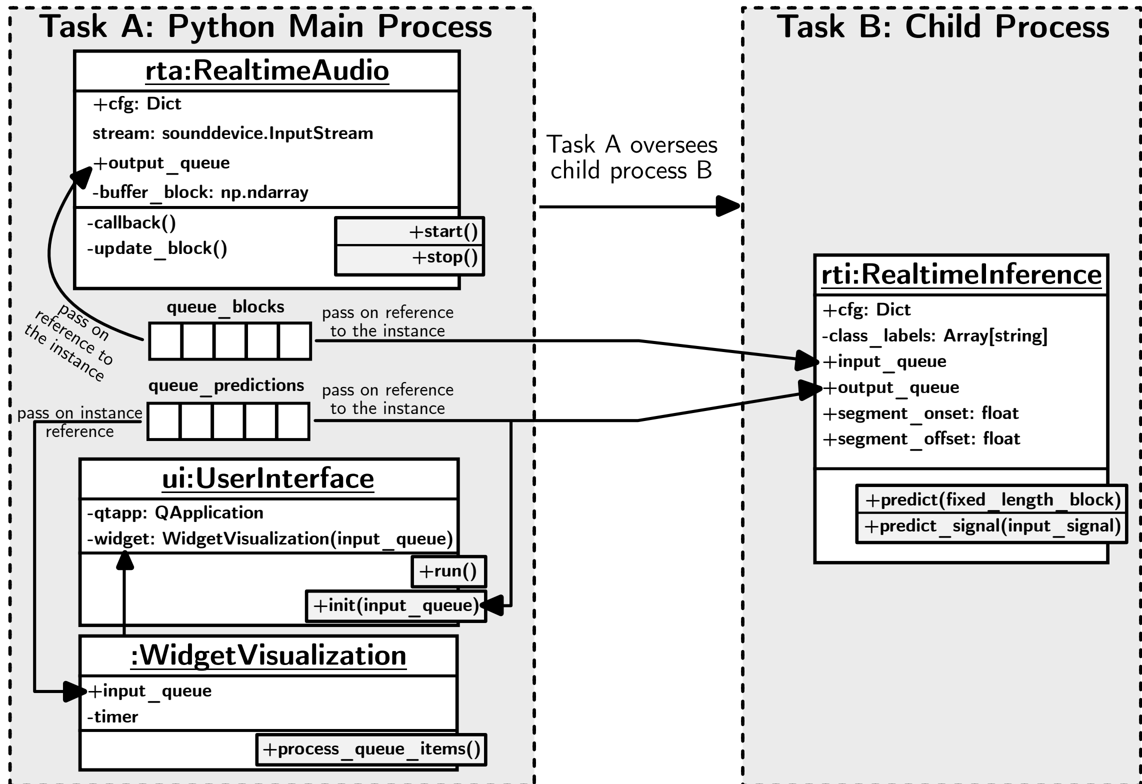


Figure 4.4. *A free-form diagram depicting the hierarchy of data in the process-based real-time SED system approach among the main and child processes.*

Task B is initialising only a single object: `RealtimeInference`. After all, it targets to only run the CPU-bound while task A consists primarily of I/O-bound tasks instead.

Several arrows are pointing out in Figure 4.4 from the two `multiprocessing.Queue` instances into the `input_queue` and `output_queue` member attributes of the four class instances. It visualises the intended sharing of queue references among involved objects that was discussed in detail in Section 4.1.2. The representation of Figure 4.4 brings in new insight by illustrating that the queues have been now configured to pass on Python objects across different processes. That is, in the manner they are meant to be utilised.

At this point, it is very simple to implement the system based on the basic system components. No changes are required to the communication between the basic system components shown in Figure 4.3, or to the most of the definitions of the components. Only the `RealtimeInference` class has to be extended with a supplied model. The system can be assembled using the following key steps based on Figure 4.4:

1. For implementing task B, define a Python function called `task_predictor` that declares and initialises variable `rti` as an instance of `RealTimeInference` class. Furthermore, `task_predictor` has a while loop with the end condition `all_blocks_predicted()`, waits for and receives new predictions by invoking `rti.input_queue.get()` and then predicts with `rti.predict()`;
2. Declare and initialise variables `queue_blocks` and `queue_predictions` as `multiprocessing.Queue()` instances within task A;
3. Declare and initialise a process with `multiprocessing.Process(target=task_predictor)` as an instance within task A. `Process` also has the `args` input parameter that can be utilised for passing on references to required queue instances. Now, start the child process;
4. Declare and initialise the rest of the basic system components designated for task A. Start the audio stream first, and then the user interface.

The above four steps can be implemented as follows:

```
import multiprocessing as mp
import time
from model import RealtimeInference
from audio import RealtimeAudio

def task_predictor(queue_blocks, queue_predictions):
   rti = RealtimeInference()
   rti.input_queue = queue_blocks
   rti.output_queue = queue_predictions
   while not rti.all_blocks_predicted():
      fixed_length_segment, rti.onset_time = rti.input_queue.get()
      label, score = rti.predict(fixed_length_segment)
      pred_duration = time.time() - rti.onset_time
      rti.output_queue.put((label, score))

if __name__ == "__main__":
   cfg = model.init_audio_parameters()
   q_blocks = mp.Queue()
   q_preds = mp.Queue()
   p = mp.Process(task=task_predictor, args=(q_blocks, q_preds))
   p.daemon = True
   p.start()  # launch the child/worker process
   rta = RealtimeAudio(cfg)
   rta.output_queue = q_blocks
   rta.start()
   ui = UserInterface(q_preds)
   ui.run()  # blocks until run() finishes
   p.join()  # blocks until the instance p quits
   rta.stop()
```

As a result, a minimal process-based real-time SED system with Python has been written in less than 30 lines of code.

4.1.4 Thread-based System

Another real-time SED system design presented in this work is called the thread-based system. Unlike the process-based approach, it will not launch any additional processes besides the mandatory main process. Threads are launched instead of processes for the CPU-bound tasks, and this section briefly investigates the immediate design consequences of that choice.

Threads spawned within the same process share a mutual memory space, which allows them to closely cooperate on a task without the need to transfer data between processes. Although, synchronised access to data shared between threads still needs to be maintained, which the thread-based approach will achieve with the help of `threading.Queue()` instances. Figure 4.5 illustrates how the components of the thread-based system should be initialised.

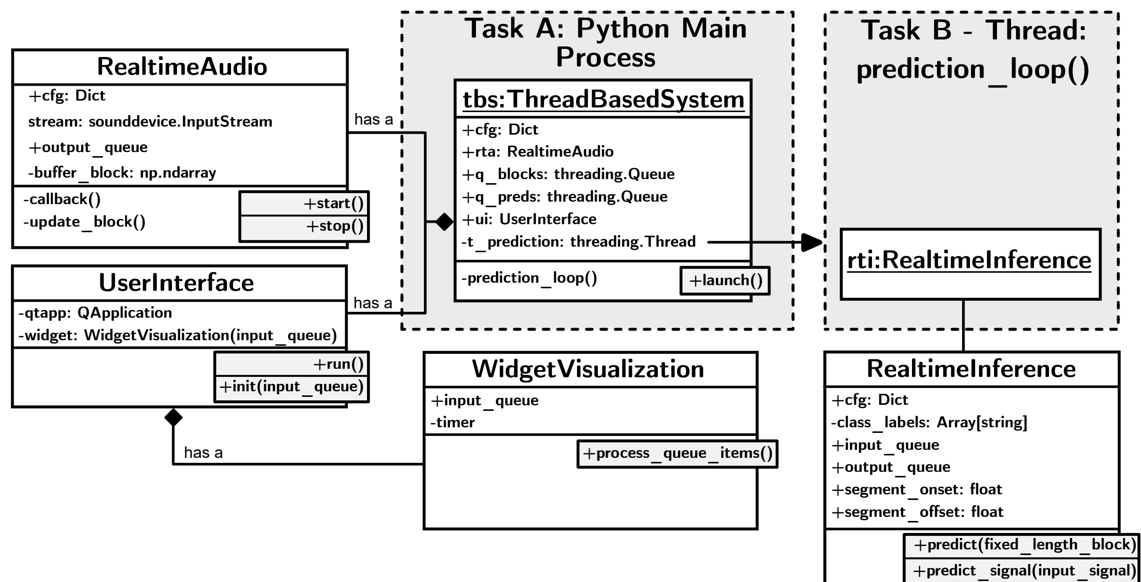


Figure 4.5. *A free-form diagram depicting the hierarchy of data in the thread-based real-time SED system approach among the main process and its worker thread.*

Besides the basic system components of Section 4.1.2, the diagram of Figure 4.5 introduces a new class called `ThreadBasedSystem` that is declared as the instance named `tbs`. The idea is that `ThreadBasedSystem` aggregates all of the basic system components under a single class along with objects that enable launching threads and communication between them.

Two major tasks can be identified based on Figure 4.5, which are tasks A and B. The instance `tbs` of `ThreadBasedSystem` is launched within task A and it initialises all basic system components except for the `RealtimeInference`. The instance of `RealtimeInference` is instead declared and initialised within task B, which represents a Python thread running the method `tbs.prediction_loop()`. The thread-based task B is in fact very similar to the process-based implementation of the same task, as they share exactly the same purpose of outputting predictions. The primary difference is the distinction between a

process and a thread, along with their benefits and disadvantages.

Similar to the process-based approach, the thread-based system design approach also leads to a very simple Python implementation:

```
from threading import Queue, Thread
import time
from model import RealtimeInference
from audio import RealtimeAudio

class ThreadBasedSystem:
   def __init__(self):
      self.q_blocks = Queue()
      self.q_preds = Queue()
      self.cfg = model.init_audio_parameters()
      self.rta = RealtimeAudio(cfg)
      self.rta.output_queue = self.q_blocks
      self.ui = UserInterface(self.q_preds)
      self.t_prediction = threading.Thread(target=self.prediction_loop)
      self.t_prediction.daemon = True

   def launch(self):
      self.t_prediction.start()
      self.ui.run()

   def prediction_loop(self):
      rti = RealtimeInference()
      rti.input_queue = self.q_blocks
      rti.output_queue = self.q_preds
      self.rta.start()
      while not rti.all_blocks_predicted() and self.rta.stream.active:
         fixed_length_segment, rti.onset_time = rti.input_queue.get()
         label, score = rti.predict(fixed_length_segment)
         pred_duration = time.time() - rti.onset_time
         rti.output_queue.put((label, score))

if __name__ == "__main__":
   tbs = ThreadBasedSystem()
   tbs.launch()
   tbs.rta.stop()
```

At least the following conclusions ca be made based on the code above:

- Virtually all of the implementation components are a part of the `ThreadBasedSystem` class definition, which is in accordance with the details of Figure 4.5;
- the `prediction_loop` method running via the launched thread is very similar to `task_predictor` function of the process-based system approach. A big difference is that `prediction_loop` has accessed class member attributes declared and initialised within another thread, since it is allowed. This would not be possible in the case of `task_predictor`, which is executed as a child process, requiring copies for the most of the conveyed data.

4.2 Graphical User Interface

Visualising the sound events outputted by the real-time SED system provides its users an engaging way to experience it in practice and assess its behaviour. In this work, Qt is the chosen GUI framework for plotting sound events, because it offers a powerful graphics canvas for fluid animations, is easy to learn and is well-established also in the Python community. Qt provides an extensive collection of various *widgets*, which are essentially individual components initialised into the user interface, where some of them are being static and some can be interacted with. There are two primary projects that enable Qt support in Python, which are the `pyqt5` and `pyside2` packages. The latter one is also known as *Qt for Python*, which is the officially supported Python bindings for Qt [42], and it is the only bindings considered in this work. Its licencing options include either the Qt commercial license or LGPLv3/GPLv3.

4.2.1 Qt Application Essentials

The first step in implementing a graphical `pyside2` Qt application is to create an instance of the `QApplication` object. Once that is done, the programmer can next move on to declaring the instances of various widget objects that morph the look and feel of the user interface. Widgets are typically also associated with some events that are triggered through user interaction with the widget in question. Qt maps these events to some pre-defined actions with the help of its *signal-slot* mechanism, e.g. the instance of `QPushButton` widget emits the `clicked()` signal when being activated. This activation in turn triggers the execution of the intended Python function definition (called a slot in Qt) that performs the desired actions. After configuring the widgets, assigning signals to slots or performing any other initialisations, the instance of `QApplication` will call its `run()` method to launch the Qt event loop, which schedules the execution of all events within a single loop at the main thread of execution. The `run()` method blocks the thread until user triggers or signals the user interface to quit, and that concludes the lifecycle of a Qt application. [43]

4.2.2 Visualizations

An explicit widget for visualising sound events was not found during the development of the system implementations, so various Python plotting libraries that allow declaring the plots as Qt widgets were experimented with. Initially, `PyQtGraph` is the first plotting library being tested, and it turned out to be very fast, but unstable: during long-running SED system sessions, there were random crashes related directly to some issues with `PyQtGraph`.

Due to this setback, `Chaco` package from Enthought was tested next and it did provide the desired stability for the long-running real-time SED sessions. The most suitable plotting representation found in the case of `Chaco` was a bar plot showing the history of recent

predictions. The horizontal axis in the plot was configured to represent the adjacent time frames for the predicted fixed-length blocks, and height of each bar in the vertical axis representing the confidence of a prediction. The oldest bar in the plot is discarded as soon as a new fixed-length segment prediction becomes available. For interpretability, the class label string representations were shown just below each bar in order to give an indication about which sound categories are the most recent ones.

The third and final visualization experiment involved creating an explicit Qt widget using the well-performing Qt graphics canvas for plotting. Essentially, one can utilise the `QPainter` object for drawing e.g. lines and rectangles within the allocated screen space of the visualization widget. The implemented widget displays a chosen amount of latest sound events such that they slide from the right side of the plot to the left: it is a constantly updating event roll for displaying constantly predicted sound events. It is straightforward to implement it as a matrix, where columns represent a fixed-length segment time frame, rows denote the category of the sound event class, and elements have values `False` (not detected) or `True` (detected). Every time the real-time SED system outputs a new prediction, the contents of the matrix are refreshed in a manner similar to a multidimensional buffer roll: the elements of the first column are set to `False` (oldest time frame), then rolling that column to become the new final column, and setting the row element of that column matching the correct category for a new prediction to `True`.

5 EVALUATION

This chapter evaluates the validity of the process and thread-based system designs of Chapter 4. Specifically, a set of experiments are performed to verify whether utilising those designs leads to implementing a fully functional real-time SED system with Python.

5.1 Method

In order to conduct the desired experiments, systems based on the thread and process-based design approaches were implemented explicitly for the evaluation purposes of this chapter. These two implementations behave exactly like the thread and process-based designs of Chapter 4, and only minimally extend their capabilities so that results can be stored and offline inference can be performed after the conclusion of a real-time SED session. Since model training is not part of this work, the evaluation systems utilise a publicly available model for sound classification. The basic idea behind the evaluation procedure of an implemented system is simple: the output results of the real-time system implementation is compared to its offline counterpart and then it is determined whether the real-time and offline output results match. Figure 5.1 illustrates the whole evaluation scheme, which has been primarily derived from the real-time and offline usage stages of Figures 2.1 and 3.1. The steps are the following:

1. perform the real-time usage stage;
2. perform the counterpart offline usage stage after the real-time usage stage has concluded;
3. evaluate whether the real-time and offline stages produced matching prediction results.

In other words, the scheme of Figure 5.1 executes two signal processing pipelines separately: the real-time and offline usage stages. These two stages share the same input signal and are also expected to produce the same output if the pipelines have been correctly implemented.

In the original offline usage stage of Figure 2.1, an audio input signal was read from a file into an array that shall hold the whole signal. The scheme of Figure 5.1 instead skips the file reading part and starts processing input signal from a provided array directly. For this purpose, the real-time usage stage of Figure 5.1 accumulates the audio samples involved in classifying a selected amount of segments into a long audio buffer

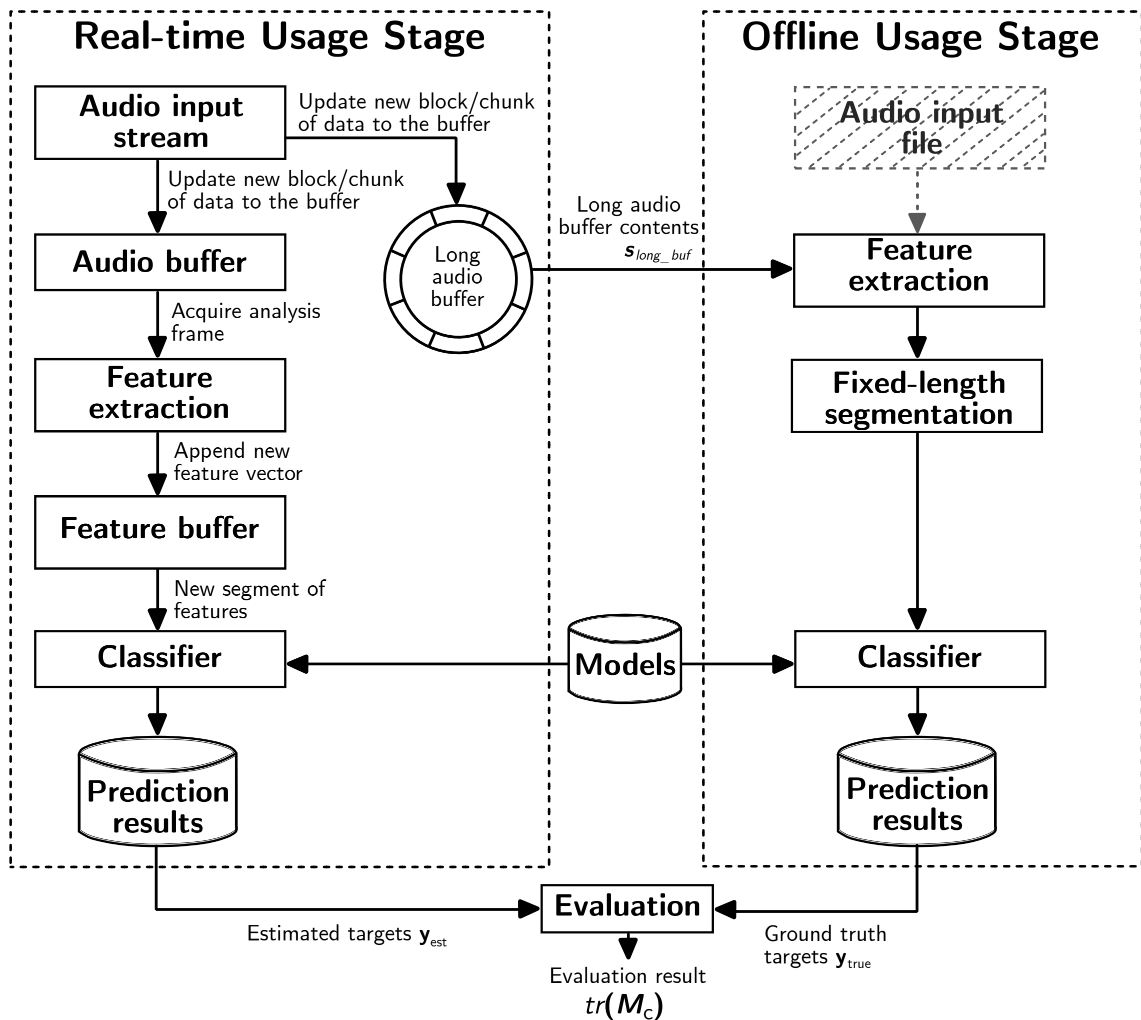


Figure 5.1. *The evaluation scheme of a real-time SED system against its counterpart offline system. Real-time usage stage is executed first and it is then followed by the offline usage stage. After obtaining prediction results from both stages, they can be validated using a pre-determined evaluation procedure.*

while simultaneously also performing real-time sound classification. After concluding the real-time usage stage, all audio samples used in classifying the selected segments are hold in the long audio buffer and the contents can be passed on to the offline usage stage as an input array. If the array that was passed on to the offline usage stage is free of any distortions, delays or samples drops and the offline classification behaves the same way as in the real-time usage stage and uses the same model, prediction results identical to the real-time usage stage are to be expected.

Performance Measure

The third step in the scheme of Figure 5.1 evaluates whether prediction results produced by the real-time usage stage perfectly match their offline counterparts. A performance measure applicable for classification problems will be utilised to verify this, since this work is an application of sound classification. In order to utilise any of the applicable

performance measures, the prediction results of the real-time and offline usage stage will be represented as *ground truth target values* and *estimated target values*. This work treats the prediction results $\boldsymbol{y}_{true}$ associated with the offline usage stage as the ground truth, and correspondingly the prediction results $\boldsymbol{y}_{est}$ of the real-time usage stage as their estimated targets.

The chosen performance measure for evaluation is the *confusion matrix*

$$\boldsymbol{M}_c \in \mathbb{Z}_{\geq 0}{}^{N_{classes} \times N_{classes}} \tag{5.1}$$

for a multi-class classification problem with $N_{classes}$ sound categories, and it can be computed with the help of the ground truth $\boldsymbol{y}_{true}$ and their estimated target values $\boldsymbol{y}_{est}$. Only the main diagonal elements containing the class-wise true positive values in the confusion matrix need to be involved in the evaluation, because the trace of the confusion matrix, $tr(\boldsymbol{M}_c)$, represents the total number of true positive values [44, p. 871]. Therefore, when the acquired total number of true positives $tr(\boldsymbol{M}_c)$ matches the number of outputted predictions, the evaluation results produced by the real-time and offline usage stages match each other.

5.2 Pre-trained Model

The training stage has been completely excluded in the evaluation scheme of Figure 5.1 due to the public availability of pre-trained SED models, which consequently allows fully focusing on developing the thread and process-based evaluation systems. Furthermore, only some of the architectural details of the model are relevant in succeeding in this part of the work, since the main objective is simply to evaluate the validity of the system designs. That is, the focus is on presenting the mainly model details that contribute to successfully utilising the model in the evaluation system implementations.

The publicly available model chosen for the evaluation system implementations is a multi-class sound classification model called *YAMNet* [45], which is based on the *MobileNet V1* [46] depth-wise separable convolution architecture. [45] has trained YAMNet against the*Audio Set* corpus [47] with TensorFlow, the model possesses approximately 3.7M total parameters and it predicts $N_{classes} = 521$ sound event classes. Table 5.1 shows a summary of the YAMNet training parameters.

Feature extraction has been embedded into YAMNet such that the model first takes a raw audio input

$$\boldsymbol{A} \in \mathrm{I\!R}^{1 \times N_{inp}} \tag{5.2}$$

as a TensorFlow input tensor (represented in vector space), and in a sequence of various tensor operations transforms it into log mel spectrogram tensor output

$$\boldsymbol{B} \in \mathrm{I\!R}^{N_{stft} \times N_{mel}}, \tag{5.3}$$

where N_{inp} is chosen as the number of audio input samples, N_{stft} is the number of short-time Fourier transform (STFT) frames, and N_{mel} is the number of mel bands. Then, the log mel spectrogram $\boldsymbol{B}$ is used as an input tensor for extracting $N_{patches}$ quantity of spectrogram patches

$$\boldsymbol{C} \in \mathbb{R}^{N_{patches} \times N_{stft} \times N_{mel}}. \tag{5.4}$$

Now, $\boldsymbol{C}$ is used as a TensorFlow input tensor for the core YAMNet model that will output prediction scores

$$\boldsymbol{D} \in \mathbb{R}^{N_{patches} \times N_{classes}} \tag{5.5}$$

for $N_{patches}$ amount of patches.

Table 5.1. *YAMNet Training Parameters.*

Parameter	**Value**	**Symbol**
Sample rate	16 kHz	–
Spectrogram patch window length	0.96 s (15360 samples)	N_{patch_wlen}
Spectrogram patch hop size	0.48 s (7680 samples)	–
Analysis frame length	0.025 s (400 samples)	–
Analysis frame hop size	0.010 s (160 samples)	–
Number of Mel bands	64	N_{mel}
Mel filter bank lower frequency bound	125 Hz	–
Mel filter bank upper frequency bound	7500 Hz	–
Number of classes	521	$N_{classes}$

A noteworthy consequence of embedding the features into the model is that separately implementing feature extraction and feature buffering for the evaluation systems is not required. Instead, implementing frame-block buffering for the real-time evaluation systems is already sufficient in order to use YAMNet.

It is not self-evident that outputting scores with YAMNet both in offline and real-time manner produces exactly the same output scores based on an identical input, because both workflows need to be carefully crafted to operate the input exactly the same way. First, in order to make the experiments of this chapter easier to implement, frame-blocking of the YAMNet spectrogram patches will be simplified such that they become non-overlapping. Solving this for the offline usage stage is as simple as setting the spectrogram patch hop size of Table 5.1 to 0.96 seconds, which the model will configured to utilise in this context.

It is somewhat trickier to perform the frame-blocking of spectrogram patches in real-time. One needs to implement a suitable buffering scheme that updates a fixed-length segment based on a block from the audio input stream. The minimum input length N_{seg} YAMNet accepts is 0.975 seconds worth of samples instead of 0.96 seconds, because the start index of the final STFT analysis frame begins at 0.95 and stops at 0.975 seconds – the model basically expects zero-padding at the end of the signal. The real-time evaluation system implementations utilise a zero-initialised audio input buffer that is two times the duration of a 0.975 seconds fixed-length segment, because samples from both segments

are needed for updating to the latest one. The portion of the data to be set as the latest fixed-length segment overlaps by 0.015 seconds with end samples of the previous one in order to preserve the (likely non-zero) "padded" samples after each update (except the first time, the padded samples are in fact zeros). This treatment of adjacent fixed-length segments relates to the discontinuity problem of Figure 4.2, and has been solved based on it.

5.3 Evaluation Scheme with YAMNet

Since YAMNet model internally implements several operations belonging to the evaluation scheme of Figure 5.1, it makes sense to simplify the scheme in order to more accurately represent the YAMNet-specific evaluation procedure. Figure 5.2 illustrates the simplified scheme.

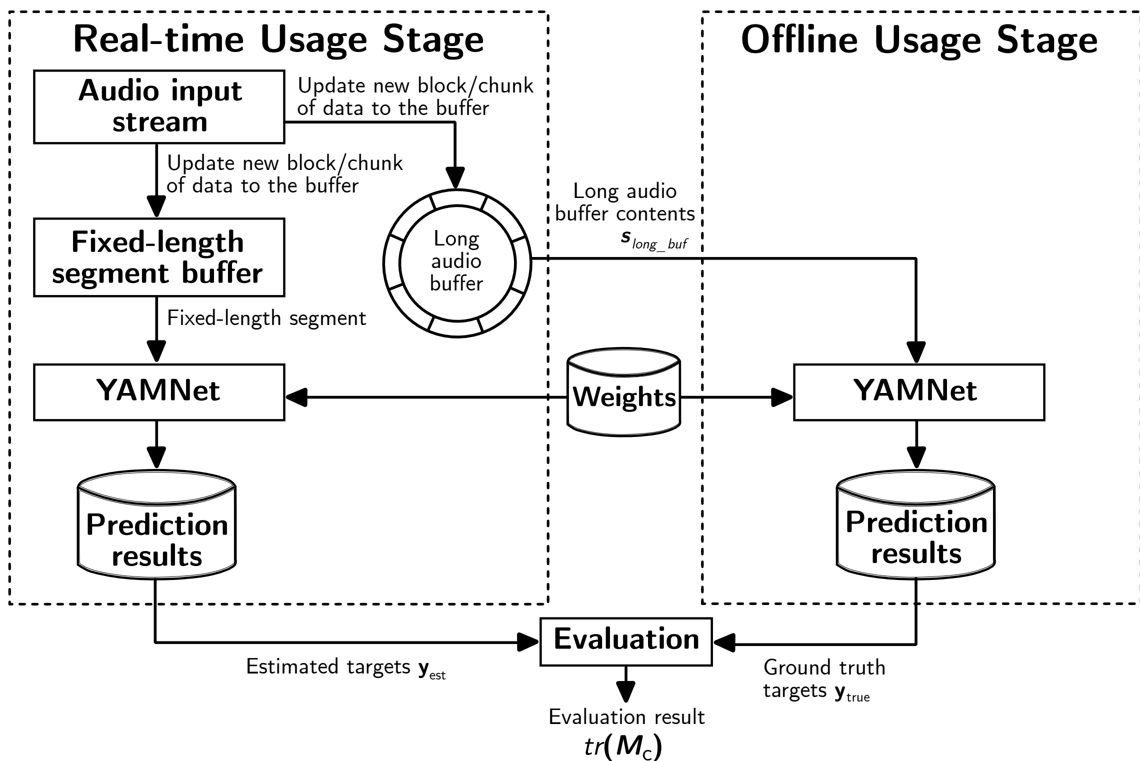


Figure 5.2. *The simplified evaluation scheme of a real-time SED system against its counterpart offline system demonstrating YAMNet as the sound classifier.*

The main differences between Figure 5.2 and Figure 5.1 are that some processing steps from the previous scheme have been integrated to YAMNet, and the audio buffer component has been relabelled as the fixed-length segment buffer instead. The difference between these buffers is merely their length, the fixed-length segment buffer corresponding to N_{seg} amount of samples as indicated by Table 5.2.

There are three final details regarding Figure 5.2 that have not been discussed yet. The first one being the total length N_{long_buf} of the long audio buffer, which is defined as a positive integer number multiple of block size N_{block} that corresponds to the audio input

Table 5.2. *Evaluation System Audio Streaming Parameters.*

Parameter	Value	Symbol
Audio input stream sample rate	16 kHz	–
Audio input stream block size	0.975 s (15600 samples)	N_{block}
Fixed-length segment buffer length	1.95 s (31200 samples)	N_{buf}
Fixed-length segment length	0.975 s (15600 samples)	N_{seg}

stream. That is,

$$N_{long_buf} = z \cdot N_{block}, \tag{5.6}$$

where $z \in \mathbb{Z}_{>0}$ is the multiple number. It makes sense to define N_{long_buf} in this manner, because the long audio buffer is being repeatedly updated with blocks incoming from the active audio input stream. Therefore, before executing an evaluation experiment using one of the evaluation systems, the number of blocks to collect has to be pre-determined (initialize z), so that the long audio buffer can be set to the correct length.

The second detail is simply about terminology. The models loaded by the sound classifier have been relabelled as weights in Figure 5.2. In this work weights are simply thought to better reflect the intended TensorFlow terminology.

Finally, the third detail defines the way the scores outputted by YAMNet are represented as the estimated target vector $\boldsymbol{y}_{est}$ based on the real-time output, or as the ground truth target vector $\boldsymbol{y}_{true}$ based on the offline output. The real-time instance of YAMNet outputs a single probability score vector $\boldsymbol{D} \in \mathrm{I\!R}^{1 \times N_{classes}}$ that is based on a given fixed-length segment input $\boldsymbol{A} \in \mathrm{I\!R}^{1 \times N_{seg}}$. Assume that z amount of fixed-length segments are collected during the evaluation session, then there exists a tuple of scores $(\boldsymbol{D}_1, \ldots, \boldsymbol{D}_z)$ at the end of the session. For each tuple item, class label index matching the highest score is extracted, which corresponds to another tuple $(arg\ max(\boldsymbol{D}_1), \ldots, arg\ max(\boldsymbol{D}_z))$. This will be represented as the YAMNet output score vector

$$\boldsymbol{y}_{est} = \begin{bmatrix} y_{est,1} \\ \vdots \\ y_{est,z} \end{bmatrix}, \tag{5.7}$$

where each element correspond to the estimated target value with the highest score.

The offline evaluation procedure for acquiring $\boldsymbol{y}_{true}$ is similar to the above except that YAMNet is given the long audio buffer contents as input, which corresponds to $\boldsymbol{A} \in \mathrm{I\!R}^{1 \times N_{long_buf}}$. Consequently, YAMNet will output z scores $\boldsymbol{D} \in \mathrm{I\!R}^{z \times N_{classes}}$ and the function $arg\ max$ is applied to each row of $\boldsymbol{D}$ in order to obtain z quantity of highest score ground

truth targets

$$\boldsymbol{y}_{true} = \begin{bmatrix} y_{true,1} \\ \vdots \\ y_{true,z} \end{bmatrix}. \tag{5.8}$$

5.4 Setup

The evaluation setup consists a single laptop computer with 16 GB RAM and Intel Core i7 8650U CPU. The computer is using Windows 10 operating system and the Python development environment has been configured with Anaconda. The computer does not possess a discrete GPU, so TensorFlow has been configured only to leverage the CPU. Internal microphone of the computer is utilised as an input device for the real-time system implementations.

5.5 Results

This section reports the evaluation results acquired using the thread and process-based evaluation system implementations. The systems are executed using the computer setup described in Section 5.4. The first experiment verifies the validity of the outputted observations for both system implementations in Section 5.5.1. The second experiment conducts a statistical significance test that determines whether the process-based approach performs computations faster compared to its thread-based counterpart.

5.5.1 Real-time System Output Validity Evaluation

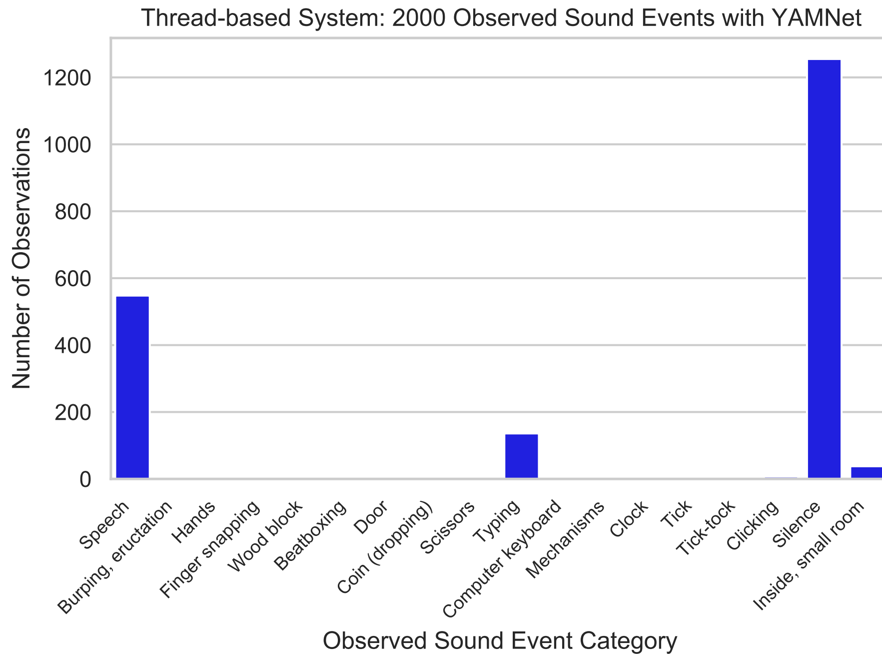
The summary of results for the first experiment is shown in Table 5.3, where both systems have been configured to output exactly 2000 prediction labels. The third column of Table 5.3 reports 2000 as the total number of true positives for both systems, so the implementations are indeed capable of producing intended results based on the proposed system designs of Chapter 4. Furthermore, neither of the systems demanded high CPU loads from the evaluation setup computer, based on the fourth column of Table 5.3.

Table 5.3. *Results of the real-time evaluation system output validity tests.*

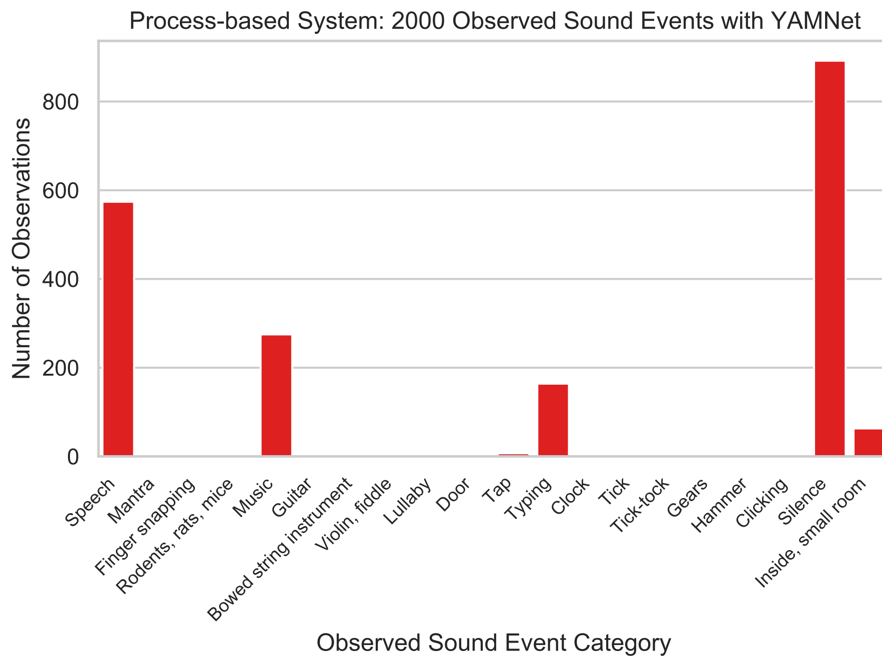
Sample ID	**System**	**True Positives** $tr(\boldsymbol{M}_c)$	**System CPU Load-% (mean ± std.)**	**Segment Inference Time Duration (mean ± std.)**
1	Thread-based	**2000**	6.9 ± 4.6%	49.9 ± 17.6 ms
2	Process-based	**2000**	16.8 ± 5.1%	39.6 ± 9.2 ms

It would be challenging to visualize the samples of Table 5.3 by utilising full confusion matrix $\boldsymbol{M}_c$ due to the high number of sound categories N_c. Instead, the non-zero main

diagonal elements of a matrix M_c will be displayed on a barplot – that is, counts of observations for each sound category in a sample. The barplots a) and b) in Figure 5.3 illustrate the observed categories of sounds for the samples of Table 5.3. Significant amount of silence was observed in both samples, and also a moderate amounts of speech, keyboard typing and music. The presence of other observed sound categories is neligible as indicated by the indistinguishable bar heights.



***(a)** Barplot corresponding to the thread-based system (sample id. 1)*



***(b)** Barplot corresponding to the process-based system (sample id. 2)*

***Figure 5.3.** Barplot representations of the samples of Table 5.3*

The fifth column in Table 5.3 reports the mean and standard deviation based on inference time durations of 2000 fixed-length segments. Each duration associated with a fixed-length segment is the computed time difference between the segment onset time, and the time of outputting the prediction. The times are estimated by invoking `time.time()` Python routine exactly as described in Sections 4.1.2, 4.1.3 and 4.1.4.

5.5.2 Estimating Inference Speed

Comparing the CPU loads between the samples of Table 5.3 suggests a possibility that the process-based system could have an increased access to the CPU compared to its thread-based counterpart, since the average CPU load is over two times higher compared to the sample of the thread-based system. This could also happen due to changes in external circumstances, e.g. the baseline CPU load of the OS being increased by other running processes. Therefore, CPU load is not a reliable way to compare the computational performance of the system implementations. Instead, a more reliable conclusion regarding the inference speeds can be drawn based on the estimated segment inference time durations reported in Table 5.3.

In order to determine that which one of the two evaluation system approaches performs model inferencing quicker using the described setup of Section 5.4, a simple hypothesis test will be performed. The two inference time duration samples of Table 5.3 are assumed independent of each other, one-tailed Welch's t-test is used as the test statistic with $(1999 - 1) + (1999 - 1) = 3998$ degrees of freedom, signifigance level is $\alpha = 0.05$, and the following hypothesis is formulated:

H_0: $\mu_1 = \mu_2$

H_a: $\mu_1 > \mu_2$

The null hypothesis H_0 states that there is no significant difference between the means of the two segment inference time duration samples of size 2000, and the alternative hypothesis H_a instead states that the mean of the thread-based system inference time duration sample μ_1 is greater compared to the process-based system sample with mean μ_2. The test statistic $stat$ and corresponding p-value p_{val} is computed using `ttest_ind` function from the `scipy.stats` package as below:

```
>>> print(f"Thread-based sample mean={np.mean(sample1)}, std={np.std(sample1)}, size={len(sample1)}")
Thread-based sample mean=0.04989682197570801, std=0.017595910833827832, size=2000
>>> print(f"Thread-based sample mean={np.mean(sample2)}, std={np.std(sample2)}, size={len(sample2)}")
Thread-based sample mean=0.03955512678623199, std=0.00921307831015021, size=2000
>>> ttest_ind(sample1, sample2, axis=0, equal_var=False)
Ttest_indResult(statistic=23.279644200457305, pvalue=2.176572983366366e-110)
```

Based on the above analysis, the null hypothesis H_0 is rejected due to $p_{val}/2 < \alpha$. Consequently, the average segment inference time μ_1 is greater compared to the average μ_2, with 95% certainty. In conclusion, circumstances exist, where the process-based evaluation system implementation achieves overall lower inference times over its counterpart.

6 CONCLUSION

In this work, two Python real-time SED system prototypes based on slightly different approaches were designed and implemented. In both cases, the implementations of the prototypes succeeded in transforming the offline system into its online counterpart that also allows displaying live prediction results through a Qt GUI visualizer. Both of these prototypes can be considered as being potential implementation approaches, but experiments conducted in Section 5.5.2 suggest that the process-based approach performs inference faster compared to the thread-based approach. However, this conclusion about the inference speed can be generalised only for the evaluation setup of Section 5.4 and with the system implementations of this work.

A non-technical way to measure the success of the implemented prototypes is the capability to design, implement and deliver them on schedule, since research projects are fast-paced. There was no previous experience in transforming off-line SED system into its real-time counterpart, so initially studying the relevant literature was required. Furthermore, it was learned that sometimes refactoring the provided feature extraction and classifier implementations can take unexpected amount of project time. In particular, when models are being updated very frequently or new machine learning libraries are being used. The worst case scenario is that potentially all of the provided off-line SED system Python code is undocumented and has to be fully refactored in order to be used for the real-time counterpart. Fortunately, the worst-case scenario did not occur and the project time spent for refactoring remained acceptable. At one point, the whole direction of the research actually changed, which somewhat altered the real-time system requirements. Despite this unexpected development, the system was still delivered on due time.

It was not initially clear how well Python scales up to the task of prototyping real-time SED systems. This work ascertains that it is possible, and even viable, to implement such systems with Python. As a consequence, estimating the overall difficulty level and choosing the right implementation approaches become clear.

For future work, at least more comprehensive evaluation should be performed, e.g. investigating the validity of the inputted raw audio of the fixed-length segments before offline evaluation. It should be also illustrated how to utilise the `Event` object from `multiprocessing` and `threading` packages for waiting a condition to be triggered within another thread or process, which then permits the thread in question to continue execution. In this work, this is useful when waiting for e.g. the GUI or model to become fully initialised before starting the audio stream.

REFERENCES

[1] T. Heittola, A. Mesaros, T. Virtanen and M. Gabbouj. Supervised Model Training for Overlapping Sound Events Based On Unsupervised Source Separation. *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. 2013, 8677–8681. DOI: 10.1109/ICASSP.2013.6639360.

[2] M. Shah, B. Mears, C. Chakrabarti and A. Spanias. Lifelogging: Archival and Retrieval of Continuously Recorded Audio Using Wearable Devices. *2012 IEEE International Conference on Emerging Signal Processing Applications*, 99–102. DOI: 10.1109/ESPA.2012.6152455.

[3] A. Härmä, M. McKinney and J. Skowronek. Automatic Surveillance of the Acoustic Activity In Our Living Environment. *2005 IEEE International Conference on Multimedia and Expo*. Vol. 1. 2005, 634–637. DOI: 10.1109/ICME.2005.1521503.

[4] Y.-T. Peng, C.-Y. Lin, M.-T. Sun and K.-C. Tsai. Healthcare Audio Event Classification using Hidden Markov Models and Hierarchical Hidden Markov Models. *2009 IEEE International Conference on Multimedia and Expo*. 2009, 1218–1221. DOI: 10.1109/ICME.2009.5202720.

[5] A. Eronen, V. Peltonen, J. Tuomi, A. Klapuri, S. Fagerlund, T. Sorsa, G. Lorho and J. Huopaniemi. Audio-based Context Recognition. *IEEE Transactions on Audio, Speech and Language Processing* 14.1 (2006), 321–329. DOI: 10.1109/TSA.2005.854103.

[6] T. Virtanen, M. Plumbley and D. Ellis. *Computational Analysis of Sound Scenes and Events*. Gewerbestrasse 11, 6330 Cham, Switzerland: Springer International Publishing AG, 2018, 1–422. DOI: 10.1007/978-3-319-63450-0.

[7] A. Tanenbaum. *Modern Operating Systems*. Upper Saddle River, N.J USA: Pearson Prentice-Hall, 2008. ISBN: 978-0-13-359162-0.

[8] A. Williams. *C++ Concurrency In Action*. 20 Baldwin Road, PO Box 261, Shelter Island, NY 11964 USA: Manning Publications Co., 2012. ISBN: 978-1933988771.

[9] D. Beazley. *Python Essential Reference*. Upper Saddle River, N.J USA: Addison-Wesley Professional, 2009. ISBN: 978-0-672-32978-4.

[10] S. McConnell. *Code Complete*. 2nd Edition. One Microsoft Way Redmond, WA 98052-6399 USA: Microsoft Press, 2004. ISBN: 978-0-7356-1967-8.

[11] F. Pedregosa et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[12] F. Chollet. *Keras*. https://github.com/fchollet/keras. 2015.

[13] P. Virtanen et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. DOI: 10.1038/s41592-019-0686-2.

[14] Y. LeCun, Y. Bengio and G. Hinton. Deep Learning. *Nature* 521.7553 (2015), 436–444. DOI: 10.1038/nature14539.

[15] S. Haykin. *Neural Networks: a Comprehensive Foundation*. 2nd Edition. Upper Saddle River, N.J USA: Prentice-Hall, 1999. ISBN: 0-13-273350-1.

[16] R. O. Duda, P. E. Hart and D. G. Stork. *Pattern Classification*. 2nd Edition. 605 Third Avenue, N.Y USA: John Wiley & Sons, Inc., 2001. ISBN: 978-0-471-05669-0.

[17] C. M. Bishop. *Pattern Recognition and Machine Learning*. 233 Spring Street, New York, NY 10013, USA: Springer Science+Business Media, LLC, 2006. ISBN: 978-0387-31073-2.

[18] T. M. Cover. Geometrical and Statistical Properties of Systems of Linear Inequalities With Applications in Pattern Recognition. *IEEE Transactions On Electronic Computers* 3.EC-14 (1965), 326–334. DOI: 10.1109/PGEC.1965.264137.

[19] X. Huang, A. Acero, H.-W. Hon and R. Foreword By-Reddy. *Spoken Language Processing: A Guide to Theory, Algorithm, and System Development*. Upper Saddle River, N.J USA: Prentice-Hall PTR, 2001. ISBN: 978-0-13-022616-7.

[20] L. Rabiner and B.-H. Juang. *Fundamentals of Speech Recognition*. A Simon & Schuster Company, Englewood Cliffs, NJ 07632 USA: PTR Prentice-Hall, Inc., 1993. ISBN: 0-13-285826-6.

[21] E. C. Ifeachor and B. W. Jervis. *Digital Signal Processing: A Practical Approach*. 2nd Edition. Copyright Licensing Agency Ltd, 90 Tottenham Court Road, London W1P 0LP, UK: Pearson Education Limited, 2002. ISBN: 0-201-59619-9.

[22] G. James. *Advanced Modern Engineering Mathematics*. 3rd Edition. Copyright Licensing Agency Ltd, 90 Tottenham Court Road, London W1T 4LP, UK: Pearson Education Limited, 2004. ISBN: 0-13-045425-7.

[23] S. B. Davis and P. Mermelstein. Comparison of Parametric Representations for Monosyllabic Word Recognition in Continuously Spoken Sentences. *IEEE Transactions on Acoustics, Speech and Signal Processing* 28.4 (1980), 357–366. DOI: 10.1109/TASSP.1980.1163420.

[24] D. A. Reynolds and R. C. Rose. Robust Text-Independent Speaker Identification Using Gaussian Mixture Speaker Models. *IEEE Transactions on Speech and Audio Processing* 3.1 (1995), 72–83. DOI: 10.1109/89.365379.

[25] A. Mesaros, T. Heittola, A. Eronen and T. Virtanen. Acoustic Event Detection in Real Life Recordings. *2010 IEEE 18th European Signal Processing Conference*. 2010, 1267–1271.

[26] E. Cakir, T. Heittola, H. Huttunen and T. Virtanen. Polyphonic Sound Event Detection Using Multi Label Deep Neural Networks. *2015 IEEE International Joint Conference on Neural Networks*. 2015, 1–7. DOI: 10.1109/IJCNN.2015.7280624.

[27] V. Pulkki and M. Karjalainen. *Communication Acoustics: An Introduction to Speech, Audio and Psychoacoustics*. The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, UK: John Wiley & Sons, Ltd, 2015. ISBN: 978-1-118-86654-2.

[28] T. Virtanen, R. Singh and B. Raj. *Techniques for Noise Robustness in Automatic Speech Recognition*. The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, UK: John Wiley & Sons, Ltd, 2013. ISBN: 978-1-119-97088-0.

[29] S. Young et al. *The HTK Book*. Version 3.3. Trumpington St, Cambridge CB2 1PZ, UK: Cambridge University Engineering Department, 2004.

[30] E. Cakır, G. Parascandolo, T. Heittola, H. Huttunen and T. Virtanen. Convolutional Recurrent Neural Networks for Polyphonic Sound Event Detection. *2017 IEEE/ACM Transactions on Audio, Speech, and Language Processing* 25.6 (2017), 1291–1303. DOI: 10.1109/TASLP.2017.2690575.

[31] C. Clavel, T. Ehrette and G. Richard. Events Detection for an Audio-based Surveillance System. *2005 IEEE International Conference on Multimedia and Expo* (2005), 1306–1309. DOI: 10.1109/ICME.2005.1521669.

[32] R. Cai, L. Lu, A. Hanjalic and L.-H. Cai. A Flexible Framework for Key Audio Effects Detection and Auditory Context Inference. *2006 IEEE Transactions on Audio, Speech and Language Processing* 14.3 (2006), 1026–1039. DOI: 10.1109/TSA.2005.857575.

[33] A. Dempster, N. Laird and D. Rubin. Maximum Likelihood From Incomplete Data Via The EM Algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)* 39.1 (1977), 1–38.

[34] A.-r. Mohamed, G. Dahl and G. Hinton. Deep Belief Networks for Phone Recognition. *Nips Workshop on Deep Learning for Speech Recognition and Related Applications*. Vol. 1. 9. Vancouver, Canada. 2009, 39.

[35] G. Hinton et al. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine* 29.6 (2012), 82–97. DOI: 10.1109/MSP.2012.2205597.

[36] I. Goodfellow et al. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[37] A. Ng and K. K. *CS229 Lecture Notes*. 2018. URL: http://cs229.stanford.edu/notes/cs229-notes-deep_learning.pdf (visited on 01/06/2020).

[38] K. Hornik. Approximation Capabilities of Multilayer Feedforward Networks. *Neural networks* 4.2 (1991), 251–257. DOI: 10.1016/0893-6080(91)90009-T.

[39] S. W. Smith et al. *The Scientist and Engineer's Guide to Digital Signal Processing*. 2011. URL: http://www.dspguide.com/pdfbook.htm (visited on 08/22/2020).

[40] R. Bencina and P. Burk. PortAudio - An Open Source Cross Platform Audio API. *Proceedings of the 2001 International Computer Music Conference*. 2001. URL: http://hdl.handle.net/2027/spo.bbp2372.2001.036.

[41] T. Rauber and G. Rünger. *Parallel Programming: For Multicore and Cluster Systems*. Tiergartenstraße 17, 69121 Heidelberg, Germany: Springer-Verlag, 2010. ISBN: 978-3-642-04817-3.

[42] The Qt Company. *Qt for Python*. 2020. URL: https://doc.qt.io/qtforpython/ (visited on 08/23/2020).

[43] J. Thelin. *Foundations of Qt Development*. 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705, USA: Apress, 2007. ISBN: 978-1-59059-831-3.

[44] T. Fawcett. An Introduction to ROC Analysis. *Pattern Recognition Letters* 27.8 (2005), 861–874. DOI: 10.1016/j.patrec.2005.10.010.

[45] S. Hershey et al. CNN Architectures for Large-Scale Audio Classification. *International Conference on Acoustics, Speech and Signal Processing*. Vol. abs/1609.09430. 2017. URL: https://arxiv.org/abs/1609.09430.

[46] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto and H. Adam. Mobilenets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *Computing Research Repository* abs/1704.04861 (2017). URL: http://arxiv.org/abs/1704.04861.

[47] J. F. Gemmeke, D. P. W. Ellis, D. Freedman, A. Jansen, W. Lawrence, R. C. Moore, M. Plakal and M. Ritter. Audio Set: An Ontology and Human-Labeled Dataset for Audio Events. *2017 IEEE International Conference on Acoustics, Speech and Signal Processing*. 2017, 776–780. DOI: 10.1109/ICASSP.2017.7952261.