UNIVERSITÀ
DI TRENTO
Department of
Information Engineering and Computer Science

Doctoral School in
Information and Communication Technology

Embedded Real-time Deep Learning for a Smart Guitar

A Case Study on Expressive Guitar Technique Recognition

Domenico Stefani

Advisor
Prof. Luca Turchet
Università di Trento

September 2023

"What can this strange device be?
When I touch it, it gives forth a sound
It's got wires that vibrate and give music
What can this thing be that I found?

See how it sings like a sad heart
And joyously screams out its pain
Sounds that build high like a mountain
Or notes that fall gently like rain"

- Peart, Lee, Lifeson - 2112

Abstract

Smart musical instruments are an emerging class of digital musical instruments designed for music creation in an interconnected Internet of Musical Things scenario. These instruments aim to integrate embedded computation, real-time feature extraction, gesture acquisition, and networked communication technologies. As embedded computers become more capable and new embedded audio platforms are developed, new avenues for real-time embedded gesture acquisition open up. Expressive guitar technique recognition is the task of detecting notes and classifying the playing techniques used by the musician on the instrument. Real-time recognition of expressive guitar techniques in a smart guitar would allow players to control sound synthesis or to wirelessly interact with a wide range of interconnected devices and stage equipment during performance. Despite expressive guitar technique recognition being a well-researched topic in the field of Music Information Retrieval, the creation of a lightweight real-time recognition system that can be deployed on an embedded platform still remains an open problem.

In this thesis, expressive guitar technique recognition is investigated by focusing on real-time execution, and the execution of deep learning inference on resource-constrained embedded computers. Initial efforts have focused on clearly defining the challenges of embedded real-time music information retrieval, and on the creation of a first, fully embedded, real-time expressive guitar technique recognition system. The insight gained, led to the refinement of the various steps of the proposed recognition pipeline. As a first refinement step, a novel procedure for the optimization of onset detectors was developed. The proposed procedure adopts an evolutionary algorithm to find parameter configurations that are optimal both in terms of detection accuracy and latency. A subsequent study is devoted to shedding light on the performance of generic deep learning inference engines for embedded real-time audio classification. This consisted of a comparison of four common inferencing libraries, which focus on the applicability of each library to real-time audio inference, and their performance in terms of execution time and several additional metrics. Different insights from these studies supported the development of a new expressive guitar technique classifier, which is accompanied by an in-depth analysis of different aspects of the recognition problem. Finally, the experience collected during these studies culminated in the definition of a procedure to deploy deep learning inference to a prominent embedded platform.

These investigations have been shown to improve the state-of-the-art by proposing approaches that surpass previous alternatives and providing new knowledge on problems and tools that can aid the creation of a smart guitar. The new knowledge provided was also adopted for embedded audio tasks that differ from real-time expressive guitar technique recognition.

Keywords

Music Information Retrieval, Embedded Audio, Deep Learning Inference, Real-time Audio, Smart Musical Instruments.

Acknowledgements

First and foremost, I would like to acknowledge the invaluable help of my supervisor, Luca Turchet. I would like to thank him for introducing me to the wonderful world of music technology research, which is still magical to me. Had it not been for him, I would be working or doing research on something that now, in comparison, seems boring, theoretical, and worlds away. I would also like to thank Luca for his sound advice, and for granting me the freedom to focus on the topics that I found the most interesting. A huge thanks to everyone at Elk Audio for the great OS that I had the opportunity to use during my entire Ph.D. Thanks also to Johan for having me as a visiting researcher at the Centre for Digital Music in London.

A huge thanks must go to my colleagues at the University of Trento: real friends who made these last three years the most exciting and great period of my life. Thanks to Luca for the neverending discussions on our many shared passions, and for being so excited about yours that you made them become mine too. Thanks to Matteo for the lighthearted fun and joy you put into everything, the deep discussions in the hardest moments of this period, and the great music too! Thanks to Nishal for the big laughs, it would have been way more boring without you. Thanks to Alberto for the wisdom: whether it was about research, life, or the most obscure music, you always had stories, wise words, and suggestions that I cherished. Thanks to Mike for the good fun too, without you, road trips and conferences would not have been as fun. Also thanks to Greg: despite arriving only recently you have already been incredibly helpful, and I see our interactions are the same that are between all the like-minded people here. Thanks also to everyone for putting up with me and my quirks on a daily basis. A big thank also to the extended spippoling group, Ardan and Francy for the late-night discussions, wisdom, and just plain fun.

Thanks also to all the great people I had the chance to meet at the Centre for Digital Music in London. I am extremely grateful to: Aditya, Antonella, Nelly, Emilian, Keitaro, Marco, Andrea, Christopher, Christian, Pedro, Jeff, Chin-Yun, Ivan, Adan, Bleiz, Carey, Elona, Vjosa, Rodrigo, Ines, Ben, Ilaria, Jordie, David, Alex, Teresa, Monserrat, Yannis, Sara, Brendan, Ilias, Remi, Soumiya, Franco, Lewis, Saurjya, Johan. Thanks for the feedback from a thousand perspectives that were different than mine. Also thanks for taking me in as one of you! A very important thanks to Adi, a friend who I already miss so much. You made my time in London so much better, and you were a true friend in a temporary period when I never thought I would find one. Insanity is contagious.

Thanks also to all the special professors I had in high school and university; I would definitely not be here without you. Many thanks also to my lifelong friends back home. Thanks for everything, and for being great friends even in these three years that brought me physically further from you. A big thanks also to my family back home and siblings around Italy and the world, for all the support. Last but not least, I am profoundly grateful to Dani for all the patience, love, and support.

:=

Contents

List of Tables

List of Figures

Acronyms

Pd Pure Data. 9, 19, 20, 42, 49, 125

AI Artificial Intelligence. 144

AMI Augmented Musical Instrument. 13, 18, 20–22

BFCC Bark Frequency Cepstral Coefficient. 49, 51, 100, 125

BiRNN Bi-directional Recurrent Neural Network. 33, 34

CNN Convolutional Neural Network. 42, 56, 128

CSV Comma-separated values. 125

DAW Digital Audio Workstation. 22, 27, 144, 149, 150, 155, 160

DMI Digital Musical Instrument. 13, 31, 143

DSP Digital Signal Processing. 3, 20, 21, 28, 42, 143, 164

EC Evolutionary Computation. 63, 66–68, 71, 73, 74, 78, 80–85, 87, 91, 172

FFNN Feed-Forward Neural Network. 42, 54, 56, 60, 100, 101, 118, 171

FN False Negative. 70

FP False Positive. 70

FPGA Field Programmable Gate Array. 28, 42, 98

GPU Graphics Processing Unit. 41, 42, 95, 99, 147, 162

gRPC Google Remote Procedure Calls. 154

GUI graphical user interface. 150, 154, 155

IE inference engine. 11, 29, 30, 41, 59, 60, 93–116, 118, 141, 143, 147, 148, 153, 162, 167, 172, 173, 175

IoMusT Internet of Musical Things. 4, 6, 21, 143, 147

IoT Internet of Things. 143, 147

IQR Interquartile Range. 85–87

KNN K-nearest neighbors. 36, 44

LSTM Long short-term memory. 96, 104, 110

MFCC Mel Frequency Cepstral Coefficient. 49, 100, 124, 125

MIR Music Information Retrieval. 4, 6, 11, 31–34, 36, 41, 59, 67, 124, 169, 170, 175

MKL Modified Kullback-Leibler. 65, 79, 85, 87

OD Onset Detection. 62–65, 67, 78, 79, 84, 85

OSC Open Sound Control. 154

PDF probability density function. 14, 15

RNN Recurrent Neural Network. 34, 56

RPI4 Raspberry PI4. 123

rtMIR real-time Music Information Retrieval. 32–39, 41–43

SBC Single-Board Computer. 26–28, 95, 144, 146, 151, 154

SGD Stochastic Gradient Descent. 50

SMI Smart Musical Instrument. 6, 7, 13, 21, 22, 27, 62, 63, 132, 141, 143

SSH Secure SHell. 154, 155

STFT Short-time Fourier transform. 34, 65, 151

TFLite TensorFlow Lite. 29, 41, 50, 58, 59, 93, 95, 97, 99, 103, 106, 109–113, 124, 125, 129

TP True Positive. 70

TPU Tensor Processing Unit. 41, 42, 95, 98, 99, 147

VPU Visual Processing Unit. 42

VST Virtual Studio Technology. 10, 21, 27, 125, 144, 146, 148–150, 155, 160, 167, 175

WSL the Windows Subsystem for Linux. 148

Chapter 1

Introduction

The topic of this thesis is real-time deep learning deployment on embedded computers for the creation of a smart guitar, with a focus on expressive guitar technique recognition. This chapter explains the motivations and aim of this work (Section 1.1). Additionally, the main contributions and outcomes of this work are presented in Section 1.2, which includes the publications associated with the thesis (Section 1.2.1), submitted articles (Section 1.2.2), demos and talks relative to this thesis work (Section 1.2.3), open source software and free dataset delivered (Section 1.2.4), and two articles in progress (Section 1.2.5). Finally, the structure of the thesis is provided in Section 1.3.

1.1 Motivation and aim

Guitar and technology

Throughout history, the guitar has undergone vast changes and transformations, as luthiers, and then engineers, applied the technology of their time to the instrument. While this holds true since the inception of stringed instruments for the improvements in choices of body material, string types, glues, and shapes, one of the most drastic transformations affected the guitar in the 1930s, when it was mated with electronic pickups. Not many decades later, in the 1960s, people started exploring distortion

and other effects that could be obtained by manipulating the guitar's signal with electronics. This experimentation fostered the creation of a wide range of effects for the instrument, in the form of pedals and similar devices [1].

However, it was with the gain in popularity of electronic synthesizers, that musicians experienced a new wealth of otherworldly musical sounds. While the keyboard started becoming the default controller for synthesizers, guitarists and engineers envisioned the possibility of using the guitar as a controller, giving guitarists the ability to play the very same synthetic sounds, with the convenience of using the gestures and the instruments they were most familiar with.

Guitar synthesizers and other guitar controllers

This concept, referred to as the guitar synthesizer, dates back to the late 1970s when commercial products such as the Roland GR-500 [2], Arp Avatar [3], 360 Systems Spectre [4], and the Jen GS-3000 Syntar [5] were created. These devices added to the sonic potential of the instrument by allowing the guitarist to control a synthesizer in real-time. To do so, these systems tracked the pitch and amplitude throughout time (i.e., envelope) of the notes played on the guitar. In particular, these adopted special hexaphonic pickups, which capture an individual signal for each string, granting a higher performance in tracking polyphonic gestures with respect to trackers that operate on a single polyphonic signal. However, because of the limited signal-processing technology available at the time, most of the commercial products of those years were rather slow and inaccurate at tracking the player's actions [6]. As a result, these systems were not adopted by many guitarists[1], with some notable exceptions (e.g., Steve Hackett with the Roland GR-500 [7], Pat Metheny and King Crimson's Robert Fripp [8] with the Roland G303 guitar and GR-300 guitar synthesizer processor).

In the following decades, most of the products focusing on the use of the guitar as a controller resorted to using buttons or sensors underneath the fretboard to capture the left-hand position, and a pickup or other types of mechanical transducers to track which string was plucked and with which intensity. Some went as far as adopting plastic for the entire construction of these controllers and replaced strings with low-tension plastic cables, such as the Casio DG-10 and SynthAxe [9]. Others, such as the Starr Labs' Ztar, featured a fretboard covered in buttons, replacing entirely the strings. Sensor-equipped controllers allowed musical instrument companies to

[1]Notably, many of these guitar synthesizers were modified by users to be controlled by a keyboard, first via control voltages, and later with the MIDI protocol.

overcome the limitations of real-guitar tracking that were due to the limited signal processing capabilities of the time, but they resulted in instruments rather different from guitars [10]. In particular, most of these controllers required players to use different gestures, actions, and forces from those that would have been used on a regular guitar. While these enhanced the possibility of the guitar as an interface, it can be argued that there is still a place for actual guitars as a controller of synthetic sounds.

Companies such as Roland kept producing and improving guitar synthesizers throughout the years, resulting in improvements in the quality of the pitch and amplitude tracking systems thanks to Digital Signal Processing (DSP). To date, the company Boss, which is part of the Roland Corporation, still produces guitar synthesizers such as the GP-10, GR-55, and SY-1000 pedalboards, along with their GK-3 and GK-5 hexaphonic pickups. Moreover, companies such as Godin produce guitars with a hexaphonic pickup integrated into the design, and Roland/Boss-compatible connectors. Other examples of guitar controller technology are represented by compact audio-to-midi converters and pickup systems such as the Sonuus G2M [11] converter and the Fishman TiplePlay systems [12]. Additionally, reportedly state-of-the-art tracking accuracy is achieved by MIDI software trackers such as Jam Origin MIDI Guitar 1 and 2 [13].

An expressive bottleneck

Despite the improvements made throughout the years in guitar synthesizers, conventional note tracking does not take into account the wide range of expressive nuances used by guitarists to shape the sound of their instruments. These differences in the timbre of guitar sounds are introduced through the use of different expressive techniques, which range from changing the point where strings are plucked, to bending or muting the strings. The use of different techniques influences the timbre of the notes, which is defined as the perceived quality of a sound. Timbre is determined by the spectral content and envelope of notes (i.e., how the sound changes over time). Therefore, expressive techniques that are not based solely on pitch and loudness variations cannot be tracked with conventional note-tracking systems. For instance, while pitch-based techniques such as bending or vibrato, and envelope-based techniques (e.g., staccato) will be detected as a change in pitch or amplitude, other techniques such as palm muting or harmonics will not be tracked. As a consequence, the output sound cannot be controlled through the use of these techniques. For this reason,

these tracking systems represent a bottleneck between the expressive playing on the musician's part and the synthesizer's sound [14]. To affect characteristics of the synthesizer's sound other than pitch and amplitude, current guitar synthesizers require players to use rather unnatural interaction patterns, such as interaction with buttons or knobs on the synthesizer interface [15].

However, real-time recognition of expressive techniques on the guitar would allow for a more accurate description of the guitarist's playing, which can be repurposed in real-time to shape the synthesizer's sound.

Expressive technique recognition

Expressive technique recognition has been a topic of interest in the field of Music Information Retrieval (MIR) [16] for many years, and it has been studied for several applications such as enhanced automatic music transcription [17, 18] and analysis of music performances [19, 20]. Compared to pitch tracking, technique recognition requires a deeper analysis of the frequency spectrum and temporal evolution of sounds. Additionally, research on instrument playing technique recognition often focuses on offline algorithms that can benefit from the availability of entire signal recordings and a lack of tight deadlines for recognition. The access to large windows of the audio signal is extremely helpful, as some high-level properties such as specific expressive techniques develop in a very clear and distinctive way only at the late phase of a note (i.e., hundreds of milliseconds after the attack) [20].

Differently from the offline case, real-time expressive technique recognition requires algorithms that can produce classification results in as little time as possible. When the technique recognition information is repurposed to produce or shape a new synthetic sound, the latency between the beginning of the note (i.e., note onset) and that of the new sound must be as low as 30 ms or smaller for the two to be perceived as simultaneous by the human hearing system [21]. A larger time discrepancy between the musician's actions and the reaction of such a system will affect their ability to follow their own rhythm. As a result, real-time recognition is particularly challenging as many data preprocessing techniques cannot be applied, and future inputs cannot be seen without adding latency between sound events and the recognition results. In addition to the immediate generation of synthetic sounds, a wide range of less latency-demanding applications can be devised for a guitar controller within the Internet of Musical Things (IoMusT) paradigm [22]: the playing technique information extracted in real-time can be repurposed to wirelessly trigger stage lighting effects,

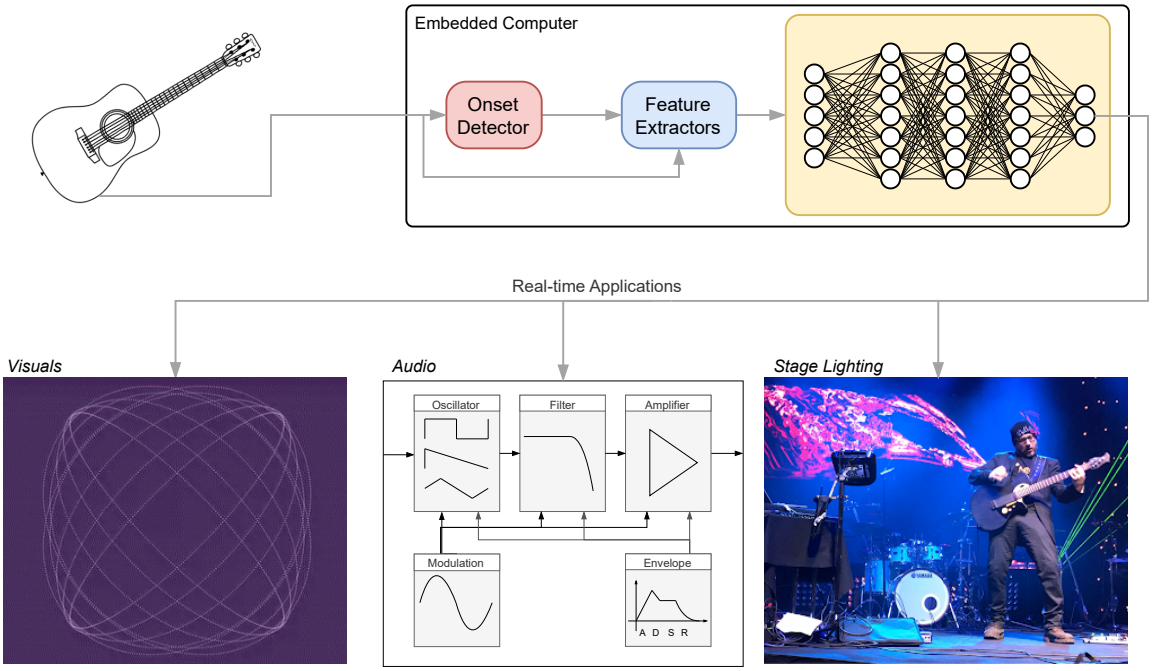


Figure 1.1: *Overview of a system for the real-time recognition of expressive playing techniques, and potential applications that repurpose the extracted information in real-time.*

control fog machines, or manipulate visuals that accompany the guitarist's performance in real-time. With such a system, using a specific expressive technique can generate a response through one or more of these media. This concept departs from the idea of an improved guitar synthesizer, which only served as a first inspiration for this thesis work, and embraces a broader pool of possibilities. Figure 1.1 shows an overview of the proposed real-time recognition system with some of the potential applications.

Real-time expressive technique recognition poses a series of additional complexities in contrast to the offline case, which derives from the need for low latency detection. In particular, while modern deep learning approaches have proved accurate for expressive technique recognition, offline approaches may not always be applicable to real-time execution. Such cases require either a redesign of the offline solutions or clever optimization of the deep learning models and optimization.

Embedded intelligence and Smart musical instruments

In order to integrate such a system into a guitar synthesizer, however, it needs to be able to run on a compact embedded computer, while still maintaining the latency

guarantees required for real-time use. Embedded execution adds additional complexity to the development of real-time audio recognition and analysis systems since embedded devices and single-board computers offer limited computing power in comparison to personal computers. However, recent years have witnessed the availability of increasingly powerful single-board computers and embedded platforms specifically targeting audio and musical applications (e.g., Bela [23] and Elk Audio OS [24]).

By focusing on embedded real-time execution, we situate our work in the context of Smart Musical Instruments (SMIs), which is a family of Internet of Musical Things (IoMusT) devices that are "characterized by sensors, actuators, embedded intelligence, and wireless connectivity to local networks and to the Internet" [25]. In this context, embedded intelligence has been defined as the integration of embedded computation and the real-time extraction of features and sound properties from the sound signal, along with context-awareness and proactivity capabilities. In particular, expressive guitar-playing techniques represent a rather high-level feature of each sound played on the instrument, and enabling their real-time recognition on compact embedded computers will enable researchers to investigate further applications that harness the interconnected nature of Smart Musical Instruments. Furthermore, investigating the technical solutions for deep learning execution for real-time analysis on embedded audio platforms can help foster the creation of new audio devices and Smart Musical Instruments. However, when this Ph.D. started (November 2020), the state-of-the-art of embedded real-time deep-learning approaches to MIR was scarce.

Research gap and aim

At the beginning of this research, we were faced with the absence of studies documenting the real-time execution of deep-learning inference for audio on embedded devices. Conversely, deep-learning approaches had become prominent for many MIR tasks, including expressive guitar technique recognition. Moreover, we found how online/real-time execution was quite rarely a constraint or metric adopted in guitar technique recognition studies, making it complicated to parse studies on the topic. These gaps acted as obstacles in the development of a smart guitar as we envisioned it, but we soon realized how their impact had broader implications on smart musical instruments, and deep-learning-equipped audio devices in general.

The aim of this work is to address technical challenges in the real-time execution of deep-learning models on embedded computers for music applications. Moreover, we adopted the guitar as a case study, focusing on aiding the development of a smart

guitar through an approach for embedded real-time expressive guitar technique recognition. Furthermore, we aimed to address the technical challenges in the deployment of deep-learning-based recognition pipelines on embedded computers for music applications. While we focused on the guitar as a case study, the technical challenges addressed have broader implications that can apply to a wider range of musical instruments, given that proper data is provided. In particular, we focus on the optimization of existing onset detectors for real-time applications, with both accuracy and latency as optimization objectives. Additionally, we investigate the current tools for the execution of deep learning models in real-time on embedded audio platforms. Finally, we aim to aid the deployment of neural networks to embedded audio platforms through the development of open-source software and the creation of detailed guides. In particular, we focused on the open-source real-time Elk Audio operating system. We aimed to extend the implications of this doctoral research besides SMIs, helping the development of deep-learning-equipped audio devices in general.

1.2 Outcomes

1.2.1 Publications

The doctoral research presented in this thesis has resulted in the following scientific publications:

[i] Domenico Stefani and Luca Turchet. *Bio-Inspired Optimization of Parametric Onset Detectors*. In Proceedings of the 24th International Conference on Digital Audio Effects (DAFx20in21), volume 2, pages 268-275, Sept. 2021;

[ii] Domenico Stefani and Luca Turchet. *On the Challenges of Embedded Real-Time Music Information Retrieval*. In Proceedings of the 25-th International Conference on Digital Audio Effects (DAFx20in22), volume 3, pages 177-184, Sept. 2022;

[iii] Domenico Stefani, Simone Peroni, and Luca Turchet. *A Comparison of Deep Learning Inference Engines for Embedded Real-Time Audio Classification*. In Proceedings of the 25-th International Conference on Digital Audio Effects (DAFx20in22), volume 3, pages 256-263, Sept. 2022;

[iv] Domenico Stefani. *Riconoscimento in tempo reale di tecniche espressive per chitarra su embedded computers*. In Corpi Fisici | Physical Bodies, Atti del XXIII Colloquio di Informatica Musicale. AIMI - Associazione Informatica Musicale Italiana, DADI - Dip. Arti e Design Industriale. Università IUAV di Venezia, 2023;

[v] Domenico Stefani and Luca Turchet. *Real-Time Embedded Deep Learning on Elk Audio OS*. In Proceedings of the 4th International Symposium on the Internet of Sounds (IS^2), Oct. 2023, Pisa, Italy.

1.2.2 Submitted Articles

[vi] Domenico Stefani and Luca Turchet. *Embedded Real-Time Expressive Guitar Technique Recognition* Submitted to IEEE/ACM Transactions on Audio, Speech, and Language Processing.

1.2.3 Demos and Talks

This research was presented with the following demos and workshop talks:

1. Domenico Stefani. *Demo of the timbreid-vst plugin for embedded real-time classification of individual musical instrument timbres.* In 27th Conference of Open Innovations Association (FRUCT), volume 2, pages 412-413, 2020;

2. Domenico Stefani. *Embedded Real-time Expressive Guitar Technique Recognition* In Embedded AI for NIME Workshop [26], International Conference on New Interfaces for Musical Expression, Jun. 2022;

3. Domenico Stefani. *Demo: Real-Time Embedded Deep Learning on Elk Audio OS.* In International Symposium on the Internet of Sounds (IS^2), Oct. 2023, Pisa, Italy.

1.2.4 Open-Source Software and Data

Part of the output of this research work is in the form of software that was released as open-source. The following is a list of the main pieces of software delivered, along with the links to the relative online repositories:

- **`C++` TimbreID library:**
  Port of sixteen Pure Data (`Pd`) objects from the Timbre ID external [27] to the `C++` language with the JUCE Framework. The library includes the expressive guitar technique recognition plugin described in Chapter 6 and a feature extraction plugin. The code has been developed solely by myself, based on the original `Pd` code for the library.
  `https://github.com/CIMIL/cpp-timbreID`

- **Deep inference wrappers:**
  Library wrappers for classification with four different deep inference wrappers. The code has been developed by myself and Simone Peroni.
  `https://github.com/CIMIL/cpp-deep-inference-wrappers`

- **Guide for Deep Learning deployment on Embedded Computers with Elk Audio OS:**
  Templates, example projects, and an in-depth stepwise guide to deploying deep

learning inference Virtual Studio Technology (VST) plugins to embedded computers with Elk Audio OS. The code has been developed solely by myself.
https://github.com/CIMIL/elk-audio-AI-tutorial

Furthermore, due to the absence of free datasets with high-quality monophonic recordings of acoustic guitar techniques (both pitched and percussive, captured with internal pickups), an extensive audio dataset was recorded, edited, labeled, and made freely available [28]. The *Acoustic Guitar Playing Technique dataset* (AGPTset) contains 15 hours and 55 minutes of monophonic recordings of 12 expressive guitar playing techniques (pitched and percussive). Of these, 10 hours and 4 minutes of recordings encompassing 8 of the 12 techniques have been labeled, meaning that onsets were identified at the millisecond level and their timestamp was annotated alongside playing technique information. The precision of the annotations of each onset is a property not often found in music datasets. As a result, 32,592 individual notes have been labeled. Recordings cover 6 guitar players on 7 different acoustic steel-string guitars.

- Domenico Stefani and Luca Turchet. *AGPTset (Acoustic Guitar Playing Technique dataset).*
https://doi.org/10.5281/zenodo.10159491

1.2.5 Articles in progress

The work done during the 6-months research period spent at the Centre for Digital Music (C4DM) at the Queen Mary University of London culminated in the development of an offline embedded classifier for intended emotion from guitar and piano improvisation excerpts. The system is in its user-study stage, and it has been tested with three guitarists and one piano player. Additionally, the development of a real-time version of the embedded classification system, based on the offline method, is underway.

- Luca Turchet, Domenico Stefani, and Johan Pauwels. *Emotionally-aware Smart Musical Instruments* to be submitted to IEEE Transactions on Affective Computing;

- Domenico Stefani, Johan Pauwels Luca Turchet. *Real-time Emotion Recognition on Smart Musical Instruments* TBD;

1.3 Thesis Structure

This thesis is structured as follows:

Chapter 2 provides the background of the thesis research and a review of the state of the art previous to our contribution.

Chapter 3 presents an overview of the challenges of embedded real-time MIR, along with potential solutions or tradeoffs. Some of the approaches and solutions presented are demonstrated with the implementation of an embedded real-time classifier of expressive guitar techniques. Reflections on the limitations of the classifier presented shaped the subsequent research direction, and therefore the content of the following chapters.

Chapter 4 presents a new approach to the optimization of parametric onset detectors for both accuracy and low latency, using evolutionary algorithms.

Chapter 5 describes a comparison of four inference engines (IEs) for the execution of deep learning models on embedded computers. The comparison focuses on assessing whether the available generic IEs are suitable for real-time execution for audio, and how their performance differs in terms of execution time on a classification problem. Furthermore, we ranked the four IEs according to six additional metrics, including resource utilization, ease of use, and quality of documentation.

Chapter 6 then builds on the works described in the previous chapters and presents a flexible-latency approach to embedded real-time expressive guitar technique recognition. The chapter delves into the problem of playing technique recognition in real-time for guitar. Here we also provide a classifier that is based on the idea that was presented in Chapter 3 as an example, which was widely improved and revised here.

Chapter 7 draws from experience collected during the previously mentioned works, and presents a guide to deep learning deployment for audio on embedded systems using Elk Audio OS. Differently from previous work, the guide is not only targeted at running MIR for classification on embedded computers, but it extends more broadly to the use of neural networks for audio (e.g., for audio effect modeling).

Chapter 8 concludes the thesis, summarizing the contributions of this work, and showing how this thesis improved the state-of-the-art presented in Chapter 2.

Chapter 2

Background and State Of The Art

The focus of this thesis is on the development of technologies that can aid the creation of a smart guitar [29], which can be defined as an example of an SMI. Smart Musical Instruments have been defined by Turchet *et al.* [30] as musical instruments "characterized by sensors, actuators, embedded intelligence, and wireless connectivity to local networks and to the Internet" [25]. These can be described as Digital Musical Instruments (DMIs) [31] with a strong focus on the self-contained nature of the instruments and their networking capabilities. Despite the very peculiar combination of features that define SMIs, their definition leaves space to overlap with that of Augmented Musical Instruments (AMIs) [31] (or instrument augmentations). Their definition will be presented later in this chapter, along with relevant guitar augmentation studies (Section 2.2).

As previously described, SMIs are complex instruments composed of input sensory stages, embedded processing capabilities, actuator technology, and network connectivity. In particular, we focus on the part of a smart guitar that manages the real-time extraction of audio features to capture musical *gestures*.

The musical gestures of interest for this work are *expressive playing techniques*. In the past, the study of physical gestures has led to different definitions and distinctions [32–35]. Expressive playing techniques fall under definitions of musical gestures such as that adopted by Miranda *et al.* [31].

Automatic recognition of expressive playing techniques has been investigated for

many instruments, including guitar, and in both offline and real-time contexts. An overview of the most relevant works is presented in Section 2.3. Furthermore, Section 2.4 will discuss the state of the technology for real-time audio analysis on embedded computers. Finally, Section 2.5 will situate the current work in the state of the art, drawing a summary of the chapter.

2.1 Terminology

2.1.1 Latency

The *latency* of a system can be defined as the delay introduced by the system itself between its input and the output results produced. Additionally, we often conceptually split larger systems into smaller blocks: we can therefore define several latencies (e.g., latency of detection, latency of classification), which sum to a total latency. However, it is very important to note that the total latency of a digital processing system is dictated by a large number of hardware and software factors and it can rapidly vary, therefore latency cannot be identified as a single number. Instead, we must refer to system latency with measurements about its distribution, e.g., average latency, worst-case latency (maximum latency), latency variability (variance, standard deviation, Inter-quartile range), and ultimately its probability density function (PDF) or density estimation from measurements.

Moreover, *jitter* is the variation within the latency of a system, indicating the degree of inconsistency of the latency itself. Jitter is a relevant characteristic of audio systems as it has been proven to affect human perception when larger than tolerable constraints [36].

For an example of the latency components of a digital audio system, we can take an audio processing software running on a personal computer as a system. In this case, we can identify latency as the delay between any one instant in the input audio signal and the relative instant in the output (processed) signal. In the example case, part of the total delay of the system is to be attributed to the analog-to-digital and digital-to-analog conversion stages. Additionally, digital audio processing is commonly performed by collecting buffers of audio samples and processing them in batches. The size of the input and output audio buffers, and therefore latency introduced by the buffers, is often a parameter choice left to the user. In modern PCs, audio is delivered to programs by the operating system, which can introduce

more or less delay depending on the nature of the audio drivers used. Additionally, when looking at the sole audio processing software, latency can be introduced both by the nature of the processing performed, and the actual processing computation to be performed on the CPU. Such an example can also show how latency cannot be a single value, as the operating system would be handling multiple tasks at any time and different calls to the audio processing routine must share computing resources with other processes. This may cause the system to take more time to complete some processing calls, therefore potentially introducing jitter.

In the chapters of this thesis that discuss expressive guitar technique recognition, we will define *recognition* latency as the delay between the onset of a note in the audio signal and the moment when the relative recognition result, i.e., the predicted expressive technique, is reported.

Figure 2.1, Figure 2.1, and Figure 2.1 show a toy example with two systems and a hard tolerability threshold on latency. Figure 2.1 represents the two PDFs where System A has a higher mean but contained variance (bell width), while System B has a lower latency mean but higher variance, which makes it not compliant with the set deadline. Figure 2.1 shows an estimation of the latency distributions obtained from measurements and displayed with histograms. Finally, Figure 2.3 presents an alternative representation of the two systems with latency observations as boxplots, along with their mean and standard deviation values.

2.1.2 Soft and Hard Real-time

Real-time computing refers to hardware and software systems that are subject to a *real-time constraint*, which requires them to guarantee a response within specified time constraints, often referred to as *deadlines*. This concept is essential in various applications where safety or human perception is involved.

Real-time systems are categorized as either soft real-time systems or hard real-time systems based on their timing constraints. In real-time computing, hard real-time systems must meet strict timing constraints, and missing a deadline can have serious consequences. On the other hand, soft real-time systems can tolerate occasional missed deadlines, which may degrade the system's performance but are not catastrophic. This distinction is relevant to audio processing systems, as hard real-time requirements are critical in applications such as live sound processing, where any delay in processing the audio data can be perceptible and disruptive. In contrast, soft real-time constraints may be acceptable in non-critical audio applications, where

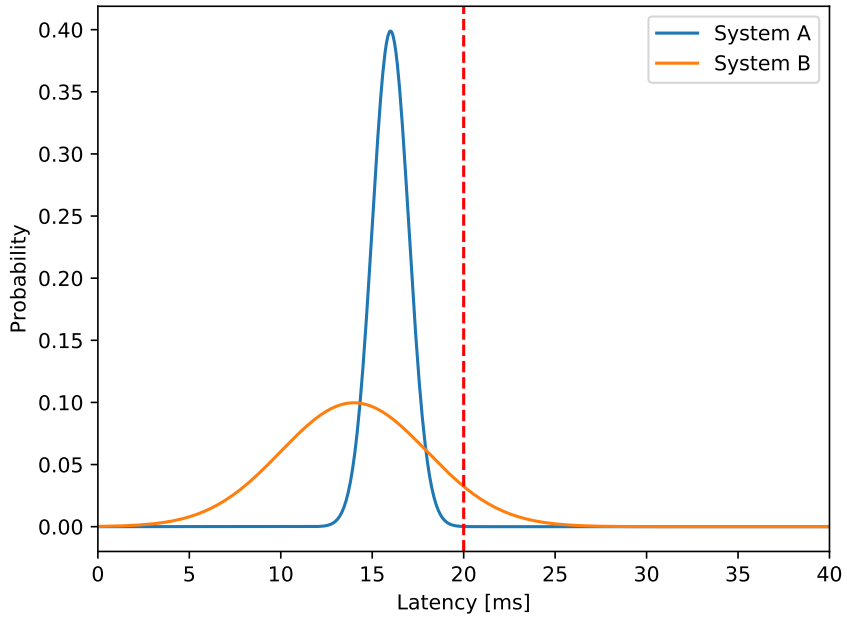


Figure 2.1: *Example of the latency PDF of two different systems and a set tolerability threshold (red line). Despite System B having a lower mean latency, jitter makes it so that it does not comply with the tolerability threshold on some occasions. On the contrary, System A has a higher mean but more contained jitter and can be more suitable for a case where the red line represents a hard deadline.*

occasional delays in processing may be tolerated without significant impact on the overall system performance.

The concepts of hard and soft real-time are therefore tightly related to the concepts of latency, jitter, their potential degree of tolerability, and the risk associated with missing deadlines. As previously mentioned, anything that produces audio samples in the processing routine of a digital audio system is subject to hard real-time constraints, as missing even a single audio deadline will lead to a significant degradation in system performance by causing audible audio artifacts. On the contrary, tasks such as reading control values (e.g., virtual potentiometers on audio processing software) have a much more limited impact on the system performance, therefore missing a "control" deadline is not a critical event, and often some degree of jitter can be tolerated, especially when it happens sporadically (soft real-time constraints).

The chapters of this thesis that discuss expressive guitar technique recognition present examples of soft real-time constraints on a recognition deadline, where the recognition result is envisioned to be used as a control parameter for eventual audio processing or generation algorithms. Therefore, missing a recognition deadline does

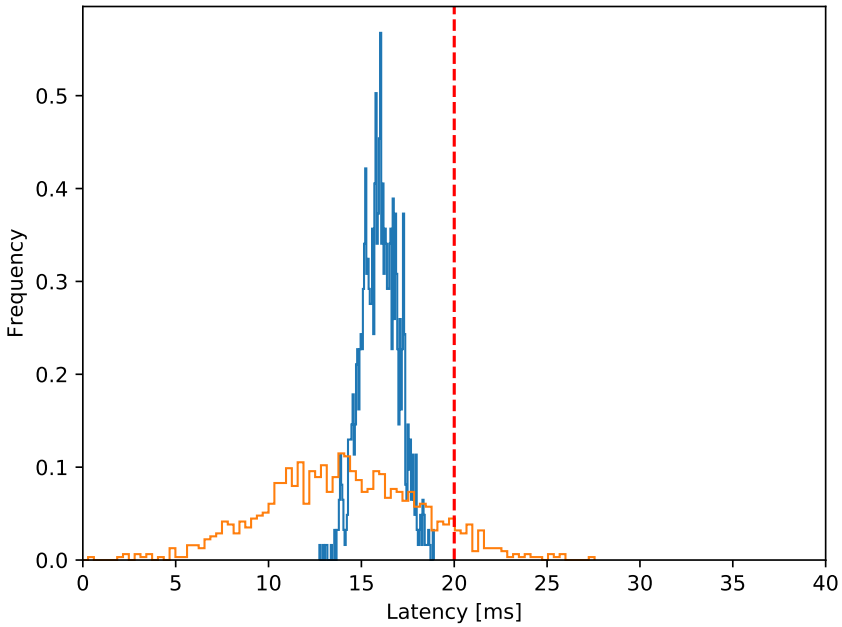


Figure 2.2: *Estimation of the distribution density of the latency of the systems in Figure 2.1.*

not have a catastrophic effect on the performance of the system.

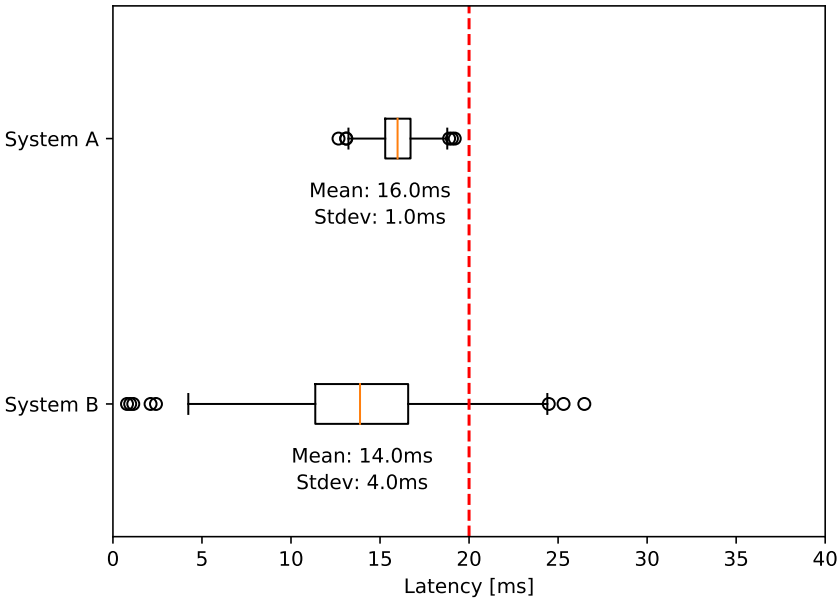


Figure 2.3: *Alternative representation of the two latency distributions from Figure 2.1 with boxplots.*

2.2 Guitar Augmentations and Smart Musical Instruments

AMIs have been defined as conventional (i.e., acoustic or electric) musical instruments that have been fitted with several sensors to provide the player with more control over the sound produced by the instrument [31]. AMIs have often been referred to as hyperinstruments or hybrid instruments [37, 38]. However, the term Augmented Musical Instrument has also been used to refer to the use of actuators [39] to act on the resonant surfaces of the instrument. Despite the existence of these two augmentation categories, i.e., augmentation "by sensors and by actuators" [40], this thesis work mostly overlaps with the first, due to the focus on expressive guitar technique recognition and use of deep learning models for real-time signal analysis.

Instrument augmentations have been proposed for a variety of musical instruments such as the piano [41, 42], violin [43, 44], cello [45], trumpet [46], flute [47], hurdy-gurdy [48], bagpipe [49, 50], mandolin [51], percussion instruments [52], and guitar [15, 53–66].

Puckette [53] proposed an interesting approach to the guitar synthesizer that involved applying several effects and morphing techniques to guitar signals, controlling a synthesis engine with pitch tracking information. Puckette's solution allowed the

signal to be processed with a latency that only depends on the signal path, while the feature extraction latency only delays the control parameters. Albeit innovative in its approach, Puckette's solution is limited to morphing the guitar sounds via audio effects.

With the Mobile Wireless Augmented Guitar [54], Bouillot *et al.* surmounted the limited computing capabilities of embedded devices available at the time by devising a wireless streaming system for both audio and control parameters. This approach enabled signal processing through Pure Data (`Pd`) patches, however, the proposed protocol compromised dynamically on PCM sample size to overcome transmission latency, and the authors suggested that a more capable embedded device could eliminate the need for the transmission of the unprocessed signal.

Reboursière *et al.* presented the Multimodal Guitar [55], which is a toolbox for guitar augmentation composed of `Pd` and Max/MSP algorithms for audio analysis, gestural control, and audio synthesis. This selection of algorithms fits well the augmentation task, however, the implementation in `Pd` and Max/MSP means that the musician is limited to a separate processing unit.

Angulo *et al.* [56] presented an approach to hexaphonic guitar transcription and visualization that consisted of a set of feature extractors for pitch, energy, chroma[1] information, and a repurposing system that produced matching visuals. As the previously mentioned solutions, it required an external processing unit.

Furthermore, the HITar by Martelloni *et al.* [57, 58] is an augmented steel-string acoustic guitar with percussive hit detection capabilities. To do so, the authors fitted the guitar with several piezoelectric transducers whose signals are routed to a soundcard and a laptop, where both classification and embedding learning are performed. The work of the authors will be further discussed in Section 2.3, as it pertains to guitar technique recognition.

Lähdeoja's augmented guitar [59] was fitted with a hexaphonic pickup and actuators that could vibrate the guitar's body for amplification of the processed signal. In particular, the separate signals from the strings were sent to a laptop, where they were processed with granular synthesis algorithms and crossfaded with slow-attack flute samples. However, despite their augmented guitar being based on multiple interconnected devices such as the guitar itself and a laptop, the authors described the importance of a self-contained instrument to overcome the divide in the interface

[1] Chromagrams are a representation of spectral energy in the 12 semitones of the musical octave [67]

design.

GuitarAMI is instead a project by Meneses *et al.* [60, 61] that features a nylon string acoustic guitar (i.e., classical guitar) and sensors such as an ultrasonic sensor and accelerometer capture gestural data. This data was then used to control algorithms that would help overcome limitations of the original instruments, such as the limited duration of note sustain. In the first prototypes, all the sensor and audio data processing was performed by a laptop, however, the authors stated how "the number of connections and cables increased the possibility of malfunctioning and made the instrument less robust" [61]. For this reason, the instrument was provided with a Raspberry Pi embedded computer for sensor and audio processing. The instrument was defined as a prototype under construction and is subject to investigations on augmentation possibilities and interaction strategies, as it was subsequently provided with different sensors [62]

GuiaRT [62, 63] is a platform for the development of a DSP-based interface for guitarists to be used in interactive musical contexts. differently from GuitarAMI, GuiaRT was fitted with a hexaphonic pickup and prepared for the extraction of low-level descriptors such as start time, string number, fundamental frequencies, spectral centroids, and some mid-level features [68, 69]. Contrarily to the previous AMI, GuiaRT was used in conjunction with the Max software [70] running on a laptop for feature extraction.

Graham *et al.* [64] instead proposed an AMI design that revolves around an innovative approach to audio spatialization for a guitar performance. The authors present a system in which features such as amplitude and pitch contour are extracted from the signals coming from an exaphonic pickup. These are combined with information about ancillary gestures (i.e., gestures that do not affect the sonic output, see [71]) captured with an Xbox Kinect. The combination of these features is used to control the specialization of hexaphonic signals in a loudspeaker array system.

The RANGE guitar by MacConnell *et al.* [15] was presented as a "minimally-invasive" AMI. RANGE was fitted with touch potentiometers and a Linux-embedded computer, i.e., Beaglebone, for control and DSP. Processing was performed with `Pd` patches loaded into the embedded computer. The sensor array and processing unit of RANGE were presented as a versatile synthesizer controller" by itself and, were considered separate from the guitar itself. In particular, the authors stated how the sensor and processing interface could be used to control synthesizer parameters while the guitar was "played as usual".

The MIT Chameleon Guitar by Zoran *et al.* [65,72] is instead an instrument with physically interchangeable acoustic resonators and DSP capabilities. The guitar was provided with piezoelectric transducers and a DSP unit, which was fitted with an algorithm designed to implement a "virtual chamber" based on the interchangeable resonator of choice. Additional DSP algorithms for the cross-modulation of different sensor signals were proposed by the authors. The Chameleon Guitar is an example of a fully embedded AMI.

Other examples of guitars with augmented sonic capabilities embedded into the instrument are *modeling guitars* such as the Line 6 Variax [73] and the Fender VG [74] electric guitars. These offer the possibility of emulating different guitars (e.g., acoustic, 12-string) and tunings, through the use of hexaphonic pickups and internal processing units. These solutions differ from MIDI guitars and simple guitar synthesizers in the increased expressiveness carried from the guitar signal and transferred to the modeled sound. This is likely to be attributed to carefully designed modeling algorithms that do not. However, expressive playing only translates to the modeling sounds provided with these instruments, while no intermediate expressive representation of the signal can be accessed for alternate uses.

Additionally, another instrument that blurs the line between AMI and SMI is the MUSE Synth Guitar [66], a custom self-contained guitar synthesizer built for the guitarist of the rock band MUSE. The guitar was fitted with a Fishman MIDI hexaphonic pickup and an embedded computer. The MIDI pickup was used to control a VST synthesizer plugin running on the embedded computer, effectively making for one of the few self-contained guitar synthesizers.

A much different approach from all the previous studies was adopted by Vanegas [75] with the MIDI Pick. It consisted of a guitar pick with sensors that were used to capture finger pressure information. Such data was captured by a microcontroller and wirelessly sent to a laptop to control effects and synthesis algorithms. Despite the wireless capabilities and execution of some embedded computations to capture sensor data, the MIDI Pick relies on a laptop for audio processing.

Smart Musical Instruments

Despite the affinity with AMIs, we situate this work in the context of Smart Musical Instruments [29]. This emerging class of musical instruments was defined as a family of IoMusT devices that are "characterized by sensors, actuators, embedded intelligence, and wireless connectivity to local networks and to the Internet" [25]. Some

examples are the Sensus Smart Guitar [30], the Smart Cajón [76, 77], and the Smart Mandolin [78].

In [40] the author proposed a reflection on the distinction between AMIs and SMIs. The author highlighted how AMIs are often based on a series of devices connected to the instrument via cables, such as a PC, a soundcard, power supplies, and a speaker or actuators. However, these often make for a setup that is cumbersome to transport and awkward to use during a performance on stage [15, 79]. Conversely, SMIs have been envisioned as self-contained instruments, where any processing or "intelligence" is embedded into the instrument. The resulting remarks by the author were that SMIs are not a subset of AMIs, but they rather share some common features.

Additionally, this thesis revolves around technologies that can enable the extraction of rather high-level features from real-time playing (i.e., use of expressive techniques) thanks to modern deep learning neural networks. The embedded execution of deep learning inference can enable advanced manipulation and analysis that is not often found in AMIs.

The most prominent example of guitar-based SMI (or *Smart Guitar*) is the Sensus Smart Guitar [29]. Sensus is a guitar that was fitted with sensors, control surfaces, wireless communication technology, and most importantly an embedded computer embedded with a Digital Audio Workstation (DAW) for internal processing of sensor data and audio. Sensus was also able to receive real-time streaming of audio content and musical control messages from multiple mobile phones for interactive jamming.

The work of this thesis partially overlaps with the field of AMIs and SMIs, therefore we compare our proposed approach to these related augmented and smart guitars (see Table 2.1). In particular, the proposed approach for expressive guitar technique recognition shares the same attention to real-time execution as any AMI, but it also focuses on the embedded execution of analysis and recognition algorithms. Furthermore, we adopt technologies that enable the integration of audio signal processing and sound synthesis in the same embedded device where recognition is performed [80]. Finally, differently from most other guitar augmentation projects, we do not focus on the acquisition and processing of sensor data other than the audio signal.

Table 2.1: *Comparison of the related guitar augmentations with respect to technique recognition and their embedded capabilities, or lack thereof.*

Augmentation	**Instrument**	**Performs Technique recognition**	**Embedded Sensor Processing**	**Embedded Signal analysis**	**Embedded Audio Processing**	**Academic or Open source**
Puckette's Patch for guitar [53]	Electric Guitar	No	No	No	No	**Yes**
Mobile Wireless Augmented Guitar [54]	Electric Guitar	No	**Yes**	no	no	**Yes**
Multimodal Guitar [55]	Any Guitar	No	No	No	No	**Yes**
Angulo *et al.* [56]	Classical guitar	No	No	No	No	**Yes**
HITar [57, 58]	Acoustic guitar	**Yes**	No	No	No	**Yes**
Lähdeoja augmented guitar [59]	Acoustic guitar	No	No	No	No	**Yes**
GuitarAMI [60, 61]	Classical Guitar	No	**Yes**	No	**Yes**	**Yes**
GuiaRT [62, 63]	Classical guitar	**Yes**	No	No	No	**Yes**
Spatial audio guitar by Graham *et al.* [64]	Electric Guitar	No	No	No	No	**Yes**
RANGE guitar [15]	Electric Guitar	No	**Yes**	No	**Yes**	**Yes**
MIT Chameleon Guitar [65, 72]	Hybrid Guitar	No	**Yes**	No	**Yes**	**Yes**
Line6 Variax [73]	Electric Guitar	?	No	?	**Yes**	No*
Roland/Fender VG [74]	Electric Guitar	?	No	?	**Yes**	No*
Elk MUSE Synth Guitar [66]	Electric Guitar	No	No	**Yes**	**Yes**	No*
MIDI Pick [75]	Pick	No	**Yes**	No	No	**Yes**
Sensus smart Guitar [29]	Electric Guitar	No	**Yes**	No	**Yes**	**Yes**
This thesis' work	Acoustic Guitar	**Yes**	No	**Yes**	**Yes**	**Yes**

* Commercial and/or closed-source projects.

2.3 Expressive guitar playing technique recognition

Expressive playing techniques, or *instrument playing techniques*, can then be defined as musical gestures that impress expressive variability to how musical notes are produced. Musical gestures are briefly introduced by Miranda *et al.* [31], while Cadoz *et al.* [81] presented a deeper analysis. In their work, Lostanlen *et al.* [82] present clear evidence on the pervasive use of playing techniques in many music genres, highlighting even how gestural information can be a superior organizing principle in music corpora, in comparison to the more widespread classification depending on organology (the categorization into different instruments). Automatic recognition of expressive playing techniques has been investigated for a wide range of musical instruments such as piano [83], percussion, clarinet [84], violin [85], erhu [86] and guitar [57, 58, 82, 87–100]. Furthermore, some went as far as proposing taxonomies and benchmarking existing timbral features for instrument and playing technique detection with a large number of instruments [82]. However, most of the existing approaches for playing technique recognition have been focusing on offline recognition for tasks such as music transcription.

Ozaslan *et al.* [87] presented an approach and preliminary results for the detection of a few different left-hand articulations on nylon string guitars. However, the results were obtained with a non-real-time system and a rather simple approach.

Reboursiere *et al.* [88] presented preliminary results on the detection of 7 guitar techniques with a hexaphonic pickup. The authors applied different signal processing and feature extraction operations to each technique. The resulting set of custom detectors showed a general detection success rate of about 90% across the different techniques, which the authors describe as proof-of-concept results. They also proposed an embedded hardware implementation that was designed around an FPGA component, however, not all the detection algorithms were adapted for real-time usage. Some of the authors went ahead to improve the system [89], reaching a 95% success rate, however, the algorithms for each technique could not be combined successfully and the results were obtained with the offline implementations.

Foulon *et al.* [90] presented an approach for interactive detection of guitar playing modes (i.e., melody, bass, and chords) with a Bayesian network classifier. The authors used a 2-step analysis process, where classification is first conducted on 50 ms frames yielding a reportedly "imperfect accuracy", while the classification results were aggregated over longer time periods to improve the accuracy of the prediction.

Abeßer *et al.* [91] developed an offline classifier for the detection of different playing styles on bass guitar recordings. The authors compared different conventional machine learning classifiers and obtained the best performance with a Gaussian Mixture Model.

Traube *et al.* [92] proposed a method for the estimation of the excitation point location on a guitar string. The authors based their work on the assumption that the power spectrum of a plucked string sound has a comb-filter shape. The plucking point estimation is performed using autocorrelation and iterative least-square estimation.

Penttinen *et al.* [93] proposed a time-domain method for estimating the plucking point, similar to the previous study.

Kehling *et al.* [94] developed a tablature transcription algorithm that classifies the string played, the plucking style (i.e., Finger, Picked, Muted), and the playing technique used. The authors used a multi-class Support Vector Machine (SVM) with a Radial Basis Function kernel and the classification accuracy obtained for playing techniques was 83%.

Barbancho *et al.* [95] proposed a technique for the detection of chord sequences and fingering configurations from guitar recordings.

Additionally, Barbancho *et al.* [96] approached the problem of detecting the pitch direction (i.e., ascending or descending) of arpeggio chords with feature extraction and conventional ML techniques. A Fisher linear discriminant algorithm and a Support Vector Machine (SVM) were compared, revealing comparable performances, with a slightly better performance provided by the SVM.

Chen *et al.* [97] presented a two-stage algorithm for detecting five guitar playing techniques in guitar solos. Although innovative, the method showed limited performance, with an average F-score of 74% across all the techniques.

Lostanlen *et al.* [82] approached the complex problem of instrumental playing technique detection with a wide range of instruments. The authors treated this problem as an extension of the musical instrument recognition problem.

Su *et al.* [98] investigated the use of sparse coding for the task of guitar playing technique classification. The results obtained in the comparison between many different feature combinations were promising for sparse coding, however, the best average F1-score obtained across 7 common techniques was 71.7%.

Part of the authors went ahead and improved the playing technique classifier, which was included in a broader transcription context in [99]. The new version used a neural network for classification (i.e., a convolutional model) and reached an

average F-score of 79.72%. As the previous iteration, the algorithm proposed works in an offline context, on pre-recorded monophonic guitar excerpts.

Percussive Fingerstyle Studies

Percussive fingerstyle is a set of techniques for acoustic guitars that involve the use of the body of the guitar as a percussive instrument to accompany a solo performance. The technique dates back to several decades ago, but it witnessed an increase in popularity in recent years. An interview study of the style was presented by Martelloni *et al.* [100] who observed percussive fingerstyle guitarists in different conditions and investigated possible guitar augmentation that could benefit their performance. The study resulted in a map of the locations of the most common percussive interactions and various common patterns in the needs of the players. These needs include a way to increase the dynamic range of the percussion sounds and the desire to have separate processing paths for different types of interaction. This suggests that this playing style may pair well with real-time classification of the percussion area used, which can be used to trigger substitute sounds (synthesized or sampled) or to control the processing of the guitar signal.

The authors went ahead to implement an augmented guitar prototype [57] with a magnetic pickup for the strings and three piezoelectric transducers attached inside the body of the instrument. The authors developed a hit classification system with 2 output classes: one for kick-drum-like gestures (produced with the wrist) and the other for every other percussive gesture. The classification algorithm was developed using Max/MSP and it triggered different sound samples depending on which of the two classes was predicted. The system designed by the authors involves the modification of an acoustic guitar with additional transducers and the sound analysis algorithms are executed on an external computer.

The latest work by Martelloni *et al.* [58] proposed the combination of classification and embedding learning for continuous control over a modal synthesis engine.

2.4 Technology for real-time embedded audio deep learning

In recent years we have witnessed an increase in the computing power and features of commercially available embedded devices and Single-Board Computers (SBCs).

While simpler microcontrollers have often been used for the creation of guitar augmentations and SMIs (See Section 2.2), more powerful Single-Board Computers offer new possibilities for embedded audio processing and the use of machine learning.

Moreover, the growth in compute capabilities of embedded computers has fostered the creation of various embedded audio platforms such as Elk Audio OS [24], Bela [23], Prynth [101], Satellite CCRMA [102], and Axoloti[2]. Nevertheless, their applicability to our embedded real-time recognition project was not clear due to the lack of information on deep learning inference engines for real-time audio tasks. Here we will briefly present relevant embedded audio platforms and inference engines.

2.4.1 Embedded Audio Platforms

Among recent embedded audio platforms, Elk Audio OS [24] and Bela [23] stand out as the most prominent and versatile systems within the open-source domain. Both are based on the Xenomai Linux real-time kernel, which allows for submillisecond round-trip audio latencies

Bela

Bela is an open-source embedded platform for sensor and audio processing [103]. It uses a Beagle Bone Black[3] single-board computer with a custom hard real-time audio environment. The Beagle Bone Black contains a 1 GHz ARM processor, 512 MB of RAM, and 4GB of onboard memory. storage. The BeagleBone also includes two Programmable Realtime Units that are used to move audio and sensor data to memory.

The Elk Audio Operating System

Elk Audio OS [24] is a Linux operating system based on the Xenomai real-time kernel developed by the *Elk Audio AB* company[4]. Elk Audio OS is optimized for low-latency audio processing on a wide range of embedded computing devices. In particular, Elk Audio OS comes with the headless DAW Sushi that can host several types of audio plugins, such as the popular VST 2 and 3 formats by Steinberg. As a result, the development of embedded audio processing software is simpler than with alternatives

[2] http://www.axoloti.com/, https://github.com/axoloti/axoloti
[3] http://beagleboard.org/black
[4] https://www.elk.audio/

such as DSP units or Field Programmable Gate Array (FPGA). Additionally, the use of the Xenomai kernel enables quick and predictable audio processing.

As a side-effect of using both Xenomai and a regular Linux kernel, Elk Audio OS provides tools to identify whenever costly or forbidden code operations are executed from the real-time processing thread, as they cause the OS to switch from the high-performance kernel to the accompanying Linux kernel. This will be further discussed in Chapter 7.

The open-source version of Elk Audio OS is supported for the Raspberry Pi 4 SBC, which features a Quad-core Cortex-A72 (ARM v8) 64-bit processor with a clock speed of 1.8GHz, and LPDDR4 memory, with different versions going from 1GB up to 8GB. Previous OS releases to version 0.10 support the RPi3 as well. Additionally, the company provides a commercial version of Elk Audio OS for the STM32MP1, NXP i.MX8M, Raspberry Pi Compute Module 4, and any Intel x86 SOC.

Elk Audio OS was chosen as the platform for most of the works presented in the thesis for the ease of development of optimized C++ code, the availability of a fully free distribution, and the possibility of running the operating system on the Raspberry Pi 4, which is rather powerful in comparison to previous versions or the hardware platform of Bela. A final benefit of using Elk Audio OS is that the plugins compiled for the system can rather easily be compiled for many other platforms, including Windows, Linux, and Mac OS computers.

2.4.2 Deep learning Inference Engines for real-time Audio

Deep learning inference is the process of computing every operation that is part of a neural network, in order to process inputs and produce an output prediction. Inference differences from network training in that it does not involve learning model weights. The execution of inference can be made with the same high-level programming languages with which models are typically trained (e.g., Python), but the real-time and embeddability requirements of this project ask for more optimized execution. In research, it is not uncommon to find examples of bespoke `C++` implementations of specific neural networks to enable quick inference. One example is the work by Wright et al. [104], who introduced a neural approach to audio effect modeling. The authors implemented from scratch a real-time inference routine using `C++` and the Eigen linear algebra library. However, this specialized practice leads to the development of inference routines that are hardly flexible. In particular, any simple change to the neural model would require modifying the code routine.

A more common approach used in embedded, mobile, and edge computing is that of inference engines (IEs). These are inference libraries that are optimized for fast inference and flexibility, allowing developers to load model structure information and weights from either a file or a byte stream. Some of the most popular IEs are TensorFlow Lite (TFLite) [105], torchlib+TorchScript [106], and ONNX Runtime [107].

However, at the time this project started, we witnessed a general confusion around the compatibility of these IEs with real-time audio applications, which require that the code does not contain any non-real-time-safe operation that can slow down the processing of the audio signal. Additionally, we acknowledged the existence of small open-source projects for inference engines that are specific to audio: Chowdhury [108] developed RTNeural, which is an IE designed to be used for real-time audio applications. The author compared the performance of the library against the regular PyTorch `C++` API (i.e., TorchScript). However, the landscape of available IEs has evolved since then, especially with the availability of the aforementioned "lite" optimized versions of most frameworks. Such confusion and lack of comparisons between IEs for audio led us to further investigate the performance of the aforementioned four IEs for embedded real-time audio classification (Chapter 5).

2.5 Summary

Despite the existence of various augmented guitars and some smart guitars, we found a lack of projects that integrated real-time expressive guitar technique recognition and embedded execution. Furthermore, despite the growing interest in deep learning audio analysis, we found scarce availability of open projects that performed **real-time** audio analysis with **deep learning** on **embedded computers**. Additionally, we found that information about the use of deep learning inference engines for real-time audio was lacking. Finally, as a consequence of the previous, guides for deep learning deployment for embedded audio platforms were, to the best of our knowledge, non-existent at the beginning of this project. In summary, the pitfalls found in the state-of-the-art were the following:

1. **Limited Integration of Real-time Expressive Guitar Technique Recognition**: The existing augmented guitars and smart guitars lacked comprehensive integration of real-time expressive guitar technique recognition and embedded execution.

2. **Scarcity of Open Projects for Real-time Audio Analysis with Deep Learning on Embedded Computers**: Despite the increasing interest in deep learning audio analysis, we found a notable scarcity of open projects that target real-time audio analysis using deep learning on embedded computers. This limitation posed challenges for developers seeking to implement such technologies in resource-constrained environments.

3. **Lack of Information on Deep Learning for Real-time Audio**: The available information on the use of deep learning inference engines for real-time audio applications was insufficient. This gap in knowledge hindered the understanding and advancement of techniques necessary for implementing deep learning solutions in real-time audio processing scenarios.

4. **Absence of Guides for Deep Learning Deployment on Embedded Audio Platforms**: The absence of comprehensive guides or resources for deploying deep learning models on embedded audio platforms created a significant obstacle. Developers faced difficulties in navigating the intricacies of compiling libraries and adapting deep learning models for real-time inference on embedded systems.

These collectively highlighted the need for advancements in expressive guitar recognition. Additionally, they point out the need to foster open projects for real-time audio analysis on embedded systems, the improvement of available information on deep learning for real-time audio, and the development of practical guides for deploying deep learning on embedded audio platforms. Addressing these challenges would contribute to the evolution of augmented musical instruments and hopefully foster broader applications in the realm of embedded real-time audio analysis and processing.

Chapter 3

Challenges of Embedded Real-time Music Information Retrieval

Real-time applications in the field of MIR are garnering increased attention, especially with the widespread adoption of deep learning for music analysis. However, incorporating deep learning into real-time MIR systems presents challenges and limitations that need to be effectively addressed to achieve accurate and speedy real-time performance. Additionally, modern embedded computers offer promising opportunities for compact MIR systems, such as DMIs. However, these embedded computing platforms are typically resource-constrained, posing additional limitations to this endeavor. In this chapter, we identify and discuss the challenges and limitations associated with embedded real-time MIR. Furthermore, we present potential solutions to these challenges. We then demonstrate the viability of our proposed solutions through the development of an embedded real-time classifier designed to recognize expressive acoustic guitar techniques. Our classifier achieved 99.2% accuracy in distinguishing pitched and percussive techniques and an average accuracy of 99.1% in distinguishing four distinct percussive techniques and an additional class for pitched sounds. The full classification task is a considerably more complex learning problem. In this case, our preliminary results reached only 56.5% accuracy. The results were produced with an average latency of 30.7 ms.

The work described in this chapter is part of a contribution presented at the 25$^{\text{th}}$

International Conference on Digital Audio Effects [109].

3.1 Introduction

MIR is a specialized research field that centers on analyzing and extracting information from music. Key tasks in MIR encompass beat-detection [110], onset detection [111], music transcription [112], and genre classification [113]. MIR research often focuses on offline methodologies, dealing with extensive datasets and not imposing strict execution time constraints. However, the sub-domain of real-time Music Information Retrieval (rtMIR) has gained notable traction in recent times due to its potential in developing music performance tools like digital musical instruments [114, 115], including smart musical instruments [25].

rtMIR presents unique challenges and constraints not present in offline contexts, mainly due to the rigid requirements on algorithm execution time and system latency in real-time systems [116]. These challenges are deepened by the computational demands of deep learning models, which, while offering high accuracy, can strain computational resources. Furthermore, embedded devices and single-board computers have emerged as powerful tools for compact learning and music systems [117–119]. However, their restricted computational capabilities pose additional hurdles for rtMIR.

In this chapter, we identify and discuss the challenges of developing embedded rtMIR systems dedicated to musical audio signals. We present a range of solutions to address these challenges. Moreover, we demonstrate the validity of select solutions by implementing an embedded real-time expressive technique classifier for acoustic steel string guitars. We focus on CPU-based embedded hardware such as single-board computers, which can run an operating system. This excludes simpler devices such as microcontrollers. In particular, we identified the following four core challenges:

1. **Causality**, or availability of only causal information in online/real-time environments;

2. **Tradeoff between accuracy and latency**;

3. **Audio processing deadlines and real-time-safe programming**;

4. **Embedded hardware and software limitations**.

The remainder of the chapter is organized as follows. Sections 3.2, 3.3, 3.4, and 3.5 present the four challenges above, accompanying each discussion with proposed solutions and viable tradeoffs. Then, in Section 3.6 we demonstrate some of the proposed solutions in a real-world use case: the implementation of an embedded real-time classifier for expressive playing technique recognition on the acoustic guitar. Finally, we summarize our findings in Section 3.7

3.2 Challenge 1: Availability of causal information only

The primary distinction between offline and rtMIR systems lies in the possibility, or lack thereof, to analyze the complete input signal at any given moment. In offline systems, this translates to the ability to collect extensive data around points of interest in the input signal, enabling the acquisition of a wealth of information. In particular, recorded sound events can be analyzed in their entirety. In contrast, at any point of execution, rtMIR systems can process only the current and past inputs. As a result, any type of real-time analysis of the input of such systems will report its results with a certain delay (i.e., latency). However, it is imperative for rtMIR systems to comply with a target latency that depends on the application, so as to be able to coherently use the analysis results for any application-defined objective (e.g., real-time sound generation).

In rtMIR systems, part of the latency between input and results is determined by the execution time of the analysis algorithms involved, but a large part is also played by having to wait for part of the input signal to be collected in order to be analyzed. The latter is a direct consequence of the sole availability of causal information in real-time. For instance, music event classification from an audio signal can only happen with a non-zero delay from each event (i.e., latency) due to the necessity of collecting a part of the signal (i.e., window) for analysis. Since increased input information regarding a sound event can enhance classification accuracy, similar rtMIR systems need to be configured to strike a balance between accuracy and the latency of result generation. The same holds for non-classification rtMIR systems (e.g., regression [120]) and more general audio tasks (e.g., environmental sound recognition [117]). This will be discussed in detail in part of the next section.

Furthermore, some methods or algorithms employed in MIR are exclusively applicable to offline scenarios, like Bi-directional Recurrent Neural Networks (BiRNNs).

These neural networks, for instance, have demonstrated success in tasks such as offline music-genre classification [121] and onset detection [122]. However, they are designed to leverage information from both past and future contexts, making them unsuitable for rtMIR.

Additionally, certain offline algorithms are incompatible with real-time systems due to their need for the entire audio signal. An example of this is the whitening technique, which involves dividing the magnitude of each bin of Short-time Fourier transform (STFT) spectrograms by the overall maximum value for the bin. While whitening has proven effective in enhancing the performance of onset detectors in various scenarios [123], it relies on non-causal information and is unsuitable for real-time onset detection. An overview of the real-time capabilities of specific onset detection methods mentioned in this context was provided by Bock et al. [124].

3.2.1 Potential solutions

It is clear how the availability of only past and current input information is a core limitation of rtMIR. As so, this limitation cannot be overcome, however, it has to be taken into account in the design of such systems.

First of all, whenever the specific task of a rtMIR system allows for a tolerable degree of latency, the signal analysis can be intentionally delayed to provide the analysis algorithm with a section of the input signal. For instance, in the expressive technique recognition algorithm described in Section 3.6, the feature extraction window was set to be as large as possible, while fitting in the maximum target latency.

Nevertheless, certain algorithms are unsuitable for real-time applications by definition. These algorithms necessitate the collection of complete signal information in advance, as exemplified by operations like the aforementioned whitening. In numerous instances, suitable approximations of these operations can be used as replacements. These approximations should depend solely on the portion of the signal accumulated up to each analysis point. An illustration of this is the *adaptive* whitening technique for onset detection introduced by Stowell *et al.* [123]. Adaptive whitening is integrated into the Aubio onset detector suite used for the classifier at Section 3.6. Likewise, some operations within neural networks are exclusively suitable for offline MIR applications. A prime example is the bidirectional information flow in BiRNNs [121, 122]. Bidirectional layers must be replaced with unidirectional Recurrent Neural Networks (RNNs) in real-time scenarios. This strategy was employed by Bock et al. [125], who introduced an effective real-time onset detector derived from a previously established

method designed for offline contexts.

The following section will present how the necessary delay introduced by waiting to collect input information can be tuned to find a tradeoff between latency and accuracy of rtMIR systems. Moreover, it discusses the additional aspects that can affect both accuracy and latency.

3.3 Challenge 2: Tradeoff between accuracy and latency

In rtMIR systems, a balance is struck between the precision of analysis results and the associated latency. This compromise arises from two key factors: the availability of only past information (causal) introduced in the previous Section, and the relation between the accuracy and execution time inherent to many analysis algorithms, especially machine learning and deep learning techniques.

As discussed in the previous section, rtMIR systems have the capacity to process only past and current inputs at any point in time. For this reason, their computations are intentionally delayed in time to accumulate a signal window. Such windows begin at a point of interest in time and end when the computations that constitute the analysis start. In this context, increasing the size of the analysis window will allow the capture of a larger portion of the sound of interest. In many cases, the availability of a larger portion of audio can directly lead to enhanced accuracy in the results of the analysis. For instance, a larger window in systems such as sound event recognition allows a rtMIR algorithm to take into account a more substantial portion of the temporal envelope and spectral content of the sound, thereby improving the quality of the features that can be extracted and reducing the likelihood of false positives. In pitch tracking, a larger window will enable a correct estimation of the pitch of lower-frequency sounds. It is worth noticing, however, as a larger window will only be useful if it includes more of the signal portion of interest, and not preceding or following portions.

At the same time, delaying signal analysis introduces latency between the input and the generation of results. This latency must be managed within specific tolerances for different tasks and applications. For instance, a system designed to respond to a sound by generating new sounds should maintain a latency of around 30 ms or less. This is because complex tones separated by intervals shorter than this delay may be perceived by the human auditory system as simultaneous [21].

In addition to the deliberate delays that can be introduced to collect more input information, the execution time of rtMIR algorithms plays a crucial role in the overall latency of real-time systems. This is particularly critical when considering modern machine learning and deep learning approaches, which are often optimized for achieving the highest accuracy in scenarios where execution time constraints are not imposed (e.g., offline MIR). While it is possible to reduce the execution time for most of these algorithms, doing so typically comes at the cost of a diminished result quality. For example, the execution time of inference using a K-nearest neighbors (KNN) classifier may be high when dealing with a large number of training samples, but it can be mitigated by removing a certain number of samples from the dataset. However, this operation effectively diminishes the amount of information that characterizes the output classes, therefore affecting the accuracy of the system. Conversely, for deep learning algorithms, the reduction of training data has no impact on inference time. Still, execution time can be significantly reduced by either decreasing the depth and width of neural network models or reducing the precision of the network's weights (weight quantization). However, any of these operations inevitably leads to a reduction in result quality.

3.3.1 Potential solutions

As mentioned in previous sections, signal analysis can be delayed up to a maximum tolerable latency. This is in itself a tradeoff since it involves forgoing a potentially high accuracy to achieve a specific target latency.

In the context of a deep classifier, there are several methods to reduce execution time. These include making the neural network shallower, or narrower, or reducing the precision of the network weights through weight quantization. The first two methods necessitate retraining the neural network and directly impact execution time by reducing the number of computations required. In our classifier, we adjusted the size of the classifier network to meet the target latency (as detailed in Section 3.6.3).

Weight quantization, on the other hand, is applied to trained models and involves converting floating-point weights into fixed-point values. This significantly reduces execution time due to the reduced computational cost of fixed-point operations. However, it also diminishes the accuracy of neural networks. The extent of this impact on results depends on the structure of the neural network, its size, and its overall resilience to weight quantization. In our expressive technique classifier, weight quantization led to a significant reduction in result accuracy (ranging from 10% to 20%,

depending on the specific quantization techniques). Consequently, we opted not to use weight quantization in the final models.

3.4 Challenge 3: Processing deadlines and real-time-safe programming

Real-time audio software handles audio signals using buffers. These audio buffers have a predetermined size in samples and are delivered to the software at a rate determined by the audio *sampling rate* and the *audio block size*. Consequently, these systems have a limited amount of time to process each input buffer before the subsequent buffer arrives. This constrained "time budget" is especially critical for systems responsible for generating an output audio signal. The time budget in seconds is simply obtained with Equation 3.1.

$$T = \frac{Audio\ block\ size}{Sample\ rate} \quad e.g., \quad \frac{64\ samples\ per\ block}{48,000\ kHz} = 1.33 \times 10^{-3}\ s \tag{3.1}$$

In the case of multichannel processing (e.g., stereo signals), the time budget will be computed with respect to the number of frames per audio block, i.e., the number of tuples of samples, with each tuple containing a sample from each channel. Any inability to complete all computations within this time window can lead to audible glitches in the output, which are directly caused by the presence of discontinuities in the output buffer left by an incomplete write. Such an event, caused by the consumption of the buffer happening at a faster pace than the writing operation, is referred to as buffer underrun (or Xrun) [126]. For rtMIR systems operating on input audio signals, a portion of the analysis computations must be executed at the same rate as audio processing. This means these computations will consume a portion of the available time budget. This also applies to rtMIR systems handling input audio signals.

Lastly, in real-time audio programming, developers must adhere to using operations that are guaranteed to complete within a known and bounded time (i.e., real-time-safe operations[1] [127]). This ensures that buffer underruns do not happen, as long as the sum of the execution times of every operation is smaller than the time budget available. In particular, operations such as dynamic memory allocation, reading data from disk, locking operations, inter-thread communication, and some system

[1] http://www.rossbencina.com/code/real-time-audio-programming-101-time-waits-for-nothing

calls are not real-time-safe, as their execution time depends either on the availability of hardware peripherals or unbounded computations on other threads. It is worth noticing that developers should not only avoid such operations in their code but also assess their absence in any external libraries employed. This can pose a challenge when integrating code designed for offline use into real-time audio applications.

3.4.1 Potential solutions

Time deadlines

In this context, there are several modes of real-time execution that allow for different solutions. The most strict constraint on execution time is posed by algorithms that need to run for each audio block that is received from the analog to digital converters. A prime example of this case is software audio effects. In this case, the entire algorithm execution should fit in the time budget available for each audio block. A first approach is to reduce the number of computations performed by the analysis algorithm, for example with the techniques mentioned in Section 3.3.1, while taking into account how this can affect the quality of the results. Additionally, the size of the audio buffer can be increased, therefore increasing the time between buffer read operations, but this will slow down every step of the analysis.

In contrast, many rtMIR algorithms can be subdivided into multiple stages of analysis, where some can run at a different rate of execution without affecting the time-budget available. For example, the expressive technique recognition pipeline presented in Section 3.6.3 is composed of an onset detector, a set of feature extractors, and a deep learning classifier (see Figure 3.5). The onset detector must run in the real-time thread of execution and will benefit from running at a high rate (i.e., small audio buffer size), which reduces the latency with which onsets are detected. Conversely, feature extraction is only necessary when an onset is detected, and the same principle applies to the computation-intensive deep learning classifier. Furthermore, for our specific task, it is reasonably assumed that there is no need to classify notes occurring less than 20 ms apart. Consequently, both the feature extraction operation and the execution of the classifiers are low-rate tasks. The onset detector was executed in the real-time thread with a buffer size of 64 samples at 48 kHz (i.e., onset detection is executed every 1.33 ms), while both the feature extraction and classification stages can exceed the time budget of the audio processing routine (i.e., 1.33 ms), and their execution can be moved to a separate lower-priority thread. A depiction of the flow

of execution of the classifier is displayed in Figure 3.1

However, this separation strategy is not applicable to end-to-end neural networks. In fact, an underlying trend in deep learning is to develop end-to-end neural networks capable of directly operating on raw data and generating results without the need for additional algorithms. While this approach allows neural networks to learn "internal" features that often outperform hand-crafted features in terms of descriptor quality, it results in large indivisible models.

Different modes of execution and optimization techniques for neural networks are further discussed in Chapter 7, Section 7.5, which present solutions in a broader context than rtMIR, including for example audio processing algorithms.

Real-time safety

In addition to optimizing the execution time of the code developed for a rtMIR system, it is crucial that developers use only real-time safe operations in audio processing threads. As mentioned before, safe operations are those that will complete execution in a known and bounded execution time. These constraints exclude operations such as dynamic memory allocation, for which standard implementations may cause the real-time thread to wait on lower-priority threads that use the same data (i.e., priority inversion). In more critical situations, the memory allocator may have to ask the operating system for more memory. Additionally, memory can be subjected to paging, where chunks of memory are temporarily stored on disk and retrieved on demand, at a higher cost. Ultimately, standard memory allocation will take an unpredictable amount of time to allocate a block. Developers should avoid this by allocating statically most of the data structures needed, dynamically allocating data only from a non-real-time thread, or pre-allocating a memory pool and implementing a non-locking memory allocator for cases that require dynamic allocation. Similar operations to avoid using in the real-time thread are any locking operation such as reading data from disk. However, as much as developers can avoid such operations in their own code, it is also crucial to use only libraries and data structures that strictly adhere to real-time safe operations. For example, if used improperly, the `std::vector` data structure from the `C++` Standard library will perform memory allocation whenever the preallocated memory is full and the `push_back` operation is used to append an element at the end of the vector. Our expressive technique recognition system uses the real-time onset detector from the Aubio library [128], and the feature extractors are `C++` adaptations of tools from the TimbreID library [27].

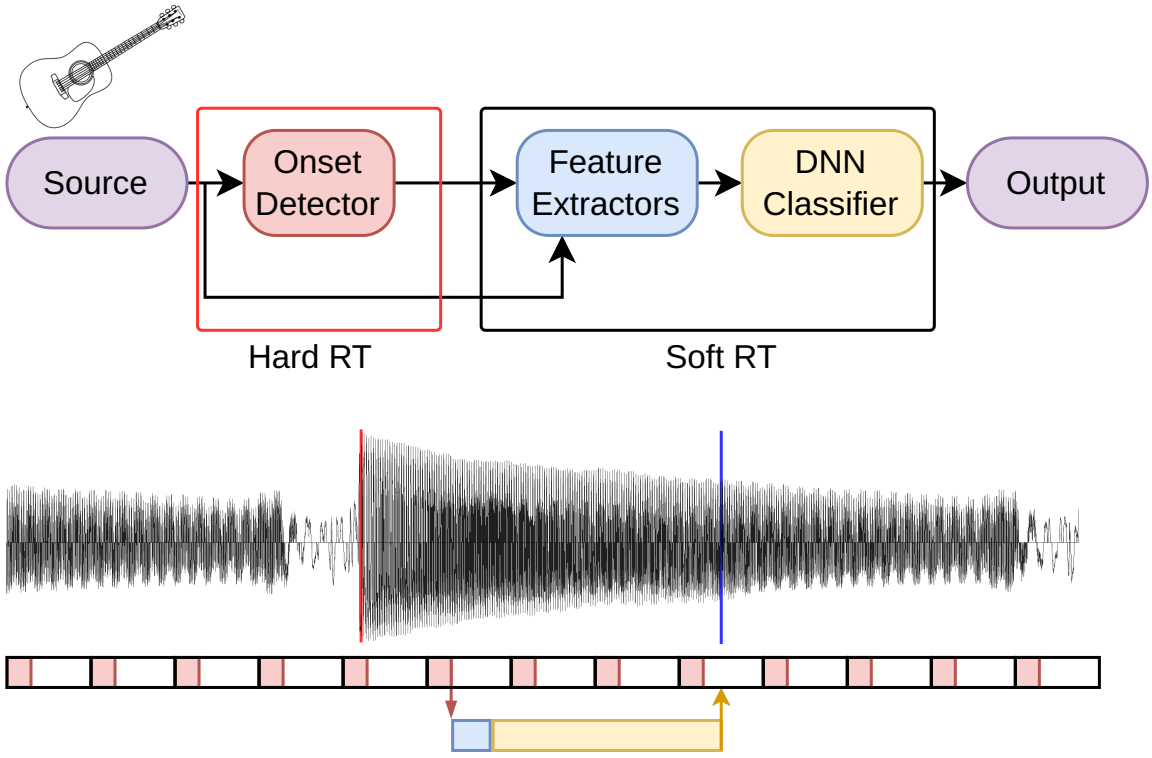


Figure 3.1: *Flow of execution of the classification pipeline presented in Section 3.6. The upper part of the image represents the classification pipeline, which is composed by an onset detector, a set of feature extractors, and a deep classifier. The lower part of the image displays a note in the audio signal and a set of adjacent rectangles with black borders, each representing the time slot to process each audio block. Each red rectangle represents the computations performed by the onset detector, which is updated at each block for low latency detection. Conversely, the cyan and yellow rectangles represent feature extraction and classification respectively, which are triggered whenever an onset is found, instead of being executed for each block. Executing the whole pipeline or end-to-end neural network for each audio block would require an increase in the audio block size, therefore increasing the latency of the entire system and any other audio processing task performed on the same device.*

Furthermore, there are several deep learning execution libraries available that enable quick neural network inference without compromising real-time-safe programming principles, including *TFLite*, *libtorch with Torchscript ONNX Runtime*, and *RTNeural* [108]. A more detailed comparison of deep learning IEs was carried out, and the results are reported in Chapter 5.

3.5 Challenge 4: Embedded hardware and software limitations

The performance of any rtMIR system is dependent on the limitations of the computing hardware used, along with the low-level software (e.g., OS) of its target computing device. Contemporary personal computers are typically powerful and equipped with hardware optimized for high-throughput operations such as Graphics Processing Units (GPUs). In contrast, embedded computers are often characterized by limited resources, which pose numerous challenges for real-time audio and rtMIR applications.

For many MIR tasks, deep learning algorithms have replaced traditional machine learning and simpler heuristics due to their superior performance in terms of result quality [112, 122, 125, 129]. However, neural networks tend to be more computationally complex than their simpler counterparts, increasing the number of operations to perform in real-time. Furthermore, embedded computers are generally resource-constrained devices when compared to general-purpose PCs. In particular, embedded computers usually have reduced CPU speeds, core count, RAM amount, and speed in comparison with regular PCs. Moreover, very few embedded computers are provided with deep-learning-enabled GPUs (e.g., supporting Nvidia CUDA drivers) or similar acceleration hardware for deep learning inference, such as Tensor Processing Units (TPUs).

Some embedded music systems use a paradigm where audio and/or sensor data is collected on-site by an embedded device, and processed in a separate powerful cloud server [130]. However, this introduces high communication latencies that are not desirable in real-time systems.

3.5.1 Potential solutions

Training and running modern deep learning models can be computationally demanding. On powerful PCs, both training and inference tasks are accelerated through the use of GPUs, which are specialized hardware designed for efficient parallel processing. However, the majority of embedded computers lack dedicated hardware support to accelerate deep learning inference, let alone training. Notable exceptions include the Nvidia Jetson family of embedded computers, which offer an integrated GPU that supports deep learning acceleration through Nvidia's CUDA drivers.

Additionally, various low-power processing units for efficient deep learning inference have emerged, including Google's Coral TPU and Intel's Visual Processing Units (VPUs). These processing units are available in various forms, including embedded computing boards like the Coral Dev Board[2], as well as external devices that connect to the CPU of embedded computers via USB, such as the Coral USB Accelerator[3] and the Neural Compute Stick[4].

Furthermore, the more versatile nature of FPGAs has garnered increasing interest with regard to deep learning inference. This is due to the creation of new tools for converting neural networks into FPGA-compatible code. Vandendriessche et al. [117] recently conducted a performance comparison of various embedded options for environmental sound recognition, including TPUs and FPGAs. Their study focused on Convolutional Neural Networks (CNNs), a type of neural network that demands a high degree of parallel processing. FPGAs have also been explored for ultra-low latency DSP. Specifically, Risset et al. presented initial results on the development of a tool capable of compiling Faust code for FPGAs [119]. Additionally, Wegener et al. recently described a method for interfacing `Pd` with an FPGA to achieve low-latency physical modeling synthesis [131].

Nevertheless, single-board computers have witnessed significant advancements in recent years, enabling them to handle relatively small neural networks like compact Feed-Forward Neural Networks (FFNNs) using just the CPU. Additionally, it is worth noting that each of the acceleration hardware solutions mentioned previously introduces varying degrees of overhead due to their communication with the CPU. This overhead may present challenges for extremely low-latency real-time systems.

In addition to the hardware alternatives mentioned, embedded rtMIR systems

[2] https://coral.ai/products/dev-board/
[3] https://coral.ai/products/accelerator
[4] https://www.intel.com/content/www/us/en/developer/tools/neural-compute-stick/overview.html

can experience significant improvements through the use of optimized real-time audio platforms [116]. Among the most advanced options are the Bela platform [23] and the Elk Audio OS [24]. Both of these solutions are built upon the Xenomai Cobalt real-time kernel, which enables the achievement of extremely low round-trip audio latency. Additionally, Vignati et al. [126] recently compared the performance of the Xenomai kernel and the more accessible PREEMPT_RT patch for the Linux kernel. Their findings indicated that Xenomai-based solutions offer superior performance, but PREEMPT_RT remains considerably easier to implement while still delivering good performance. In summary, these low-level software solutions empower embedded rtMIR systems to harness more CPU performance and reduce latency compared to the standard Linux kernel. Our expressive technique classifier is deployed on a Raspberry Pi 4 computer running the Elk Audio OS (for the reasons discussed in Section 2.4.1), and deep neural inference is executed on the CPU due to the considerations mentioned earlier.

3.6 Expressive Guitar Technique Classifier

To illustrate some of the potential solutions outlined in the previous section, we developed a real-time embedded classification system capable of recognizing the expressive playing techniques employed by an acoustic guitar player. The system is designed for monophonic settings, where the musician plays one note at a time, similar to a guitar solo performance. This is a notable restriction that was, however, found to be needed, as the collection of polyphonic recordings and onset labeling process was found to be considerably more complex than monophonic signals. Moreover, polyphonic onset detection from a single audio signal is rather complex. However, polyphony could be tackled with a hexaphonic separated pickup, and 6 separate onset detectors. Additionally, a single classifier could be trained on the mix of the six channels to retrieve the predominant technique, or 6 separate classifiers could be employed, although at a higher computational cost.

We performed the detection on a steel-string acoustic guitar, a versatile instrument that enables a wide range of techniques, including percussive techniques, which are drum-like sounds produced by striking the instrument's body. Real-time identification of these percussive hits can be utilized for purposes like triggering drum samples or controlling percussive synthesis algorithms during a performance. Our classifier was deployed on a Raspberry Pi 4 embedded computer running the Elk

Audio OS. The source code for the expressive guitar technique classifier is available online[5]. In particular, the classifier includes a deep neural network to predict the playing technique from lower-level timbral features. Figure 3.2 depicts the hardware setup of the classification system. In previous efforts, we employed simpler machine learning techniques, such as KNN [132], which however yielded lower accuracies and showed to be more subject to misclassification in the presence of feature noise.

Figure 3.2: *Hardware setup of the expressive technique classifier. The audio signal is acquired from the guitar pickup and channeled to a Raspberry Pi via the Elk PI Hat, which is an audio interface with analog-to-digital and digital-to-analog converters. The incoming audio is processed by the classification pipeline, and the result of the classification is conveyed through the pitch of a short tune produced in the output. Monitoring the classifier with one side of the headphones allows players to assess the accuracy of the result and whether the sound is produced with a perceivable delay from the input notes. The size of the computing device allows it to be embedded in the guitar body, and even smaller peripherals can replace the current audio hat for a reduced footprint.*

The technique classifier was trained for three progressively complex tasks, as outlined in Section 3.6.1. Section 3.6.2 provides insights into the dataset employed for training the classifier, while Section 3.6.3 delves into the specifics of the classification pipeline. Lastly, Section 3.6.4 showcases the system's performance results in terms of accuracy and latency.

[5] https://github.com/CIMIL/cpp-timbreID/

3.6.1 Classification tasks

This study was structured around three classification tasks that incrementally vary in complexity:

1. **Task A** [binary]: The initial task is a binary classification scenario, where the neural network's output must determine whether a note was played using a **pitched** or a **percussive** technique.

2. **Task B** [percussive+]: The second task extends to multiclass classification, encompassing four distinct percussive techniques alongside a unified class for all pitched techniques, as further detailed in the next section.

3. **Task C** [full]: The third task encompasses the comprehensive classification problem, requiring the discrimination of twelve individual expressive playing techniques. This task presents a notably higher degree of complexity compared to Task A and Task B.

3.6.2 Dataset

The classification implemented for this study includes a deep neural network, therefore we required an audio dataset of expressive guitar techniques played on the acoustic guitar. In particular, we required recordings from a variety of acoustic guitars, each meticulously annotated with timbre labels for every played note. Moreover, the recordings needed to be monophonic and to contain a number of relevant playing techniques. Finally, as the final system must work without external microphones, the dataset had to be recorded through pickups embedded into the guitar. Given the scarcity of free datasets of guitar recordings complying with our specific requirements, we proceeded to define and record a new dataset.

Assisted by professional guitarists, we assembled a list of twelve distinct acoustic guitar techniques that include both percussive and pitched playing techniques:

1. *"Kick" technique* (**percussive**): producing a sound that resembles a kick drum by hitting the lower right part of the top of the guitar body;

2. *"Snare-A" technique* (**percussive**): producing a sound by hitting the lower right side of the guitar body;

3. *"Tom" technique* (**percussive**): producing a sound by hitting the area of the guitar body near the top of the end of the fretboard, using the thumb;

4. *"Snare-B" technique* (**percussive**): producing a sound by hitting the muted strings over the end of the fretboard;

5. *Bending technique* (pitched): pulling the strings, raising the pitch (half-tone interval);

6. *Hammer-on technique* (pitched): sharply bringing a finger down onto the fingerboard, creating a legato sound (half-tone interval);

7. *Natural Harmonics* (pitched): plucking the strings while lightly touching the string with the fretting finger (i.e., not pressing the string fully), therefore letting only some harmonic overtones ring;

8. *Palm Mute*: partially muting the strings with the palm of the picking hand, resulting in a muffled sound.

9. *"Pick Near Bridge"* (pitched): plucking the string near to the guitar bridge, producing sounds with great high-frequency content;

10. *"Pick Over the Soundhole"* (pitched): plucking the string over the soundhole, producing sounds with lower treble content and greater intensity;

11. *Staccato* (pitched): playing short notes;

12. *Vibrato* (pitched): Moving the fretting finger to warp the pitch and tone of the sound.

The choice of percussive techniques drew inspiration from the interview study on percussive fingerstyle by Martelloni et al. [100]. The specific areas of the guitar body used for each of these techniques are visualized in Figure 3.3. Hand positioning is shown in Figure 3.4.

The dataset was created in collaboration with five experienced guitarists, each using a different guitar, and recorded using the built-in pickups of their respective guitars. The percussive techniques were recorded by each guitarist at three different intensity levels (i.e., *piano*, *mezzoforte*, *forte*), with 10 repetitions for each level. For the pitched techniques, individual notes were played within specified key ranges[6]. Each key on each string was played three times for each of the intensity levels mentioned earlier. The onset detector indicated that the total recordings in the dataset

[6]Natural harmonics were recorded only for frets 5, 7, and 12, while the remaining pitched techniques covered a range from open strings up to a fret between the 15^{th} and 20^{th}, depending on the physical attributes of each guitar (e.g., cutaway, string gauge, guitar scale)

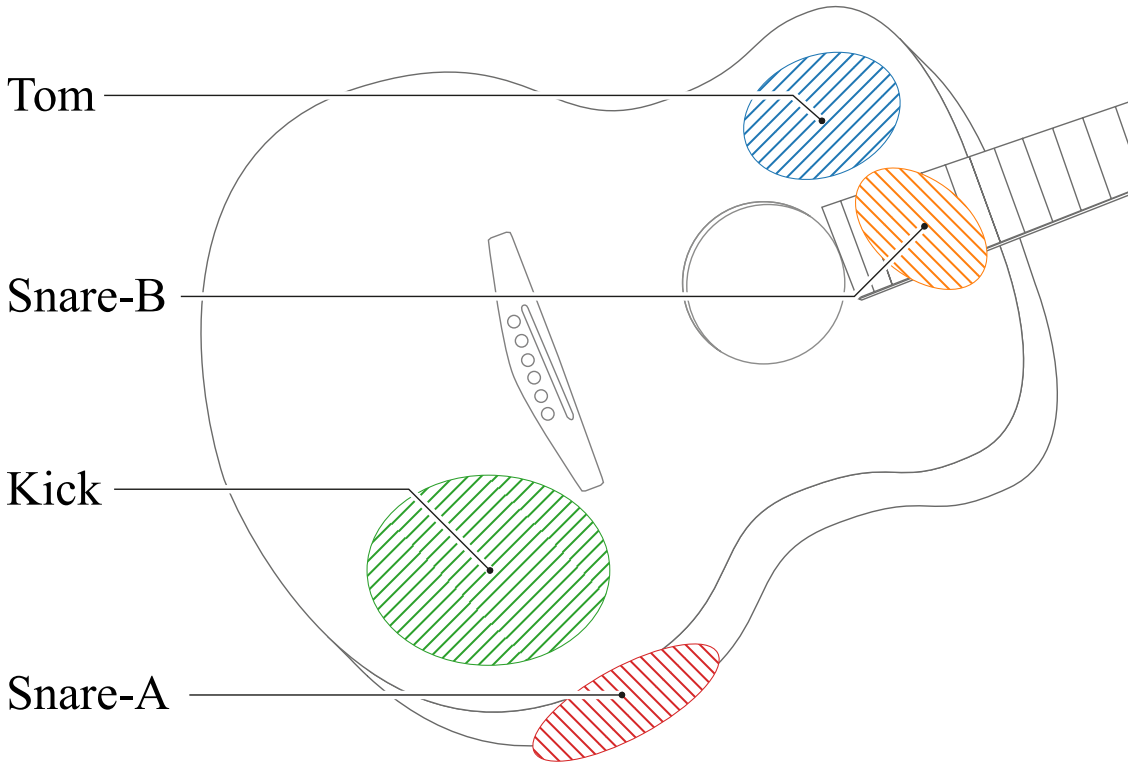


Figure 3.3: *Percussive techniques of choice, with the relative guitar areas used. The hand positions for each technique are illustrated in Figure 3.4.*

encompassed approximately 35,035 notes. A later version of the dataset, with more recordings and precisely labeled notes, was made freely available at [28].

3.6.3 Classification Pipeline

This section outlines the design and practical implementation of the classification pipeline for real-time expressive guitar technique recognition on embedded devices. Our chosen embedded platform is the Raspberry PI 4 (4 GB RAM version) with the Elk Audio OS based on the considerations discussed in Section 3.5.1. To meet the computational constraints of the Raspberry PI 4, we have adopted a multi-step classification pipeline (see Section 3.4.1). This pipeline consists of three main components: an onset detector, a set of feature extractors, and a deep learning classifier (refer to Fig. 3.5).

This approach enables us to distinguish between tasks that require a high refresh rate, such as onset detection, and those that can be executed at a lower frequency, like

Figure 3.4: *Hand position for the four percussive techniques in the dataset. Top row, from the left: Kick, Snare-A. Bottom row: Tom, Snare-B*

feature extraction and classification. Additionally, the classification process can be carried out on a separate lower-priority thread, decoupled from real-time execution. Consequently, the latency introduced between sound and results by the software system can be characterized as the sum of the latencies associated with each stage of the pipeline (see Fig. 3.6). In the following sections, each step of the pipeline will be described in detail.

Onset Detection

Onset detection involves identifying the starting point, i.e., the onset, of individual sounds in an audio signal. Unlike the technique classifier at the end of the pipeline, the onset detector must analyze the audio signal at a fast rate for low detection latency. This requirement limits the selection of detectors to those that operate quickly, precluding slower offline solutions like many deep learning approaches. We found *aubioonset* [128] to be reliable and able to detect onsets with a small latency interval.

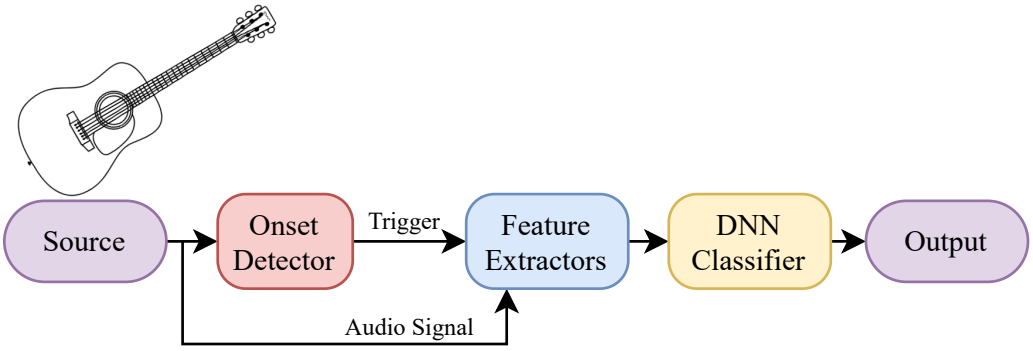


Figure 3.5: *Classification pipeline. From the left, the signal from the guitar pickups is fed to an onset detector, which detects the beginning of sounds in the audio signal. Whenever this happens, the onset detector triggers the computation of several timbral audio features from the audio signal. The trigger happens with a brief and deliberate delay from detection (i.e., post-onset delay) so to collect a larger part of the signal and align the beginning of the feature extraction windows with the onset. Then, a deep classifier is employed to predict a technique class from the timbral features. Finally, the resulting class determines the pitch of a short sinusoidal tone that is played in output.*

Feature Extraction

While it is feasible to design a neural network that directly processes the raw audio signal, its high dimensionality may be difficult to handle, and this can be counterproductive for discriminative tasks [133] where the output must be a lower-dimensional, high-level property. In these cases, an alternative approach is to use transformations that more effectively capture the relevant signal properties of interest. Since expressive techniques affect mostly the timbral content of notes (i.e., spectral content and temporal envelope) we employed several timbral feature extractors. In particular, we used extractors from the TimbreID library [27], which were ported from `Pd` externals to `C++` classes.

On the full dataset, the following feature extractors of the TimbreID library were used: Attack time, Bark Spectrum Brightness, Bark Spectrum, Bark Frequency Cepstral Coefficient (BFCC), Cepstrum, Mel Frequency Cepstral Coefficient (MFCC), and Peak sample. As introduced in Section 3.2.1, the feature extraction was delayed as much as possible while fitting into the target latency, resulting in a feature analysis window of 1,024 samples, which corresponds to 21.33 ms of audio at 48 kHz. To do so, the computation of features was delayed by a brief time interval after each onset detection (i.e., Post-onset delay), to align the beginning of the window with the note

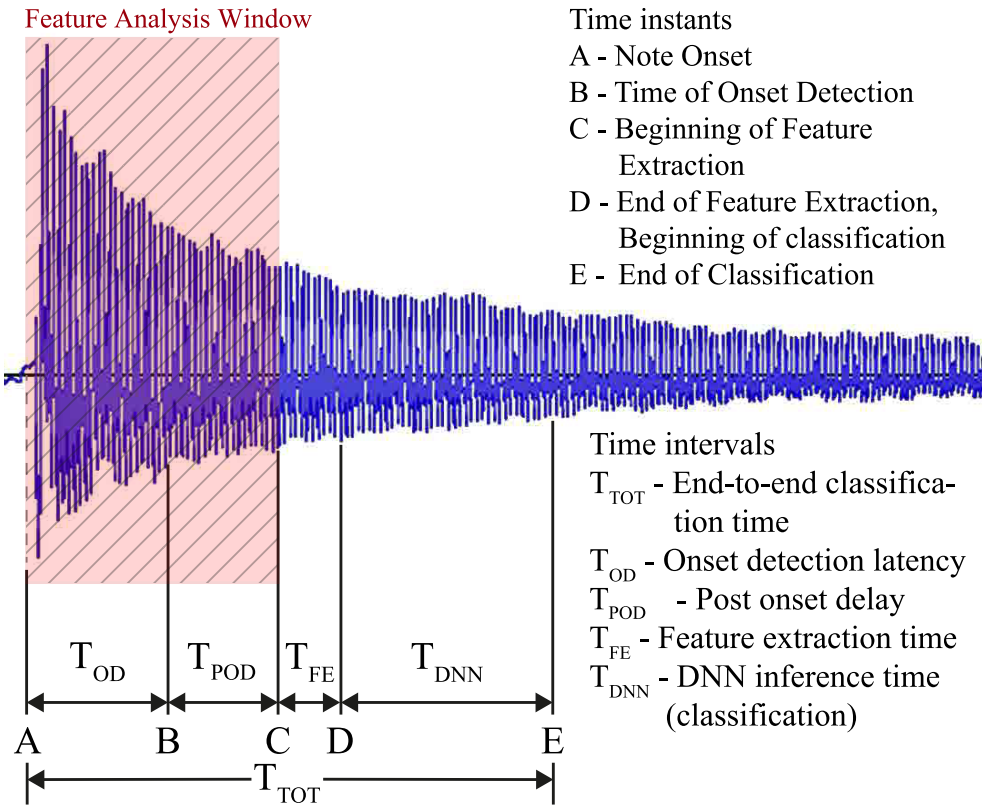


Figure 3.6: *Graphical representation of a guitar sound in the audio signal and all the components of the total latency between a note onset and its classification. T_{OD} is the latency of detection of the sole onset detector. T_{POD} is a deliberate delay introduced after the detection to align the feature extraction windows with the actual onset. T_{FE} is the actual time required to compute the features from the buffered audio in the analysis window, and finally, T_{DNN} is the time required by the deep classifier to produce a prediction.*

onset. For each classification task, a different feature subset was selected through a process of trial and error.

Classification

The expressive playing technique classification process involved the use of a neural network model loaded in TFLite. Each task required a different model to be trained. Each model was a feed-forward neural network with dropout and batch normalization. The development of each model followed an iterative trial-and-error approach, wherein we experimented with various combinations of input features, layer configurations, dropout probabilities, and optimization parameters.

For optimization, we found that the Stochastic Gradient Descent (SGD) optimizer

yielded the best results across all learning tasks. We used a learning rate of 1e−3 and experimented with momentum values ranging from 0.7 to 0.9. In terms of the loss function, Sparse Categorical Cross-entropy was chosen, as it is well-suited for multiclass classification problems.

To determine the appropriate model size, we measured the execution time of models with different configurations on the target platform. Model sizes were then adjusted to meet the specified target latency (as described in Section 3.3.1). Each neural network underwent training for a varying number of epochs, ranging from 1,000 to 2,000, using a 75% stratified random split of the original dataset. The following three neural networks were used for Task A, Task B, andTask C:

- *Network A* [binary]: Network A uses 110 input features (50 BFCC + first 60 Real Cepstrum features) and it is composed of four dense hidden layers (fully connected) with 500 neurons each. The final layer has 2 neurons. Each of the hidden layers used the LeakyRelu activation function, and they were interleaved by dropout layers for regularization, as suggested by Sigtia et al; [134].

- *Network B* [percussive+]: The model used for task B was similar to that of Task A, differing in the number of output neurons (i.e., five outputs for Network B). The first four outputs represent the prediction of one of the four percussive techniques, while the fifth corresponds to any pitched sound;

- *Network C* [Full]: Similarly, the same network of Task A and Task B was used for the full classification task, except for using 513 input features (i.e., all the Cepstrum coefficients) and having 12 outputs, one for each expressive technique.

3.6.4 Results and Discussion

Accuracy of the classifier

The success rate of the classifier was evaluated using key metrics, including accuracy, precision, recall, and F1-score. These metrics were computed on a stratified random split of 25% of the complete dataset, while the neural network models were exclusively trained on the remaining 75% of the data. The optimal results achieved through an iterative process for Task A, Task B, and Task C are detailed in Tables 3.1, 3.2, and 3.3, respectively. Additionally, Figures 3.7, 3.9 and 3.9 show the confusion matrices for the three tasks respectively, normalized over the rows.

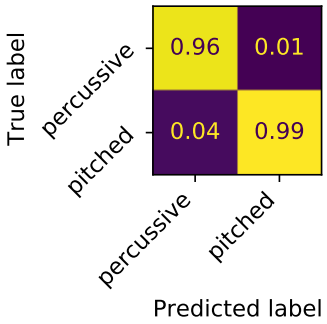


Figure 3.7: *Confusion matrix for Task A.*

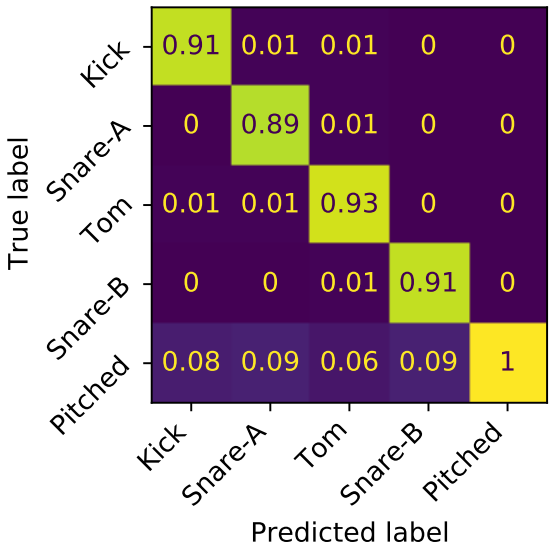


Figure 3.8: *Confusion matrix for Task B.*

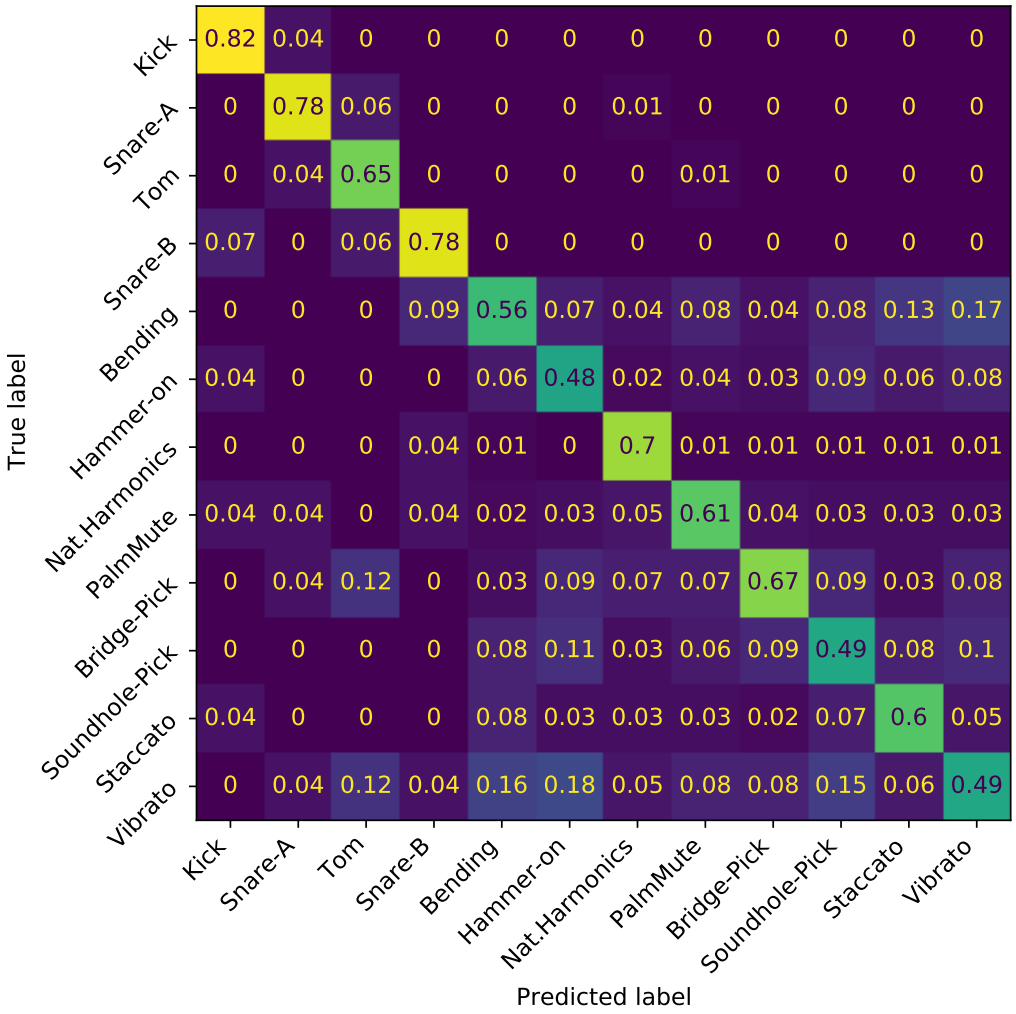


Figure 3.9: *Confusion matrix for Task C.*

Table 3.1: *Summary of the results of Task A.*

Class	**Precision**	**Recall**	**F1-score**
Percussive	95.6%	92.6%	94.1%
Pitched	99.5%	99.7%	99.6%
Macro avg.	97.5%	96.1%	96.8%
Accuracy			**99.2%**

Table 3.2: *Summary of the results of Task B.*

Class	**Precision**	**Recall**	**F1-score**
Kick	91.4%	95.5%	93.4%
Snare-A	89.3%	95.0%	92.1%
Tom	92.8%	91.8%	92.3%
Snare-B	90.9%	96.8%	93.7%
Pitched	99.7%	99.4%	99.5%
Macro avg.	92.8%	95.7%	94.2%
Accuracy			**99.1%**

These results show how both Task A and Task B can be tackled effectively with a rather simple FFNN method. A trial-and-error method was employed to determine the input features and classifier parameters. Notably, the classifier achieved an accuracy of 99.1% on Task B, surpassing previous results obtained with smaller feature vectors on the simpler task of percussive technique recognition (92.5% [132]). The performance of the classifier on task A shows how with as few as 21.33 ms of the attack phase of a guitar sound, a set of feature extractors and a simple FFNN can distinguish between percussive and pitched sounds. Such a model could be used as a first step of a hierarchical classification system, where different approaches can be adopted to further refine the classification, depending on whether the current sound is found to be percussive or pitched. Task B however, is an example of a complete system that could be used to trigger drum samples whenever a guitarist hits a percussive area. In this case, pitched sounds are classified rather accurately as the "pitched" class, so as not to trigger unwanted percussive samples. Conversely, Task C proved to be considerably more complex. The confusion matrix in Figure 3.9 clearly shows how pitched techniques are affected the most, while percussive techniques show higher, despite reduced, precision.

Here we compile a list of the most probable causes for the underwhelming performance of the classifier on the 12-classes task, which was derived from further reflection

Table 3.3: *Summary of the results of Task C.*

Class	**Precision**	**Recall**	**F1-score**
Kick	82.1%	76.7%	79.3%
Snare-A	78.3%	66.7%	72.0%
Tom	64.7%	39.3%	48.9%
Snare-B	78.3%	60.0%	67.9%
Bending	55.7%	42.8%	48.4%
Hammer-on	48.0%	48.8%	48.4%
Nat.Harmonics	69.7%	50.9%	58.8%
Palm Mute	61.1%	82.0%	70.0%
Bridge-Pick	67.2%	59.8%	63.3%
Soundhole-Pick	49.4%	52.6%	51.0%
Staccato	60.4%	71.2%	65.3%
Vibrato	48.5%	41.6%	44.8%
Macro avg.	63.6%	57.7%	59.8%
Accuracy			**56.5%**

on the system:

1. **Simplistic Technical Choices:** We deliberately adopted a simplistic approach to support the discussion on challenges and potential solutions entertained in the previous sections and [80]. However, this classifier served as a starting point for our expressive guitar technique recognition project. Nevertheless, the following technical choices should be more deeply investigated:

   (a) *Onset Detection:* The onset detector used in the pipeline was manually optimized, and its performance was evaluated on a small labeled subset of the dataset. As a result, the training set contained a few false positive detections. In particular, onset detection proved to be more difficult with pitched sounds, which caused a lower detection accuracy with those techniques. Another result of naive parametrization of the onset detector is that the variability of the detection latency was higher than desired, making it difficult to properly align the feature extraction window with the actual onset. As a result, in some cases, the actual onset would be excluded as the window started from a slightly later point in time, while in others the onset was properly included, but some pre-onset noise was too. Additionally, the post-onset delay was overestimated based on the

theoretical onset detection latency: using the actual measured T_{OD} would allow for better alignment of the feature window. Preliminary results at this point showed that a more proper parametrization that could yield a lower latency variability can improve the classification accuracy.

(b) *Feature Selection:* the small and efficient FFNN showed that feature selection was beneficial to the performance of the models. However, here we resorted to a manual trial-and-error selection process, which was only informed by the results obtained and previous research. A more proper feature selection mechanism can help the performance.

(c) *Single-dimensional description:* Due to the small size of the feature window (i.e., 1024 samples, 20.33 ms) we used the entire window as the buffer and hop size of most windowed feature extractors, resulting in a mono-dimensional feature vector for each note. This is convenient as simple and efficient FFNNs or 1D CNNs can be used to perform classification, but this cannot describe feature variation along the time axis. A more advanced approach would be to capture a number of smaller windows in the analysis buffer, therefore obtaining a 2D feature matrix for each note. Classifying such a feature matrix would require a different neural network, such as a CNN or RNN.

(d) *Neural network:* A more in-depth optimization of the neural network hyperparameters would yield better results. In particular, the addition of more regularization layers such as batch normalization showed promising results in later studies.

2. **Pitched Techniques:** The classification of all the pitched techniques in the dataset has proved to be significantly more complex than distinguishing the sole percussive sounds. This can be attributed to the higher variability in pitch, loudness, and arguably timbre that is possible with notes played on the string. In fact, the same technique played on an open string or the 17^{th} fret would retain only some of its sonic characteristics even to our ears, while pitch and timbre will change to different extents. As a result, the brief section of sound captured for classification at the beginning of notes may either (1) not be enough to distinguish complex pitched techniques or (2) require more refined technical tools, features, and machine learning approaches to extract such high-level information.

3. **Attack-based approach:** Due to the nature of the real-time system we had in mind when starting this project (i.e., an advanced real-time guitar synthesizer) we adopted a classification approach that focuses only on the very first milliseconds of each note, therefore its *attack* phase. However, not all expressive guitar techniques are attack-based. Namely, bending, staccato, hammer-on, and vibrato are techniques that produce their characteristic effect after the attack phase (or the first attack in the case of hammer-on). Properly classifying these techniques would require defining a new classification framework where attack-based classification can be later refined whenever a note morphs from a base technique to a bend, vibrato, or other.

Further considerations on the conclusions drawn from these points, and the directions followed after this study, are reported in the final summary of the chapter (Section 3.7).

Classification Latency

Measuring precisely the end-to-end latency of the classification pipeline would require feeding the embedded computer with a prerecorded guitar signal, and having the classifier produce a clear signal mark when classification is completed. Each test would require the recording of both the input and output signal and each onset and mark in the signal would need to be hand-labeled. This would allow for a precise measurement of the delay introduced by the software between onsets and classification results. Such a technique was employed in a later study (see Chapter 6).

For simplicity in this demonstrative case, we used an approximate measurement setup. First, the delay between each onset and its detection (Onset detection latency) was conducted independently on 211 individual notes, approximately 100 percussive sounds, and 100 pitched notes. Actual onsets and detection times were labeled and the latency, i.e., the difference between each couple of timestamps, was recorded. The average onset detection latency measured was 19.00 ms. It is important to note that the latency of detection depends on the parameters of the onset detector, its internal buffer, and the type of input sounds, and is different from the execution time of the detector itself. The execution time of the detector is considerably shorter than the 1.33 ms time-budget of our real-time system (64 samples at 48 kHz), hence the detector was executed at each call of the audio processing routine (see Figure 3.1).

Conversely, all the remaining delays (T_{POD},T_{FE},T_{DNN}, see Figure 3.6) were measured inside the classification software in real-time using a high-precision timer. This

is possible because all the time instants after the actual onset can be probed and measured in the software, whereas the real onset can only be estimated by the detector (hence why T_{OD} was measured separately). These measurements were taken by playing 200 notes (100 percussive and 100 pitched) with the guitar connected to the system, and the results from each timer were saved in a log file. These delays included the post-onset delay, the feature computation time, and the inference time of the classifier (as shown in Fig. 3.6). On average, the post-onset delay was **7.77 ms**, the feature computation time was **0.78 ms**, and the inference time of the classifier was **3.15 ms**. The individual latency distributions are represented in Figure 3.10. The sum of all the delays apart from T_{OD} is **11.70 ms**. The approximate nature of the latency measurement is only present when summing these delays with T_{OD}. As an indicative measure, the sum totals at **30.7 ms**. The following studies adopted a precise latency measuring approach (see Chapter 6).

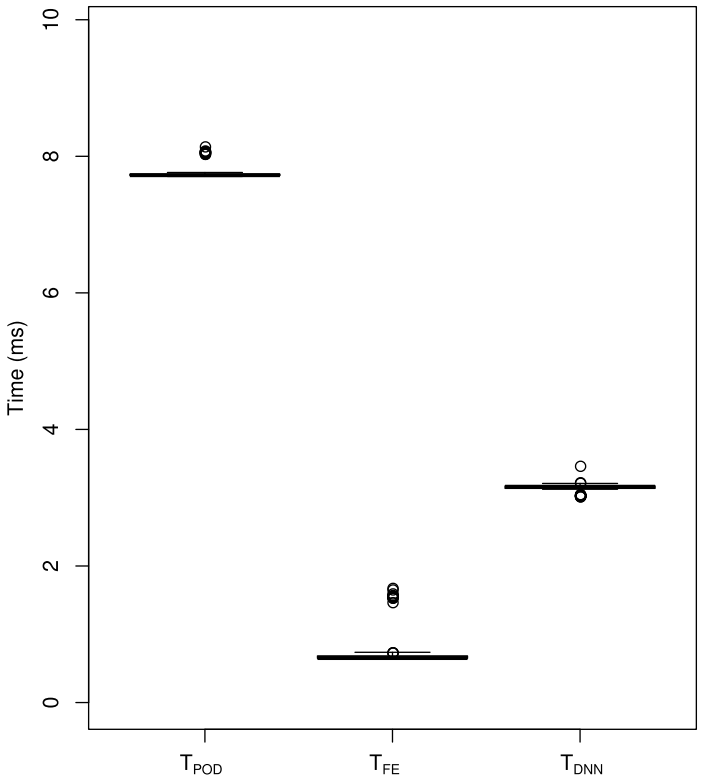


Figure 3.10: *Distributions of the T_{POD}, T_{FE}, and T_{DNN} delays showing small variance around each mean.*

It is with noticing that the neural classifier was deployed to the embedded implementation using TFLite[7] out of simplicity since the models were trained with Keras

[7] https://www.tensorflow.org/lite

and TensorFlow. However, at the time this demonstrative classifier was developed, it was not clear whether different IEs (i.e., libraries for neural network execution) would yield a similar execution time or if the T_{DNN} could be further reduced with a more efficient alternative to TFLite. This was further investigated in a subsequent study, which is presented in Chapter 5.

Nevertheless, the system was connected to the guitar for real-time performance and accompanied by a simple sinewave synthesizer, which was triggered with a different frequency each time that classification was completed. The delay between the guitarist's actions and the synthetic tone was found to be hardly perceivable whenever playing the guitar, indicating that a smart guitar could use a similar system to trigger different synthetic sounds based on the player's technique. Some conclusions on the subsequent direction of the research project are described in the next section.

3.7 Summary

This chapter delved into the challenges of developing embedded real-time Music Information Retrieval systems for music signals. These challenges include the constraints of having access to only past and current input data, the delicate tradeoff between system accuracy and latency, real-time audio deadlines, real-time-safe programming principles, and the limitations of embedded hardware and low-level software. Additionally, we illustrated some of the tradeoffs and solutions to these challenges through the implementation of a real-time embedded expressive guitar technique classifier. The classifier was successfully deployed on a Raspberry Pi 4.

The practical implementation achieved an accuracy of 99.2% in distinguishing pitched and percussive techniques, along with an average accuracy of 99.1% in distinguishing four distinct percussive techniques, alongside a fifth category dedicated to pitched sounds. Conversely, the task of classifying the full set of twelve different techniques proved to be more difficult, and our proposed approach did not obtain satisfactory results. Nevertheless, the classification pipeline with the models for Task A and Task B was successfully integrated into a Raspberry Pi 4 with Elk Audio OS, and the classification outcomes were generated with an average latency of roughly 30.7 ms from the note onset, a delay that was hardly perceptible.

The solutions demonstrated by the experiment include the possibility of breaking down the classification into a stepwise pipeline, each operating at different rates, fine-tuning the system latency, leveraging a real-time embedded operating system to

achieve optimal performance on resource-limited devices, and employing real-time-safe coding practices.

However, the demonstrative classifier is limited in several ways. In particular, we focused on an attack-based real-time classification approach, and we assembled the classifier here as a minimum viable example for the solutions discussed above, which leaves space for better technical choices and tools. Additionally, we adopted a simple approach with a FFNN classifying 1D feature vectors extracted from the feature window. In summary, we found that this research could develop in two different directions in future studies:

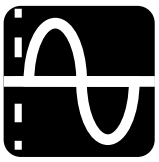
1. The classification framework and pipeline can be redesigned from the ground up to include non-attack-based techniques.

2. The problem can be restricted to the attack-based techniques and the classification pipeline can be refined given the large margins of improvements left by the simple approach adopted here.

We opted to follow the second alternative since we found that most of the non-attack-based techniques can be detected through pitch and envelope tracking (i.e., pitch bending, vibrato). In particular, when thinking about the use of a technique classification system in the context of an advanced guitar synthesizer, it is clear how some of the pitch and envelope-based techniques such as bending, vibrato, and staccato, would not require classification because the pitch tracker and envelope followers integrated into guitar-synthesizer instruments and pedals can already carry these expressive traits over to the synthetic sound.

Furthermore, the refinement of the current pipeline leaves space for interesting problems to solve. Namely, defining a real-time aware optimization for onset detectors, refining feature selection, improving the neural classifier, determining the possibilities and performance of different inference libraries for neural networks on embedded computers, and ultimately aiding the deployment of deep audio models in general on embedded computers. For this reason, the following chapters will discuss subsequent works dedicated to the optimization of onset detectors (Chapter 4), the evaluation and comparison of the performance of embedded IEs for deep learning (Chapter 5), an improved expressive guitar technique recognition system with a more in-depth analyisis of the problem (Chapter 6), and the development of a standard procedure to deploy neural networks for real-time inference on embedded computers with Elk Audio OS (Chapter 7).

Chapter 4



Bio-inspired Optimization of Parametric Onset Detectors

Onset detectors are used to recognize the beginning of musical events, such as notes, in audio signals. Onset detectors that rely on an onset function and peak picking stage expose a series of parameters that can be adjusted to optimize the performance of the algorithm, in terms of how accurate and quick the detection is. However, manually adjusting parameters for these detectors can be a time-consuming task. While there are existing automated approaches, these often only focus on maximizing a single metric that reflects the quality of detection, in terms of accuracy, f1-score, or similar. These approaches can be successfully applied to offline detection systems. However, more complex scenarios require considering multiple competing metrics and finding a suitable tradeoff. In particular, real-time systems that integrate onset detection cannot tolerate optimal detection accuracy at the cost of high detection latency, and conversely, a very low latency parametrization could not justify poor detection accuracy.

To address this issue, we proposed a flexible optimization algorithm that takes into account multiple objective metrics. Our procedure employs a bio-inspired evolutionary computation algorithm to replace costly grid search algorithms or manual parameter tuning. In the proposed approach, multi-objective optimization is achieved by computing the Pareto frontier, composed of the non-dominated solutions in the

space defined by two competing metrics.

The proposed approach was evaluated on all the onset detection methods of the Aubio library, using a dataset of monophonic acoustic guitar recordings. Results show that the proposed solution is effective in reducing the human effort required in the optimization process: it replaced more than two working days (i.e., 16 man-hours) of manual parameter tuning with 13 hours and 34 minutes of automated computation. Moreover, the resulting performance was comparable to that obtained by manual optimization.

This chapter discusses the contribution presented at the 24th International Conference on Digital Audio Effects [135].

4.1 Introduction

Audio Onset Detection (OD) is the process of identifying the beginning of musical notes within audio signals and is commonly utilized for tasks such as automatic music transcription, beat tracking, acoustic event classification or to improve how many standard audio effects adapt to the signal [111]. Additionally, it plays a vital role in many interactive music systems [136], especially novel SMIs [25] where *embedded intelligence* can be used to extract information at the note level. In OD research and practical use, two principal application scenarios can be identified: offline OD and real-time OD. Offline OD is well-suited for music database analysis, where the recognition algorithm processes entire audio recordings, and detection time is not a critical factor. On the other hand, real-time OD operates continuously on an audio stream, and it is often required to perform detection within predetermined time constraints, in order for the detection to trigger subsequent analysis or synthesis algorithms with an imperceptible delay. As previously discussed in Chapter 3, in real-time detection, algorithms can only analyze the portion of the audio signal available at a given time and the preceding historical context (i.e., causal information), and they cannot peek into future audio segments without introducing latency between onsets and their detection.

Historically, a large part of OD research has focused on offline applications [122, 137,138], along with real-time scenarios with generous time constraints [125]. Various probabilistic methods, including deep learning approaches, have been developed for these purposes, typically running on high-powered computers or PCs. However, less attention has been directed towards the development and utilization of OD meth-

ods designed explicitly for real-time scenarios, capable of functioning on resource-constrained embedded devices such as Raspberry Pi or BeagleBone Black single-board computers [139]. Such methods are particularly relevant for applications within the emerging field of SMIs [25], enabling the repurposing of information extracted from the signal with imperceptible latency for the player (see, e.g., [136]).

Deterministic OD algorithms are suitable for these applications, requiring less computational resources than existing neural network-based methods [122,125]. They align well with the limitations of embedded devices. However, most deterministic OD algorithms necessitate precise parameter tuning to achieve high recognition accuracy. Moreover, different choices of parameter values affect the delay with which an onset in the signal can be detected (i.e., recognition latency). The tuning process, which can be conducted through manual adjustments or grid search, entails evaluating the detector's performance across various parameter values using a designated test dataset of audio recordings. Both manual tuning and grid search are time-consuming and often impractical. In particular, grid search is inefficient as it consists of testing all the combinations of parameter values within predefined ranges and at specific intervals. In this way, grid-search cannot automatically steer the search toward areas of the parameter space that are more likely to be a minimum or tune how fine the intervals between parameter values are. Differently, manual optimization can help direct the search toward a global minimum, but it is time-consuming. Additionally, due to the prevalence of offline applications, automated solutions used in research often optimize a single objective function (e.g., detection accuracy), while real-time applications require the optimization of multiple metrics (i.e., detection accuracy and latency).

In [135], we proposed an approach to automated parameter tuning and performance evaluation of time-constrained real-time onset detectors, which can be applied to any parametric OD tool with minor modifications. To optimize detector parameters, our approach employs an Evolutionary Computation (EC) algorithm that models solutions (i.e., any set of parameter values) as individuals within a population and parameters as genetic material. The EC population undergoes iterative updates through operations inspired by natural evolution, including selection of the fittest, reproduction, mutation of genetic material, and generational replacement. Selection steers the evolution towards a predefined goal by enabling the fittest solutions (according to a fitness function) to participate in reproduction and mutation processes, ultimately generating better solutions. Moreover, the proposed method is suitable

for multi-objective optimization.

We applied this proposed procedure to the OD algorithms of the free and open-source Aubio library, aiming to optimize the detector for a real-time timbre recognition method focused on acoustic guitars. The target system's objective was to detect each onset in the audio signal, which in turn triggered a classifier to identify the expressive playing techniques employed by the guitarist. The classification result would then be repurposed in real-time to trigger the synthesis of new sounds. Given that complex sequential sounds tend to be perceived as simultaneous when separated by less than 30 ms [21], this interval was set as the maximum end-to-end latency for the recognition and repurposing system. A maximum target latency of 20 ms was allocated to the timbre recognition algorithm alone (thus excluding sound generation), which can be split between OD, feature extraction, and classification. Measuring the last two tasks on our pipeline resulted in a consistent 6 milliseconds execution time, resulting in a maximum latency of 14 ms that could be used by the OD task. Additionally, we aimed for low variability in detection latency across different sounds, allowing for accurate estimation of actual onset times (see, e.g., [139]). This is particularly relevant as it allows a subsequent feature extraction algorithm in the pipeline to accurately align the extraction window with the onset of the relative note, thus helping pre-onset noise.

The following is the outline of the remainder of this chapter. Section 4.3 describes the proposed method for multi-objective optimization of onset detectors. Section 4.4 reports the evaluation of our method and a discussion of the results. Finally, we summarize the work and draw our conclusions in Section 4.5.

4.2 Background

4.2.1 The Aubio library

The Aubio library [128, 140] is a free and open-source library for audio and music analysis. Since the first version of Aubio, numerous improvements and algorithms were added to its functionalities [1]. At the moment of writing this, the latest version is *0.4.9*. One of the most relevant additions is that of Adaptive Whitening[2]. The Adaptive Whitening technique proposed by [123] is a method for preprocessing spectral frames to improve performance in real-time onset detection. It does so by

[1] https://github.com/aubio/aubio/releases
[2] Added in Aubio version 0.4.5 https://aubio.org/pub/aubio-0.4.5.changelog

normalizing the magnitude of each frequency bin in STFT frames with respect to a recent maximum value for the bin. This reduces the effect of spectral roll-off and can account for variations in the dynamic of the audio signal. However, while Adaptive Whitening improves the performance of various onset functions, it was shown to have a detrimental effect on the Modified Kullback-Leibler (MKL) distance formula [140, p. 42, formula 2.9]. Other whitening techniques were proposed throughout the years, such as approaches based on the Discrete Wavelet Transform [141].

The onset methods that are available in the Aubio library are:

1. **energy:** Energy-based distance, which calculates the local energy of the input spectral frame.
2. **hfc:** High-Frequency content [142], which computes the high-frequency content of the signal (See Equation A.1).
3. **complex:** Complex domain OD function [143], which uses information in both frequency and phase to determine changes that can correspond to musical onsets (See Equation A.2).
4. **phase:** Phase-based OD function [144] (See Equation A.3).
5. **specdiff:** Spectral difference OD function [145].
6. **kl:** Kullback-Leibler OD function [146] (See Equation A.5).
7. **mkl:** Modified Kullback-Leibler OD function [128, Chapter 2] (See Equation A.6).
8. **specflux:** Spectral flux [147] (See Equation A.7).

While current efforts in OD research focus on probabilistic and data-driven solutions such as neural networks [122,125,137,138], deterministic OD methods still guarantee to be able to run within tight time constraints for real-time detection. Moreover, methods are also less computationally expensive than most neural network counterparts (see e.g., [122,125]), which is an essential requirement for resource-constrained embedded devices.

Similar libraries with Onset detection algorithms (e.g., Essentia[3] [148], Librosa [149], Madmom [150]) offer a very similar selection of deterministic OD methods, with

[3] http://essentia.upf.edu

the sole exception of some advanced methods (e.g., SuperFlux [151]). Nevertheless, the proposed method could be applied to a wide range of methods and libraries since it does not make any assumption on how the onset detection algorithm works (e.g., whether it is a straightforward function or a complex neural network).

4.2.2 Evolutionary Computation

EC is a family of optimization algorithms that are inspired by natural evolution. These algorithms model candidate solutions as a population of individuals. Each solution, or individual, is defined by the counterpart to genetic information, which in the case of EC algorithms is a set of values. The information that composes each individual is called *genotype*. The population, initially composed of random solutions, undergoes many stages of evolution, called *generations*, where the fittest individuals are selected for reproduction, their genome is randomly mutated, and they are eventually replaced by their offspring.

The fitness of an individual in EC algorithms is effectively the objective metric to either minimize or maximize, depending on the optimization problem. The strength of EC is in how the fitness function can be any function, from a simple mathematical distance or metric, or a complex metric that relies on software simulations of the evaluated solution. The evolutionary process explores the optimization landscape and searches for the global optimum.

The exploratory nature of EC algorithms lends itself well to creative areas of music research, such as generative audio synthesis and algorithmic composition. A common use of evolutionary algorithms in sound synthesis is to tune the parameters of a synthesizer to match target sounds (see [152, 153]).

Along these lines, Garcia [154] proposed the use of evolutionary optimization to suggest topological arrangements for the functional elements of a sound synthesis algorithm, as well as to optimize the parameters of these elements. Garcia's solution and many other EC approaches employ a fitness function in the form of a mathematical equation (e.g., a distance function between the sound generated by a candidate solution and a target sound). However, EC does not pose particular requirements on the fitness function itself. As an example, Johnson [155] devised an interactive EC system to explore the sound space of a synthesizer. In the system proposed by the author, users were presented with nine candidate sounds at a time and asked to rate each one. The rating was followed by a round of the evolution process, where the rating for each sound was treated as the fitness function result, effectively involving

humans in the fitness function evaluation.

The same flavors of EC approaches (i.e., algorithmic fitness or interactive evaluation) were taken in the field of evolutionary music composition, where some authors defined specific composition goals along with a fitness function (see e.g., [156, 157]), while others embrace the interactive approach with a user-evaluated fitness function (see [158, 159]).

Nevertheless, EC have also been used on occasion for parameter tuning in the field of MIR. Vatolkin et al. [160] applied an EC algorithm to the optimization of feature selection for a musical instrument classifier. The approach proposed by the authors consists of using the performance of the machine learning classifier as a fitness function. Such an application shows how EC algorithms can perform well in the absence of a derivable objective function for the problem. The same advantage is shown by Faragó et al. [161] who successfully applied EC to the design of the sound processor for a hearing aid. Similarly, Pepe et al. [162] applied two different EC algorithms to multichannel audio equalization, where optimization was used to tune filter parameters and match a desired frequency response.

Finally, while research on EC for musical onset optimization is scarce, there have been several successful applications of these algorithms to the problem of Electromyography onset detection for muscle activation analysis. Some examples are the work by Rashid et al. [163] and Magda et al. [164]. Even though electromyographic OD is a different task than musical OD, the two problems show similarities in how they expose simple parameters to tune and how the fitness evaluation can be performed (in terms of detection accuracy).

The EC optimizer proposed in the present study was developed using the Inspyred Python library[4] [165], which provides a wide range of bio-inspired algorithms, including EC and swarm intelligence.

4.3 Proposed method

We proposed a procedure for the optimization of parametric onset detectors that considers both the accuracy of the detection and its latency (i.e., the delay between an onset in the signal and its reporting). The multiobjective optimization method proposed can be summarized with the following six steps:

1. **Dataset preparation:** a dataset of audio recordings from the target applica-

[4] https://pythonhosted.org/inspyred/overview.html

tion must be created and onset times must be annotated. The dataset is then split into validation and test sets;

2. **Fitness function preparation:** devise an evaluation algorithm that, given a set of parameter values for the onset detector, can execute the detector on the validation set and compute the metrics of interest (e.g., accuracy, f1-score, or maximum/average/variability of the detection latency);

3. **Parameter separation:** First, identify potential parameters with problem-dependent values (*fixed parameters*, e.g., minimum inter-onset interval). Subsequently, distinguish the parameters that determine the latency of the computation (we will call these *A-parameters*) from those that do not and can be tuned independently without affecting detection latency (*B-parameters*);

4. **Evolutionary Optimization for single-objective:** Run the proposed EC algorithm on each combination of A-parameters, with the B-parameters as the genotype of the individuals in its population. The fitness of each solution is evaluated on the validation set by the algorithm devised for Step 2, and the evolution is performed for a set number of generations;

5. **Pareto front computation (multi-objective step):** For each final solution (i.e., the best solutions for each combination of A-parameter values), compute the set of non-dominated solutions that compose the tradeoff curve (i.e., the Pareto front) between the two metrics of interest (e.g., accuracy and latency);

6. **Solution Selection:** Since all the solutions in the Pareto front are viable tradeoffs between the two objectives, a suitable solution must be chosen from this set. The choice can depend on the shape of the front and the application requirements. The selected solution, which is a set of parameter values, can then be evaluated on the test set to obtain a final estimate of its performance.

The following sections describe each step of the proposed method in detail. The details on the evaluation of the method on our problem are discussed in Section 4.4 instead.

4.3.1 Dataset Preparation

The first step involves preparing a dataset of audio recordings and providing quality annotations of each onset. The datasets used for optimization can be a subset of a

larger dataset, reducing the time required for labeling and making it possible to check the quality of the labels for each onset sound. To extract a subset that maintains the characteristics of interest of the original dataset (e.g., distribution of different types of sounds), stratified random sampling can be used. Then, all the onsets in the sampled set have to be labeled, meaning that one or multiple files with the timestamp of each onset have to be created. The time resolution of the labels can vary depending on the application of interest and the labeling process can be carried out with the help of annotation software. Audacity[5] and Sonic Visualizer[6] [166] are two pieces of free and open-source software that enable annotation of audio tracks. In these programs, the time accuracy of the labels can be adjusted by tuning the zoom level with which the audio signal is visualized. When using both the signal and spectrogram for reference, the parameters of the spectrogram should be tuned to reduce the size of the analysis frame, thus increasing time accuracy at the cost of a reduced resolution of the frequency axis. Figure 4.1 shows the resolution settings for the spectrogram. Additionally, the range of the vertical axis in the waveform visualization of the signal should be set low enough to include quiet sounds, but to exclude noise where possible (see Figure 4.2). Figure 4.3 shows an annotated onset in Audacity, with the side-by-side view of the waveform and spectrogram. Furthermore, the same recordings can be labeled by more than one annotator to reduce potential errors, as suggested by Brossier [128, p. 52].

The labeled dataset can then be divided into a validation set and a smaller test set: the first will be used to drive the optimization process, while the second will be used to test the generalization performance of the final solution. The dataset used for this study is described in Section 4.4.1.

4.3.2 Fitness Function

The second step involves the development of a fitness function, which is the objective measure that will drive the optimization towards the best solution. The input of the fitness function should be one solution (e.g., in this case, a value assignation for each parameter of the onset detector) and the output should be multiple measures of the success of the onset detector on the input data. It is worth noting that the fitness function does not need to be differentiable and can even be a black-box program.

To measure the degree of success of detection, we can define a tolerance win-

[5] https://www.audacityteam.org/
[6] https://www.sonicvisualiser.org/

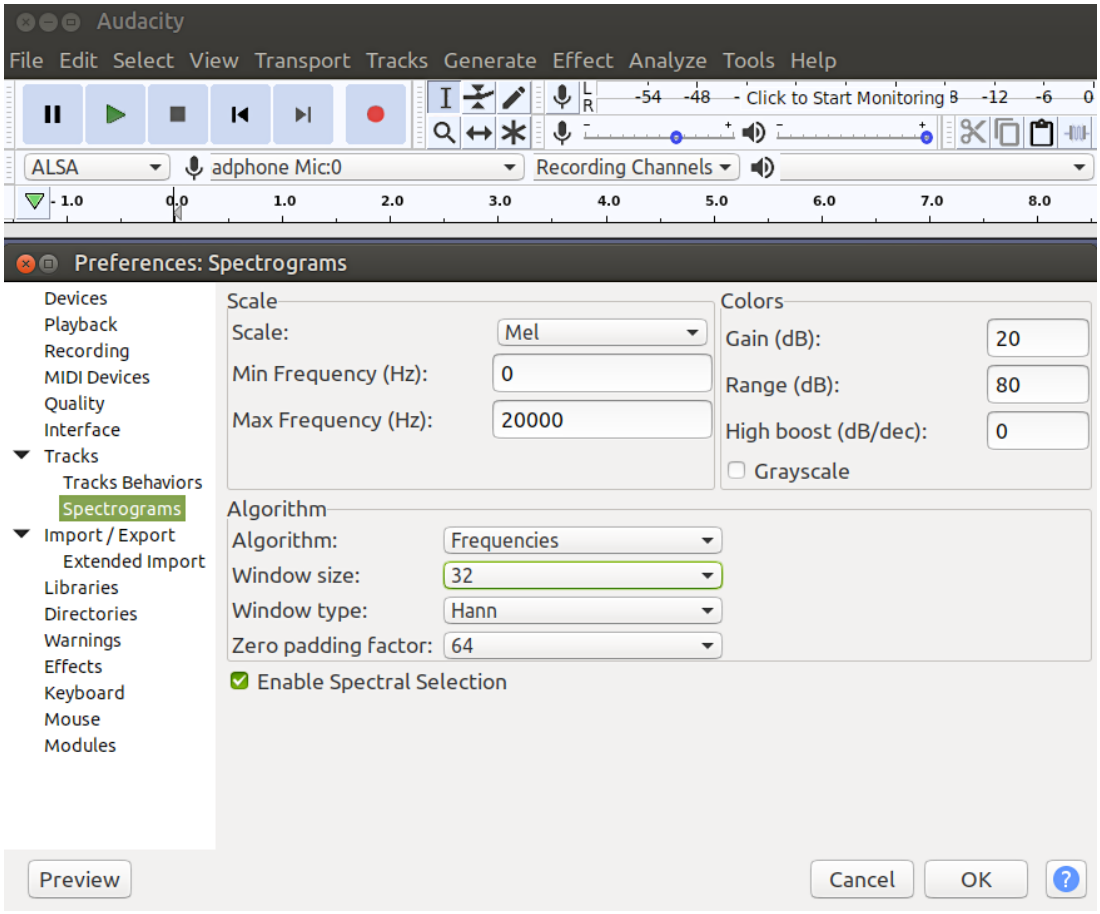


Figure 4.1: *Resolution settings for the spectrogram view for onset annotation on Audacity.*

dow around each hand-labeled onset and consider *correct detection* (or True Positive (TP)) any detection that falls within said window [128, Chapter 2.5.2]. In addition to correct detections, errors can be further divided into False Positives (FPs) and False Negatives (FNs): the former are onsets that are detected outside a tolerance window, while the latter describes true onsets that have not been detected. Duplicate detections (more than one onset detected in the same window) can either be counted separately or considered as FPs (except for the first correct detection). Additionally, the f1-score is a metric that takes into account both correct detections and errors.

$$F_1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{4.1}$$

The f1-score is defined as the harmonic mean of Precision and Recall (Eq. 4.1)., where Precision is the ratio of correct detections to the number of onsets reported by the detector (Eq. 4.2), while Recall is the ratio of correct detections to the number of

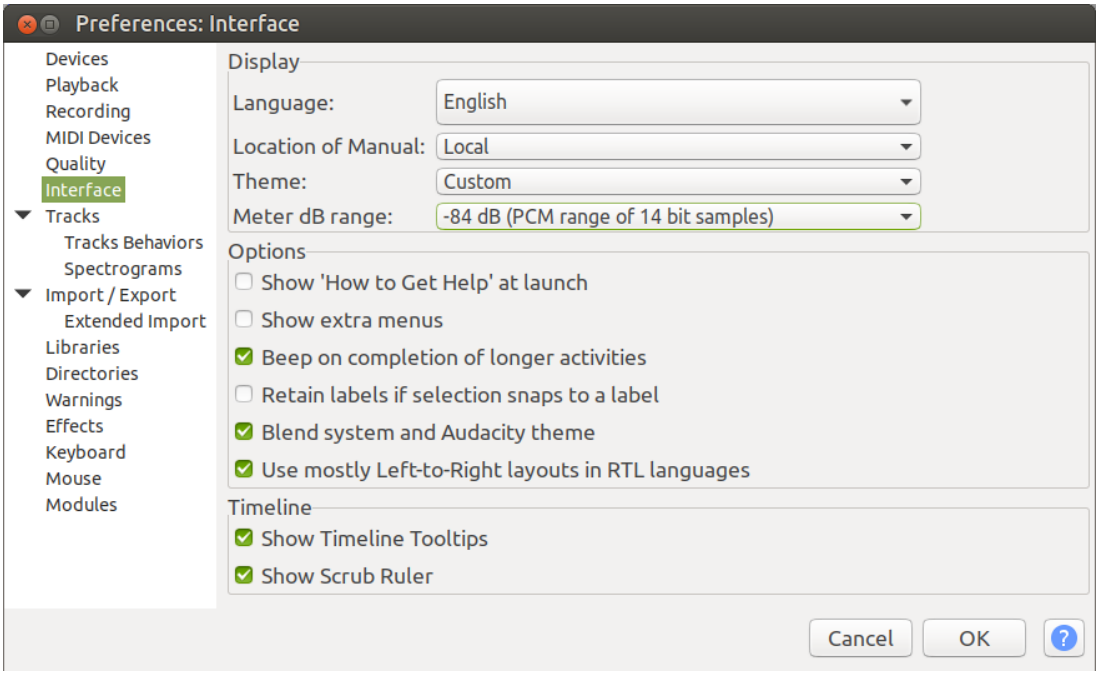


Figure 4.2: *Range of the signal visualization on Audacity.*

labeled onsets (Eq. 4.3).

$$Precision = \frac{TP}{TP + FP} \tag{4.2}$$

$$Recall = \frac{TP}{TP + FN} \tag{4.3}$$

To measure the latency of the detector instead, it is sufficient to measure all the time intervals between correct detections and the relative ground-truth label. Subsequently, a single statistic of interest can be taken to represent the distribution of the detection latency, such as the maximum value, mean, or variance. However, while in practice we computed the distribution of detection latency at this stage, the proposed procedure consists of the execution of an EC algorithm with a single objective (maximization of the detection f1-score) for parameters that do not impact latency, and a subsequent step of multi-objective optimization (see Section 4.3.5).

4.3.3 Parameter Separation

The third step involves dividing the parameters of the detector into three distinct categories:

1. Fixed-parameters;

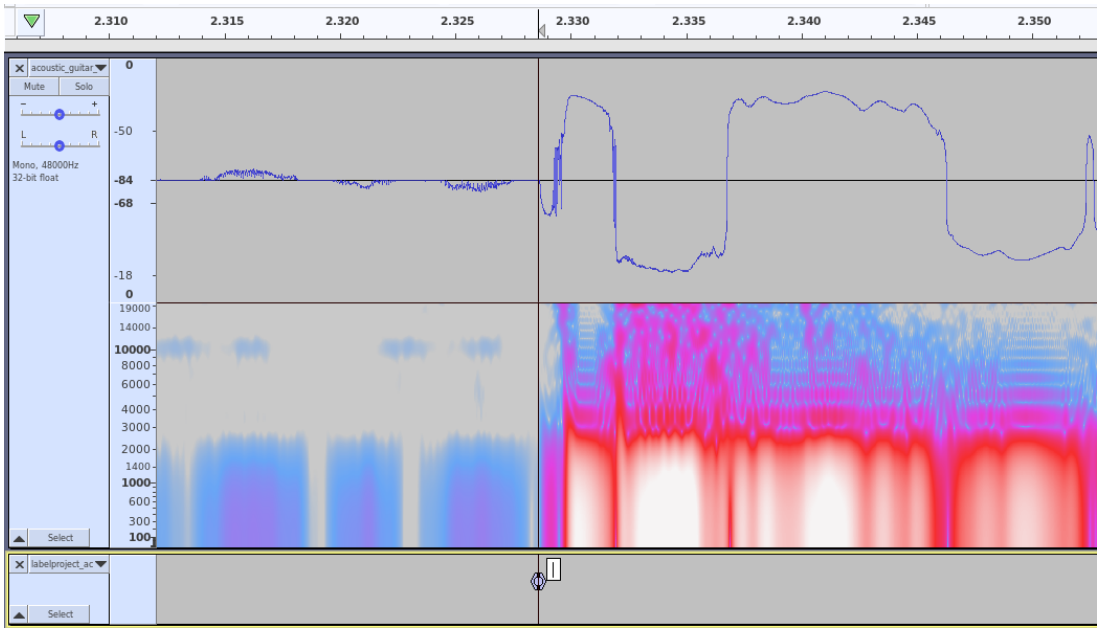


Figure 4.3: *Annotated onset in Audacity with the time zoom set so that the time ruler ticks are one millisecond apart.*

2. Free *A-parameters*;

3. Free *B-parameters*.

Fixed parameters are parameters whose values depend on the problem and cannot be subject to tuning or optimization. One example can be the minimum inter-onset interval present in many onset detectors [128]. The minimum inter-onset interval is the duration of the time interval for which the detector is disabled after each detection. Such a mechanism is used to prevent the high energy of the signal at the onset moment from causing repeated detections (i.e., false positives). The value of the minimum inter-onset interval can be dependent on the problem and can be defined, for example, by determining the fastest reasonable playing pace for the target instrument or defining the fastest playing pace for which we desire the detector to consider multiple onsets as distinct entities.

Free parameters are divided into those that affect both latency and detection accuracy (i.e., *A-parameters*) and those that only affect accuracy (i.e., *B-parameters*). This is done so that multiple combinations of A-parameters can be selected, each constituting a single-objective optimization problem on the remaining B-parameters. Each subproblem can be solved independently (and in parallel), and then a multi-objective optimization step can be performed, taking into account both detection

accuracy and latency.

4.3.4 Evolutionary optimization for single-objective

This step of the proposed procedure involves optimizing the B-parameters for every combination of the A-parameters. For this task, we employ an EC algorithm since this class of optimization methods requires no assumption on the problem or the input parameters (black-box optimization). EC algorithms model the candidate solutions of an optimization problem as individuals of a population and evolve such population through multiple generation rounds using operations that are inspired by natural evolution. In each generation round, the fittest individuals of the population according to an objective *fitness* metric are selected for breeding, which leads to the creation of a number of new individuals (i.e., offspring) to promote the recombination of good solutions. Then, some individuals can be subject to mutation, which alters the nature of the solutions to promote exploration of the search space. Finally, the population is replaced, either entirely or partially by the offspring. After replacement, the evolution procedure can start with a new generation.

The individuals of the population of an EC algorithm are defined by their genotype and phenotype. The genotype of an individual is a parameter configuration for the algorithm to optimize, i.e., a set of values, containing a value for each parameter of the problem. The phenotype is instead the manifestation of the genotype, which in this case is the algorithm with its candidate parameter configuration and its resulting performance. The phenotype is what can then be evaluated, resulting in a measure of the fitness of the solution (i.e., how close the solution is to a global optimum). The evolutionary process can lead to an optimal solution through exploration of the search space and refinement of good solutions, which get closer to global optima while weak solutions do not survive.

The steps of a generic EC algorithm are shown in the form of pseudocode in Alg. 1 (inspired by [167]).

In the proposed method, the genotype is a vector of B-parameter values. The phenotype is the performance of the detector of choice with the parameter values of each candidate solution. The fitness function is a measure of said performance and consists of the f1-score of the detection with respect to ground truth onset labels. Selection, Crossover, Mutation, and Replacement strategies are further discussed in Section 4.4.

Algorithm 1: Evolutionary Algorithm

```
Generate initial random population
while generation <= max_generation do
    generation = generation+1
    compute fitness of each individual
    select individuals depending on their fitness
    perform crossover with probability crossover_rate
    perform mutation with probability mutation_rate
    use replacement strategy to create the new population
end
```

4.3.5 Pareto Front Computation

The evolutionary optimization step is performed for each combination of A-parameters, and it involves tuning B-parameters. For each combination of A-parameters, the EC returns the single solution with the highest detection accuracy obtained on the validation dataset.

The following step is to take the set of solutions obtained through the EC algorithm and compute the set of optimal solutions with respect to both the success metric of choice (e.g., f1-score) and a measure of the detection latency (e.g., maximum value or variance). The optimal solutions are obtained by computing the Pareto front, which is the set that contains all the solutions for which none of the objective functions can be improved further, without reducing some of the other objective values [168]. Therefore, all the solutions in the front, which are called non-dominated or *Pareto optimal*, are all acceptable, equally valid from an objective standpoint, and offer different compromises between detection success and latency.

4.3.6 Solution Selection

The last step consists of selecting the single most desirable tradeoff between detection accuracy and latency among the optimal solutions in the Pareto front. Since all solutions in the front are acceptable and equally valid with respect to the multiple objectives, selecting a solution can depend on the shape of the front and the problem domain. Figure 4.4 is an example of a Pareto front between the f1-score and maximum latency of a set of onset detectors and parameter configurations. Since the f1-score must be maximized and the maximum latency minimized, the ideal solution would be at the top left corner of the plot (highest f1-score, lowest latency). By contrast, any solution near the bottom-right corner has high latency and a low f1-score.

Furthermore, any solution that is not part of the front performs worse than a solution in the front. In this case, solutions 1 and 2 may have too low of an f1-score for the problem. Additionally, the shape of this particular front highlights how there is little change in f1-score between solutions 6 and 7, but the slight increase in performance of solution 7 comes at the cost of a sizeable increase in latency. Without any additional assumption on the problem, solutions 3, 4, 5, and 6 may seem the best candidates as they describe a convex knee in the front shape towards the ideal optimum (i.e., the top left corner of the graph). However, depending on the application, extremely low latency may be much more relevant than detection accuracy, meaning that the latency of solution 1 could outweigh the decrease in accuracy from 2 and all the other solutions in the front. Alternatively, the latency of solution 7 could still be within reason, and it could be outweighed by its accuracy. Independently from the specific final choice, the nature of the Pareto optimal front ensures that no other solution was found to have both better accuracy and latency. Figure 4.5 shows a different front with more concave areas. In this case, solution 2 has a marginally better f1-score than 1 with very little increase in latency, and 3 offers close to no advantage in f1-score while showing quite a larger latency result. A similar reasoning holds for solution 4 when compared to 3 and 5.

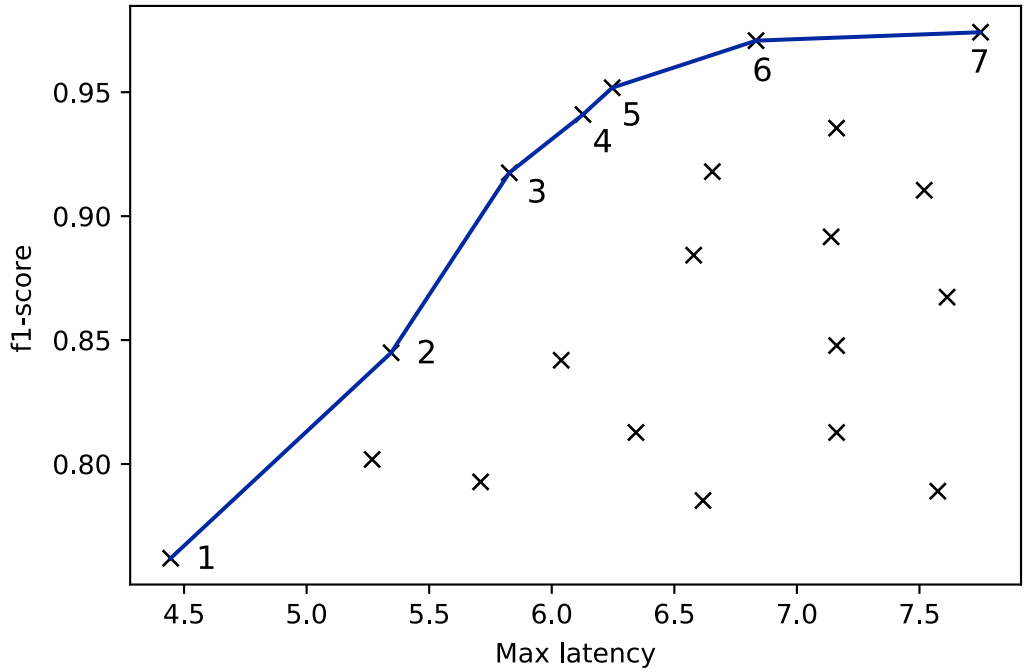


Figure 4.4: *Pareto front between the f1-score (to maximize) and maximum latency (to minimize) of a set of onset detectors and parameter configurations. Each solution in the front is non-dominated, which means that any other set of parameters results in a point in the lower-right area (i.e., dominated area) defined by the curve.*

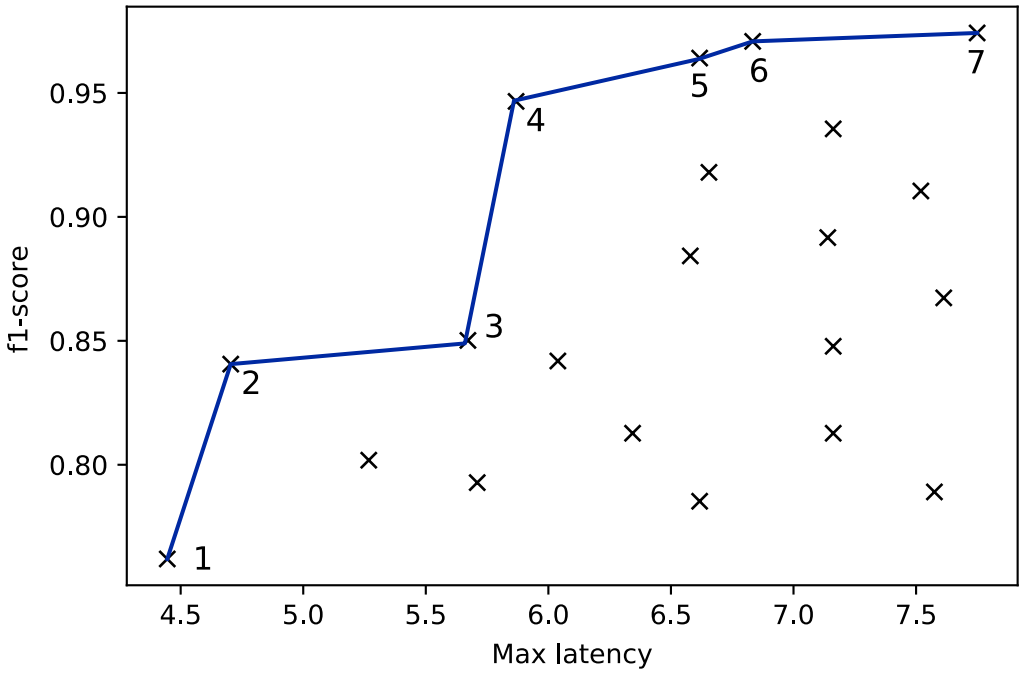


Figure 4.5: *Additional example of a Pareto front between f1-score and maximum latency (see Fig. 4.4). This case shows a front with more concave segments, highlighting how solutions such as 3, 5, 6, and 7 may be easily discarded since there are neighboring solutions that greatly benefit one of the objective metrics without affecting sensibly the remaining (i.e., solutions 2 and 4).*

4.4 Evaluation and discussion

The proposed optimization procedure was applied to all the OD methods of the Aubio library. The solutions found by the EC algorithm were compared to those obtained by manually tuning the parameters. The source code is available online[7].

4.4.1 Input data

The input data used is a representative sample of our dataset composed of individual acoustic guitar sounds [28].

For the optimization of the onset detector, we extracted two samples as the validation and test sets. The validation set consisted of a sample of 1328 individual sounds that was extracted from the main dataset with stratified random sampling to preserve the original distribution of samples for each playing technique, guitar, guitarist, and dynamic level. The test set was composed of 336 sounds extracted in the same way as the validation set. Finally, the onset times were labeled by one annotator with the open-source Audacity audio software, providing ground truth labels for the evaluation of the detector. The data used for this study is publicly available in the project repository.

4.4.2 Evaluation algorithm

Step 2 consists of developing a fitness function that can measure the success of the onset detector on the input data. In addition to the fitness metric used in the single-objective optimization step (i.e., f1-score), we opted to also compute the latency at this stage.

Success was quantified using the average f1-score metric (Eq. 4.1) for each playing technique in the dataset, considering as correct the first detection within a tolerance window of 20 ms that follows each hand-labeled onset. The exact interval between each onset label and its relative time of detection was measured. The latency distribution for each parameter configuration is used at the later stage of multi-objective optimization.

The evaluation program takes as input a set of parameter values for the detector and runs the aubioonset[8] executable provided with the library. The aubioonset

[7] https://github.com/domenicostefani/BioInspiredOnsetDetection
[8] https://aubio.org/manpages/latest/aubioonset.1.html

executable normally produces an output file with a list of onset times predicted by subtracting a set time interval from the time of detection (i.e., predicted detection latency), so it was modified to produce the list of the actual times at which onsets are detected. The fitness evaluation program was devised to use different folders in the disk for each instance so that it is possible to execute more instances in parallel.

4.4.3 Onset detector parameters

Step 3 involves distinguishing fixed parameters from free parameters, where the latter are further divided into parameters that affect the detection latency (**A-parameters**) and the remaining which only affect accuracy (**B-parameters**). The values of fixed parameters are imposed by the requirements of the problem at hand. In this case, the fixed parameters were the **hop size** and the **minimum inter-onset interval**. The hop size is the number of samples between two consecutive analyses, and it determines the rate at which detection is performed: a value of 64 samples was deemed adequate for our problem (64 samples at 48000 Hz equals 1.33 ms) as it results in a rapid update of the onset detector. The minimum inter-onset interval is the shortest time interval between the onsets that the analysis can report, and it is used to avoid double detections. A value of 20 ms for the minimum inter-onset interval was considered appropriate since it matches the real-time latency requirement described in Section 4.1, and was found not to affect the detection of successive, reasonably quick, guitar onsets. The size of the buffer and the algorithm used for detection directly influence latency by determining the number of computations performed at each analysis, while the silence and onset thresholds only modify the signal level considered as the noise floor and how strong a peak must be in order to be considered an onset.

Contrary to fixed parameters, free parameters must be optimized and were further divided into A and B-parameters. In the case of Aubio, A-parameters are the **buffer size** and the **OD function** used. B-parameters are the **silence threshold** and the **onset threshold**. Table 4.1 shows all the parameters of the detector.

4.4.4 Single-objective evolutionary optimization step

For step four, we selected a range of A-parameters, consisting of buffer size values between 64 and 2048 samples (i.e., 64, 128, 256, 512, 1024, and 2048) and all 8 available onset methods, plus MKL with adaptive whitening disabled at initialization. Each combination of A-parameter values requires the optimization of the remaining

Table 4.1: *Summary of Aubioonset parameters with the range considered for optimization, their category, and whether they affect directly the detection latency or not.*

Aubio Parameter	**Optimization Range**	**Category**	**Determines latency**
Hop size	-	Fixed	Yes*
Min. i.o.i.**	-	Fixed	No
Buffer size	[64,2048]	A-par.	Yes
Method	Aubio methods (See Table A.1)	A-par.	Yes
Silence threshold	[-60 dB, -30 dB]	B-par.	No
Onset threshold	[0.1, 3.6]	B-par.	No

* Hop size affects the detection latency but it is not an A-parameter since it is constrained by the problem requirements (fixed).

** Minimum inter-onset interval.

B-parameters to obtain the most accurate configuration (in terms of f1-score). Since B-parameters do now affect latency, a single-objective optimization step (accounting only for detection f1-score) can be performed for each combination of the remaining free parameters.

Previous to the use of an automated optimization strategy, we manually selected different B-parameter values and measured performance using the evaluator script to further refine the parameter values and repeat the measurements. The manual procedure followed can be defined as a coarse grid search, followed by a refinement of the parameters near performance peaks in the search space. This technique was used because the brute-force approach of a fully automated grid search was shown to be either too coarse or too time-consuming. Manual optimization proved to be time-consuming and required more than 2 workdays. Given the large amount of human effort required and the limited scalability of the manual procedure, an automated EC algorithm was used.

We used the *Inspyred* Python library was used to help the development of the optimizer. Only a few modifications are necessary to adapt the base EC optimizer structure of the Inspyred library to the specifics of our problem. In detail: the individuals of the population were defined as vectors with two elements (i.e., silence threshold and onset threshold). Moreover, the evaluator of the EC algorithm was

set to call our custom fitness function (see Section 4.3.2). As a result, the fitness metric used was the f1-score yielded by the individual's parameter configuration. The Inspyred library contains several standard evolutionary algorithms such as Genetic Algorithm, Evolution Strategy, and Simulated Annealing as well as a custom EC framework that allows different evolutionary operators to be composed. The custom framework was used.

The different settings of the optimizer were refined through trial and error, resulting in the following evolutionary operators and values:

- **Population:** 30 individuals, where each individual is a vector containing a value for each B parameter. A greater number of individuals could increase the possibilities for improving the solutions even with fewer generations, however, it would require more execution time.

- **Evolution termination:** automatic termination after 30 generations. The termination strategy can be defined by trying different values and evaluating the fitness plots: if fitness keeps increasing over time the termination deadline can be moved further away, while a stagnating behavior shows that termination can happen earlier.

- **Selection strategy:** tournament selection with size 4, which holds a "tournament" by randomly sampling n individuals (4 in this case) and choosing the one with the best fitness (see [169]).

- **Crossover:** Arithmetic and Laplace recombination operators (rate = 0.7), which are common choices for combining good solutions when using real-valued genotypes (see [170, 171]).

- **Mutation:** Gaussian mutation operator, which adds a random value from a Gaussian distribution to each element of an individual's genotype to produce a new offspring. The probability of performing the mutation (i.e., mutation rate) was set to 0.7, while the mean and standard deviation of the Gaussian distribution were 0 and 3.0 respectively. Mutation helps candidate solutions to move away from potential local optima in the fitness landscape.

- **Replacement strategy:** Generational Replacement with elitism, meaning that the entire existing population is replaced by the offspring at the end of each generation, except for the best n solutions (in this case with 1 elite),

which survive if they are better than the worst n offspring. Elitism helps to preserve the best individuals.

A summary is presented in Table 4.2, while Fig. 4.6 shows the algorithm procedure. The parameters and operators for the evolutionary algorithm were obtained by testing different combinations on a reduced number of generations for a quicker process.

Table 4.2: *Settings used for the EC algorithm.*

Setting	**Value**
Individual genotype	`<Silence threshold, Onset threshold>`
Population Size	30
Termination	30 generations
Selection strategy	Tournament selection, size: 4 (see Section A.2.1)
Crossover	Arithmetic crossover + Laplace crossover (see Sections A.2.2 and A.2.3)
Mutation	Gaussian mutation operator, mutation-rate: 0.7, mean: 0.0, and standard deviation: 3.0
Replacement	Generational Replacement with elitism, num elites: 1 (See Section A.2.4)

Manual and Automatic optimization comparison

The automated optimization procedure was executed with 11 parallel instances on a laptop with an Intel ® Core™ i7-10750H CPU. Since the execution time is directly proportional to the buffer size of the detector, optimization instances were scheduled so to distribute the load between parallel processes. The optimization instances for the three largest values of the buffer size were scheduled on the first nine parallel processes, while the remaining instances were executed on the last two processes (see Fig. 4.7). The last process to terminate took 13 hours and 34 minutes from the beginning of the optimization to termination.

The highest f1-score values obtained for each instance were comparable to the results obtained by manual optimization. The f1-score was computed on the test set to evaluate how each parameter configuration applies to new data. The mean of the difference between the f1-score obtained in the test dataset with the EC and the

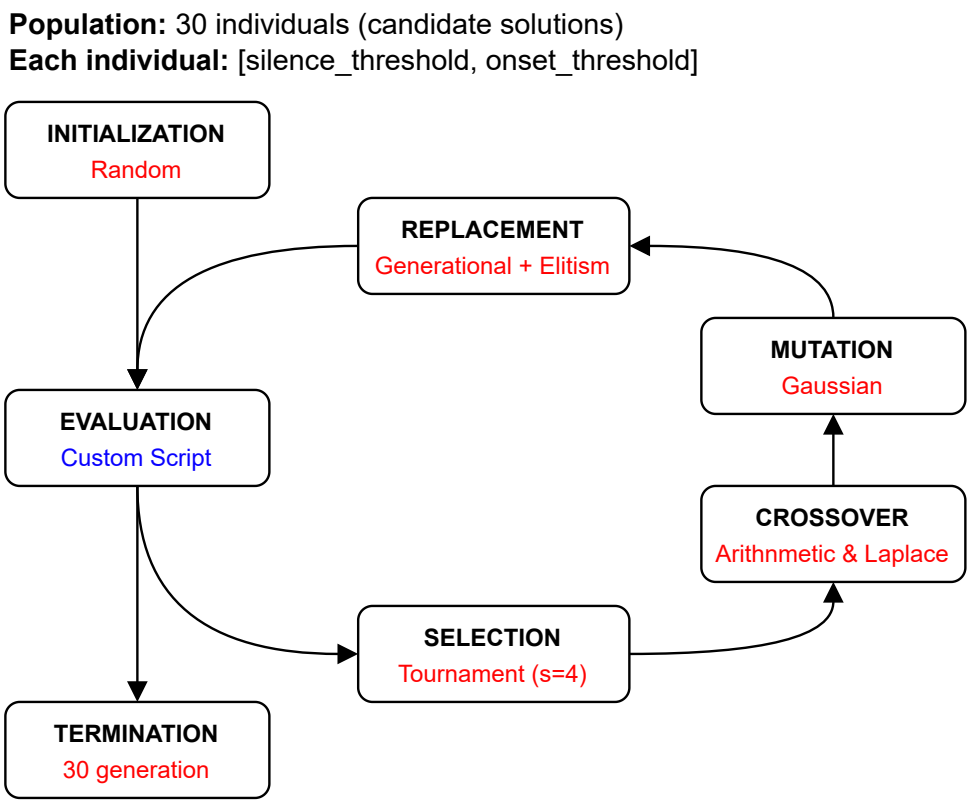


Figure 4.6: *Flow of operations of the proposed evolutionary algorithm.*

manual optimization process was 1.4×10^{-3} points and its standard deviation was 1.2×10^{-2}.

The performance gain of the results of the EC algorithm over manual optimization is shown in Fig. 4.8 as the difference of the f1-score values in each instance (Eq. 4.4, 4.5, and 4.6). Table 4.3 then shows the best f1-score results obtained for each optimization instance.

$$Gain_{b,m} = ECF1score_{b,m} - ManualF1score_{b,m} \tag{4.4}$$

$$b \in [64, 128, 256, 512, 1024, 2048] \tag{4.5}$$

$$m \in aubio\_onset\_methods \tag{4.6}$$

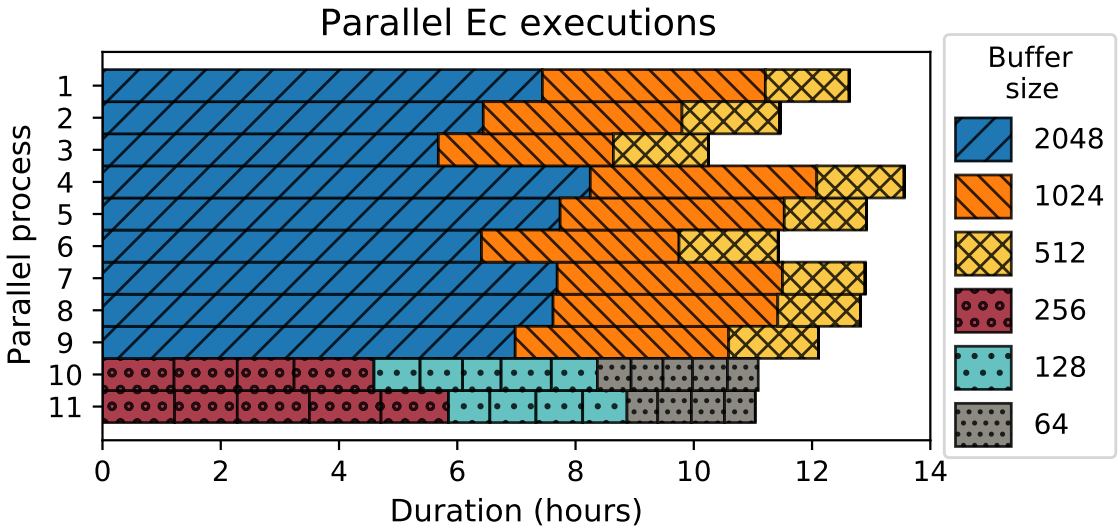


Figure 4.7: *Parallel EC process schedule. Each stack of bars represents the execution time of one of the parallel processes, while each individual bar indicates a single EC optimization instance.*

4.4.5 Multi-Objective Optimization

The result of step four is a set of solutions that are optimized for each combination of A-parameters, while step the subsequent is to perform multi-objective optimization to obtain the best overall performance in terms of both detection accuracy and latency.

The two main requirements for our target application are the following:

1. The maximum detection latency must be lower than 14 ms to comply with the end-to-end 20 ms deadline defined in Section 4.1 for the timbre recognition system. The system is composed of the OD algorithm and a classification

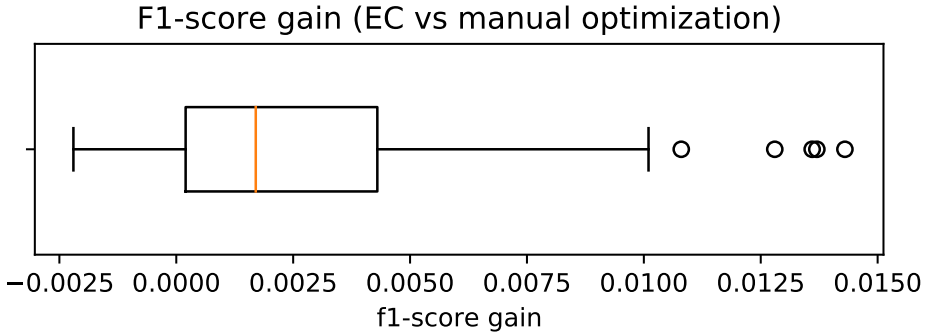


Figure 4.8: *F1-score Gain of the proposed EC algorithm results with respect to manual tuning*

Table 4.3: *f1-score values (percentage) of the best solutions obtained from the EC algorithm for each combination of OD method and buffer size used. The highlighted values are the non-dominated solutions of the Pareto front computed in Section 4.4.5 (Table 4.4).*

		Buffer size					
		64	128	256	512	1024	2048
Method	hfc	93.37	92.31	91.26	89.68	90.15	88.56
	energy	94.10	94.20	94.80	94.82	95.58	91.63
	complex	83.71	85.09	86.80	87.55	86.66	80.45
	phase	76.20	82.34	87.40	82.06	74.26	71.62
	specdiff	86.67	93.67	95.35	95.29	95.49	93.39
	kl	85.17	86.16	87.95	87.52	89.19	82.41
	mkl	84.84	85.90	87.22	86.96	88.18	88.14
	specflux	**84.49**	91.75	**92.43**	91.21	87.97	86.82
	mkl(noaw)*	**95.18**	97.08	**97.42**	97.38	97.30	96.30

* The MKL method with adaptive whitening disabled on initialization.

algorithm that consistently takes 6 ms to execute, hence the remaining time interval of 14 ms that is assigned to the detection;

2. The variability of the latency distribution must be as low as possible. Thanks to this, it will be possible to estimate the time of the actual onset with high confidence by subtracting a fixed temporal interval from the detection time (e.g., average or maximum detection latency).

In order to comply with these requirements, both the maximum latency and its variability were calculated. Maximum latency was expressed in the form of the upper Tukey fence. Tukey fences [172] are values that define the range of a data distribution while ignoring data points that differ significantly from other observations (i.e., outliers). Tukey fences are commonly used to determine the position of box plot whiskers and are computed using quartile values (i.e., Q_1, Q_2, Q_3) and a constant k, with a value of 1.5 for outliers [172] (Eq. 4.7).

$$TF = \big[Q_1 - k(Q_3 - Q_1), Q_3 + k(Q_3 - Q_1)\big],\ k = 1.5 \tag{4.7}$$

On average, 95.7% of the detected onsets were within the upper and lower fences computed.

Onset detection latency was instead described with the Interquartile Range (IQR)

of the latency distribution. The IQR is a measure of the dispersion of a distribution and it is defined as the difference between the third and first quartile values (Eq. 4.8).

$$IQR = Q_3 - Q_1 \tag{4.8}$$

The IQR was used because, unlike variance, it is expressed in the same unit of measure of the distribution data (milliseconds). IQR can also be multiplied by four in order to find the interval between the upper and lower Tukey fences. Both Tuckey fences and IQR are used to produce box plots, which are a common and clear method to visualize many properties of a distribution.

As described in Section 4.3.5, the multi-objective optimization step consists of computing the Pareto front of the solutions obtained in the previous step. The Pareto front was computed using f1-score and latency IQR as objectives. All the solutions whose maximum latency was greater than the threshold defined (14 ms) were discarded.

The solutions are presented in Fig. 4.9 along with the Pareto front (dotted line) and the discarded results. Discarded results are indicated by the larger red circle markers in the upper right area. A more detailed view of the front is shown in Fig. 4.10 and the solutions are listed in Table 4.4.

Table 4.4: *Solutions in the pareto front of figures 4.9 and 4.10, with f1-score as the first objective and IQR of latency as the second.*

#	**Method**	**Buffer** Size	**f1-score** (%)	**Low Tukey fence (ms)**	**Latency mean (ms)**	**High Tukey fence (ms)**	**IQR (ms)**
a	specflux	64	84.49	2.2	3.8	5.3	0.80
b	specflux	256	92.43	3.1	4.8	6.3	0.81
c	mkl(noaw)	64	95.18	2.7	4.5	6.2	0.88
d	mkl(noaw)	256	97.42	4.2	6.0	7.7	0.89

Differently, if low latency variability was not a requirement, the second objective could be the minimization of the *maximum* latency. We show an example of the Pareto front computed for f1-score maximization and maximum latency minimization in Fig. 4.11.

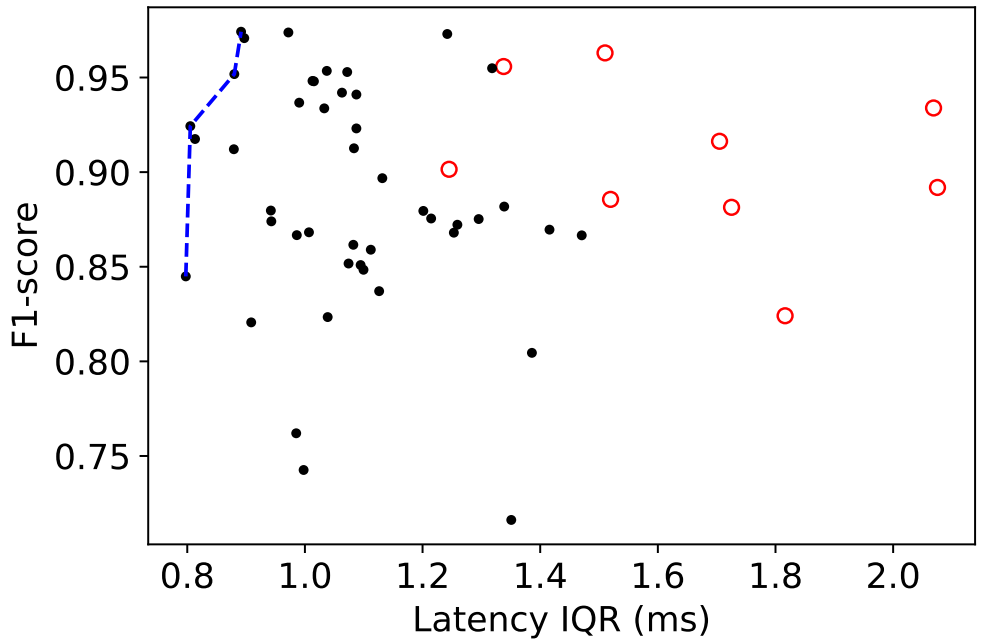


Figure 4.9: *Solutions of the EC optimization step represented in the space described by the f1-score and the latency IQR objectives. The blue dotted line represents the Pareto front that minimizes IQR and maximizes the f1-score, while the solutions represented with the red hollow circles are the ones discarded because their maximum latency is over the maximum value allowed (14ms).*

4.4.6 Choosing a solution

By definition, the Pareto front contains all the best solutions that offer different compromises on the two objectives. For this reason, any tradeoff belonging to the Pareto front can be chosen depending on the problem requirements and the shape of the curve.

For this problem, solution $\#d$ (Fig. 4.10, Tables 4.3 and 4.4) is chosen since it provides the greatest f1-score value without having excessive latency IQR. Solution $\#d$ was obtained with the MKL method, no adaptive whitening, a buffer size of 256 samples, a silence threshold of -51.7 dB, and an onset threshold of 1.18. On the test dataset, solution $\#d$ obtained an f1-score of 97.33%, an IQR of 0.58 ms, an average latency of 6.0 ms, and lower and upper Tukey fences of respectively 4.8 and 7.2 ms. The performance of solution $\#d$ the test set shows that it generalizes to new data.

However, it is worth noticing that if the Pareto front had a different shape (e.g., more convex, more concave, or with a different slope) the chosen solution could be different. In particular, the solution that has the best result according to one of the

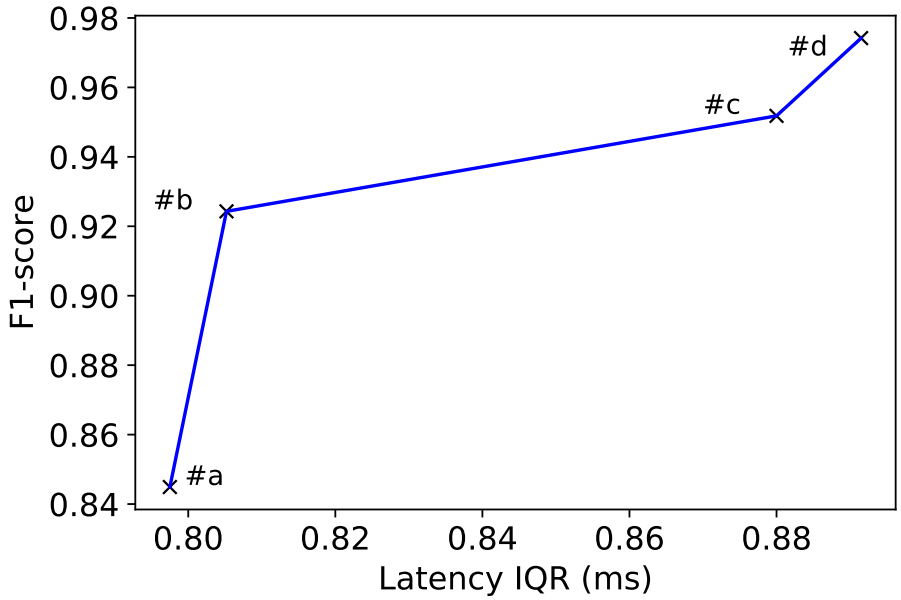


Figure 4.10: *Closer view of the Pareto front shown in Figure 4.9.*

objectives may not be the most desirable tradeoff. Section 4.3.6 presented possible alternatives in the case of different Pareto front shapes.

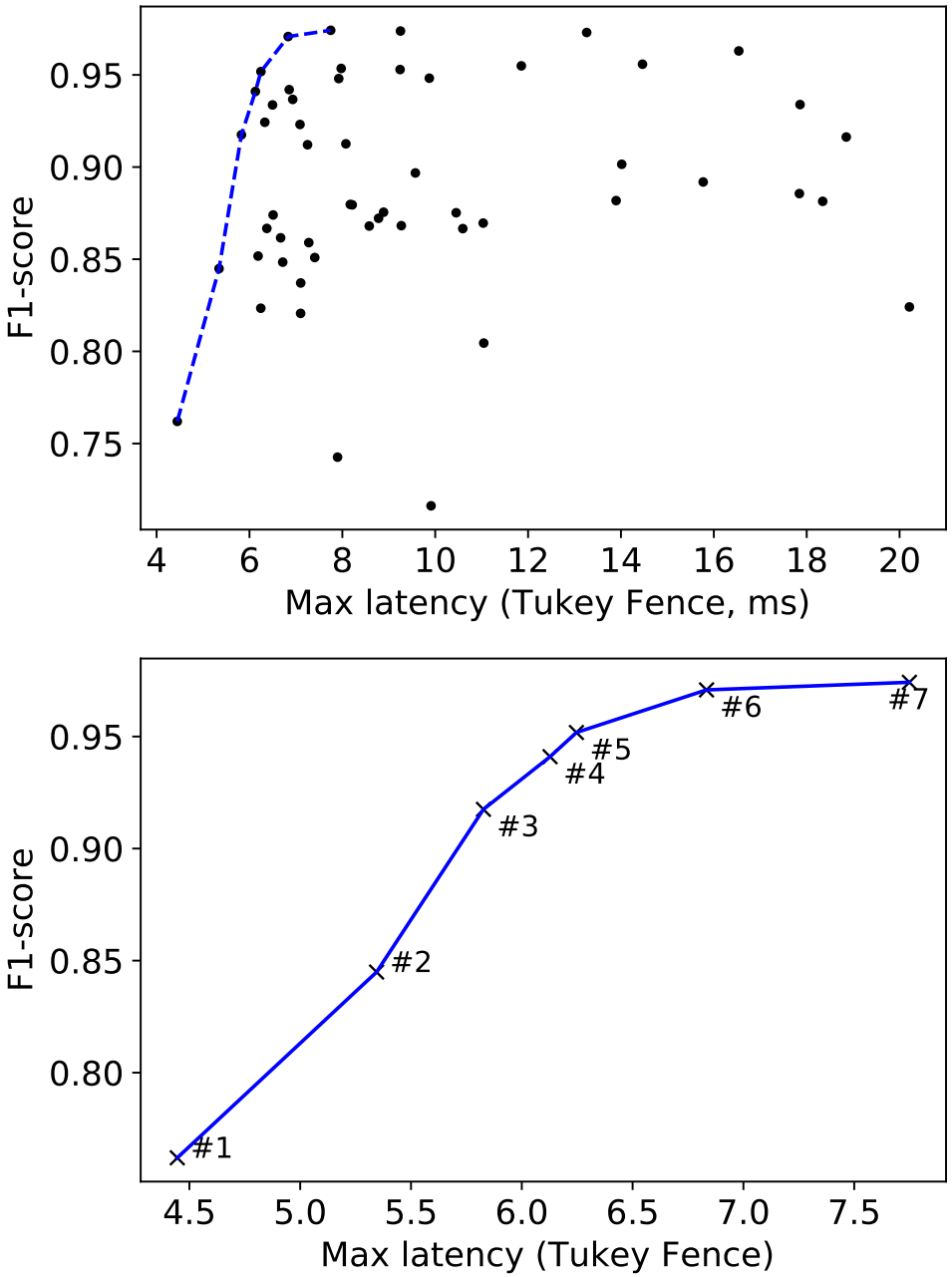


Figure 4.11: *Solutions of the multiple optimization instances, represented in an alternative way to maximize f1-score while minimizing the maximum latency, instead of its variability. These graphs are displayed to demonstrate alternative possibilities for optimization, while the Pareto front in Figure 4.10 was used to select the final solution.*

4.5 Summary

In this chapter, we presented a procedure to optimize the performance of parametric onset detectors for a set of data of interest. Our approach focuses on improving the accuracy of detection and reducing the time delay, i.e., latency, between the actual onset and its reporting.

We successfully applied our proposed technique to the onset detectors of the Aubio library. An Evolutionary Computation algorithm was used to achieve automated optimization of the input parameters of each detector, and the Pareto front was used to outline the set of best solutions. This method was compared to manual optimization, proving to significantly reduce the amount of man-hours effort required while producing comparable results. It can be argued that the proposed approach employed an execution time that was not extremely dissimilar from the manual optimization, especially when considering it along with the time required to develop the proposed approach. However, it is to be noted that the entirety of the execution of the proposed approach was automated and unsupervised, thus the notable reduction in valuable man-hours. Furthermore, the proposed approach can be applied to new datasets very quickly. In particular, the optimization code was made available online[9].

When comparing the F1-score values obtained using the evolutionary computation algorithm to those achieved through manual optimization, we observed an average difference of 1.4×10^{-3} F1-score points with a standard deviation of 1.2×10^{-2}. Furthermore, the automated algorithm took 13 hours and 34 minutes to compute the best results, while the manual procedure required over two working days of human effort. By employing this approach, we managed to enhance the effectiveness of a real-time onset detector significantly, while also reducing the time it takes to detect onsets. These enhancements were evaluated using a separate test dataset.

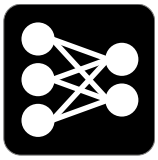
Using only a subset of an audio dataset may pose limitations: the suggested method could derive greater benefits from utilizing the entire dataset of interest. However, this would result in increased human labeling and optimization time. To accelerate optimization in such scenarios, the exploration could be narrowed down to more pertinent parameter ranges. Moreover, the number of parallel optimization instances could be increased, especially with a more powerful computer. A higher degree of parallelism could also be exploited by allowing the evolutionary algorithm to

[9] https://github.com/domenicostefani/BioInspiredOnsetDetection

compute the fitness of more than one individual simultaneously, on multiple processing threads. Additionally, we did not compare the proposed method with grid-search algorithms, as they involve scanning a large number of combinations of parameters without the ability to automatically focus the search on potential optima in the search space. However, a comparison between the proposed approach and grid search would further help clarify the strengths and drawbacks of the proposed method. Finally, the optimization performance may be further improved by using fully multiobjective EC algorithms, such as NSGA2.

The proposed procedure is expected to extend to any parametric onset detector algorithm.

Chapter 5



Comparison of Deep Learning Inference Engines for Embedded Real-time Audio Classification

Recent advancements in deep learning have demonstrated great potential for audio applications, significantly surpassing the precision of previous approaches to tasks like music transcription, beat detection, and real-time audio processing. Concurrently, the proliferation of increasingly powerful embedded computers has led deep learning framework developers to devise software optimized for executing pre-trained models in resource-constrained contexts. Consequently, the use of deep learning on embedded devices and within audio plugins has become more widespread. However, confusion has been rising around deep learning inference engines (IEs), regarding which of these can run in real-time and which are less resource-hungry. This chapter presents a comparative analysis of four deep learning IEs focused on real-time audio classification on the CPU of an embedded single-board computer: TFLite, TorchScript, ONNX Runtime, and RTNeural. The results show that all the investigated IEs compared are capable of executing neural network models in real-time, provided appropriate code practices are implemented. However, the execution times vary across engines and models. Notably, the study reveals how most of the less-specialized engines offer remarkable flexibility and can be effectively used for real-time audio classification,

often yielding slightly superior outcomes compared to real-time-specific approaches. Conversely, more specialized solutions present a lightweight and minimalist alternative, particularly suitable for use cases that demand reduced flexibility.

This chapter discusses one of the contributions presented at the 25th International Conference on Digital Audio Effects [109].

5.1 Introduction

Deep learning has witnessed widespread adoption across diverse domains of data processing and analysis, prominently within audio and music processing. Numerous sound-related tasks have benefited from the successful application of deep learning methodologies. For instance, deep learning has been effectively employed in music tagging [173], beat detection [110], onset detection [125], instrument classification [174], and more recently, in real-time audio processing [104]. Research in deep learning for audio and music has often focused on offline learning, where deep learning models have substantially supplanted traditional machine learning approaches, achieving superior accuracy and lower error rates. In contrast, real-time execution of deep learning includes additional difficulties on the development part, in the form of stringent real-time execution deadlines and the intrinsic time-consuming nature of neural network executions, which often surpass the speed of conventional machine learning counterparts.

In recent years, there has been a growing interest concerning the deployment of deep learning algorithms in practical, real-world applications through the use of embedded computers. This trend is particularly pronounced in the domain of real-time audio processing, giving rise to the development of numerous embedded platforms tailored for audio applications. Examples of such platforms include Elk Audio OS [24], Bela [23], and Prynth [101]. These platforms have significantly contributed to advancing the capabilities and applications of real-time audio processing within the domain of deep learning.

However, the requirements of deep learning and conventional frameworks can particularly stress the limited computational resources of even the most recent embedded devices. This has led to a surge in the availability of deep learning IEs tailored for embedded devices and single-board computers. Within the realm of deep learning, terminology such as “inference engine”, “inferencing library”, or “runtime” are used to refer to tools, specifically code libraries, capable of executing pre-trained neural net-

works. These tools play a crucial role in deploying and leveraging pre-trained neural networks effectively.

Some of these IEs derive from deep learning frameworks such as TensorFlow, PyTorch, and ONNX. Despite their availability, it is unclear whether they can safely execute neural network models safely in real-time audio contexts. In these contexts, it is imperative to avoid any operation that could potentially slow down or halt audio signal processing. This uncertainty has led developers to devise their specialized approaches for deep learning inference in real-time audio (e.g., RTNeural [108], and [104]). However, specialized approaches often suffer from limited flexibility. In contrast, widely used deep learning IEs have the ability to load a wide range of neural network models. Additionally, it is not clear whether the same exact model can be executed more quickly with an IE than another.

In this chapter, we conduct a comparative analysis of four deep learning IEs for real-time audio classification on an embedded CPU. Specifically, we evaluate TensorFlow Lite (or TFlite, from TensorFlow, Google), TorchScript (from Torch/PyTorch, Facebook's AI Research lab), ONNX Runtime (from ONNX, Microsoft), and RTNeural [108]. We conduct a comprehensive comparison of these tools based on various critical aspects including their compliance with real-time-safe programming principles[1] [127], execution speed across multiple neural network models, utilization of computational and memory resources, ease of use, and the quality of documentation.

Our evaluation focuses on CPU-based model execution, given that this is the primary viable option for readily available embedded SBCs. More often than not, these devices are not equipped with a deep-learning-enabled GPU (i.e., a Nvidia CUDA compatible GPU). The comparison of even more specialized approaches, such as the use of TPUs, DPUs, and FPGAs, is outside of the scope of this study. We conducted our comparison using a Raspberry Pi 4 single-board computer coupled with Elk Audio OS, an open-source, real-time operating system optimized for low-latency embedded audio processing. The Raspberry Pi has gained widespread usage in deep learning, being supported by a multitude of inference engine developers. In addition, Elk Audio OS enables low-latency audio processing thanks to its real-time capabilities. The neural networks used for this comparison were tailored to classify eight distinct expressive guitar techniques. The output of each of these models represents a prediction of the technique used for each note, in the form of the distribution of probability over all the classes. The code relative to this project was made available

[1] http://www.rossbencina.com/code/real-time-audio-programming-101-time-waits-for-nothing

in an online repository [2].

This chapter is structured as follows: Section 5.2 presents the background regarding the use of deep learning for music applications and embedded computing platforms within the musical audio domain. In Section 5.3, comprehensive descriptions are provided for the deep learning IEs, evaluation metrics, and models used in our comparative analysis. The outcomes of the comparison are discussed in Section 5.4. Lastly, Section 5.5 encompasses a summary and our concluding remarks and insights.

5.2 Background

Neural networks have been applied successfully in contexts like onset detection, such as in the work of Eyben et al. [122], where the authors presented a bidirectional Long short-term memory (LSTM) network that was able to surpass previous state-of-the-art scores. A similar neural network was applied to the problems of beat detection and tracking [110], achieving state-of-the-art results. Similarly, Gómez et al. presented a successful approach to instrument classification that uses a convolutional neural network [174]. The authors managed to improve the accuracy of their method through the use of a pre-processing step based on source separation, and a transfer learning approach. However, most of the deep neural networks proposed in research for audio tasks focus on offline inference and are generally unfit for real-time usage, as they can either result in computationally expensive operations (e.g., in [174]) or require non-causal information (e.g., the bidirectional network of [122]).

An interesting approach is described by Sigtia et al. in [112], where a neural network was employed for polyphonic transcription of piano performances. While the main solution presented by the authors was rather computationally complex, the authors proposed the use of an optimized search algorithm for real-time contexts. Moreover, Bock et al. [125] presented a new version of their offline onset detection model, designed to operate in real-time contexts. This approach only uses causal information from the audio signal presented and it reduces the latency between onsets in an input audio signal and the moment at which they are reported. More recently, Wright et al. [104] introduced an end-to-end neural approach to audio processing, which managed to model several distortion pedals and guitar amplifiers. The authors presented a real-time implementation, which is implemented by scratch using a linear

[2] https://github.com/domenicostefani/deep-classf-runtime-wrappers

algebra library (i.e., Eigen) and executed on a desktop computer. This is a very specialized approach that can help produce extremely optimized code, but it completely lacks the flexibility of deep learning IEs.

These IEs, which include TFLite, TorchScript, and ONNX Runtime, can easily load almost any neural network model during execution, without recompiling any code. However, as mentioned in Section 5.1, there is a general confusion around the compatibility of these IEs with real-time audio applications, which require that the code does not contain any non-real-time-safe operation that can slow down the processing of the audio signal. This forms a central aspect of the investigation in this study, aiming to provide clarity and insights into the performance and real-time suitability of these IEs for audio processing.

For this reason, Chowdhury [108] developed RTNeural, which is a "neural inferencing library" (i.e., deep learning IE) designed to be used for real-time audio applications and similar deep learning tasks that must meet hard time deadlines. The library's design and functionality are centered around supporting essential but meaningful deep learning operations, with plans for further expansion to include a broader array of neural network layers. The author compared the performance of the library against the PyTorch `C++` API (i.e., TorchScript). However, the landscape of available IEs has evolved since then, especially with the availability of "lite" optimized versions of most frameworks. Moreover, while computation time is crucial for real-time applications, it is fair to compare different IEs also in terms of other metrics, such as how many neural operators are supported, the use of computing resources, memory, general ease of use, and quality of documentation.

Rtneural was successfully used for the implementation of an embedded guitar effect[3] which was successfully deployed on a Raspberry PI 4 with Elk Audio OS.

Along with the increase of real-time deep learning approaches for audio classification and processing, there has been a growing number of embedded devices for audio processing. Meneses et al. [175] presented a comparison of three open-source embedded audio platforms: *Prynth* [101], the *Bela* framework [23], and a custom processing unit. According to the authors, all the solutions presented different characteristics with no clear winner. More recently, the Elk Audio OS [24] was presented as an open-source real-time operating system for embedded hardware. Similarly to Bela, Elk Audio OS uses the Xenomai Cobalt real-time kernel to handle low latency audio processing, but it is not limited to a single hardware platform and it offers high-

[3] github.com/GuitarML/NeuralPi/releases/tag/v1.3.0

definition audio inputs and outputs. The work by Vignati *et al.* [126] is significant in its comparison of two key real-time kernels: Xenomai Cobalt kernel and the Preempt RT kernel patch. The comparison aimed to evaluate the performance of these kernels, particularly in heavy processing applications. The results highlighted the advantages of the Xenomai Cobalt kernel, showcasing its superior performance compared to the Preempt RT kernel patch. This information is valuable for developers and engineers seeking to optimize real-time processing for their applications, providing insights into the strengths and weaknesses of different real-time kernel options.

More recently, the work by Vandendriessche et al. [117] delved into exploring hardware acceleration possibilities for deep learning inference in audio processing. The study investigates the potential of using specialized hardware like TPUs, FPGAs, and similar devices to accelerate deep learning inference. However, the authors acknowledge that these solutions are highly specialized and may not be universally accessible across various platforms due to factors such as cost or hard real-time requirements. Therefore, the focus of this comparison remains on CPU-based inference, which is viable for both embedded implementations and desktop audio plugins, providing a practical and widely applicable approach.

In a similar vein, model-specific optimizations such as weight quantization and other techniques can be instrumental in alleviating the computational load on constrained computing devices. Lane et al. [176] provided a comprehensive overview of limitations imposed by mobile and embedded devices, suggesting strategies to mitigate these challenges. The authors proposed a sparse coding approach that effectively reduces the utilization of computational resources, showcasing promising results in tasks like speaker recognition and acoustic environment classification by significantly reducing model size. However, it is important to note that these optimizations may lead to a reduction in model accuracy, the extent of which depends on various parameters, including the structure of the target neural network. Given the focus of this comparison, which centers on general-purpose deep learning IEs, model-specific optimizations fall beyond the immediate scope of this study.

5.3 Methodology

This section outlines the details of the comparison, which includes the selection of deep learning IEs, the benchmarking task, the neural networks under evaluation, and the key metrics of interest.

This study focuses on comparing various deep learning IEs for inference on embedded CPUs. Hence, acceleration capabilities utilizing GPUs and TPUs will not be considered in this comparison. Additionally, model optimizations such as weight quantization and pruning are beyond the scope of the comparison.

5.3.1 Inference Engines

The comparison will comprise the following deep learning IEs:

1. **TFLite** 2.4.1[4]: TFLite is an IE provided by TensorFlow (Google) for executing neural network models on embedded devices. The TFLite system includes a converter that transforms models created and trained in TensorFlow into `.tflite` files, and an Interpreter that can load these TFLite models and perform inference.

2. **TorchScript** 1.10.0[5]: TorchScript is a system provided by PyTorch developers that allows models trained in a Python environment to be converted into code that can be executed in environments without Python dependencies.

3. **ONNX Runtime** 1.7[6]: ONNX Runtime is an IE provided by Microsoft specifically for ONNX (Open Neural Network Exchange) neural network models. It is said to be designed to deliver significant speed improvements for both training and inference of neural networks due to its optimization and acceleration capabilities.

4. **RTNeural**[7]: RTNeural [108] is a custom IE designed specifically for audio processing in hard real-time contexts. It offers a compact and easy-to-use library, although it supports only a limited range of neural layers. At the time of writing this contribution [109], RTNeural did not support functionalities like Max-Pooling and Batch Normalization layers. Despite its limitations, RTNeural can be a strong competitor to more popular alternatives in certain audio processing contexts. It provides both a dynamic model loading mode, like other IEs, and a compile-time mode intended to reduce execution time for small networks, as mentioned in the library's documentation.

[4] https://github.com/tensorflow/tensorflow/releases/v2.4.1
[5] https://github.com/pytorch/pytorch/releases/tag/v1.10.0
[6] https://github.com/microsoft/onnxruntime/releases/v1.7.0
[7] https://github.com/jatinchowdhury18/RTNeural
commit: `be0ccbc6b6ed180ba6ef65896628ae4ab2a25362`

5.3.2 Task

In this comparison, we assess the performance of four distinct IEs outlined in Section 5.3.1 using a series of neural network models for real-time classification of expressive guitar playing techniques. The input of each model is a set of features from the very first milliseconds of a note in the signal, while the output represents a probability distribution across eight specific techniques. For this task, the neural classifier constitutes the final stage of an execution pipeline, which includes an onset detector and a set of feature extractors, as illustrated in Figure 5.1. Only one classification model is needed, but we use three models that differ in size (therefore computational complexity of inference) for this comparison.

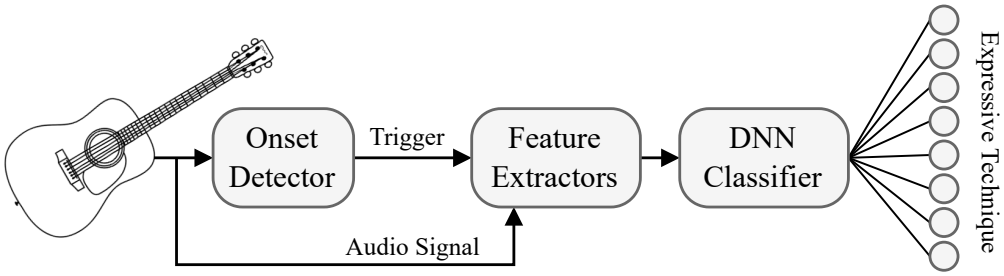


Figure 5.1: *Expressive guitar technique Classification pipeline.*

In this setup, upon detecting a note onset, the feature extractors compute various timbral features (e.g., MFCC, BFCC) from the audio signal. These features are arranged in a one-dimensional vector and fed as input to a classifier model. Our models are tailored to classify eight distinct categories of expressive guitar techniques, therefore, each classification model has 8 output neurons, where the one with the highest activation value represents the predicted technique. As a consequence, the neural network models compared are all examples of FFNNs with no recursions.

In our application, the classifier operates within a computation deadline of 20 ms from the generation of a note onset [135]. This constraint arises because the classification outcome is employed to create new sounds that need to feel simultaneous to the input sound for human auditory perception. Generally, the human auditory system struggles to differentiate between complex tones separated by less than 30 ms [21]. Adhering to a strict 20 ms deadline allows for the utilization of complex synthesis algorithms based on the classification result. Unlike tasks with rigid hard real-time constraints, such as audio processing, this classification task involves a soft real-time deadline. Hence, we can execute it on a high-priority thread distinct from

the hard real-time processing of the input/output audio signal [80].

5.3.3 Models

The models chosen for our comparison are the following:

- **Model A**: the first model is a FFNN composed of four dense hidden layers with 800 neurons each, an input layer with 180 neurons, and a final layer with 8 outputs. Batch normalization was used between each hidden layer with a positive impact on model accuracy. The model comprises a total of 2,083,208 parameters.
- **Model B**: Model B is a smaller version of Model A, with six hidden layers of 350 neurons each, an input layer with 173 neurons, and 8 model outputs, resulting in a total of 677,958 model parameters. Model B was included to test the performance of the RTNeural framework, which a the time of writing this contribution, did not support the Batch Normalization layers used in Model A.
- **Model C**: The last model is a drastically smaller version of Model A, with one dense hidden layer with 350 neurons, 173 inputs, and 8 outputs, resulting in a total of 63,708 parameters. Model C was chosen because Model A and Model B were found to be too large to be executed in the real-time execution thread[8], so only soft real-time constraints can be guaranteed with those models. While this is allowed by our specific task, we offer a comparison that includes the capabilities of each IE to execute models in the real-time thread, which would be required by any model that performs signal processing. The small size of Model C ensures that execution will take less than the time budget between audio interrupts. Model C is used to verify whether the IEs compared here can run without breaking real-time processing constraints (e.g., not allocating dynamic memory or waiting for lower priority tasks and mutexes).

The optimization and choice of hyperparameters are not relevant to this particular chapter, as accuracy is not a metric of comparison that depends on the IE used. All the models were defined and trained using TensorFlow and the Keras Sequential API. Subsequently, each model was converted to the required formats for the respective IE. Details regarding the conversion process can be found in Section 5.4.5.

[8]The audio plugin created for this task was executed at the rather fast pace of 64 samples at a sample rate of 48 kHz (i.e., 1.33 ms in between audio interrupts) for low-latency audio input and onset detection.

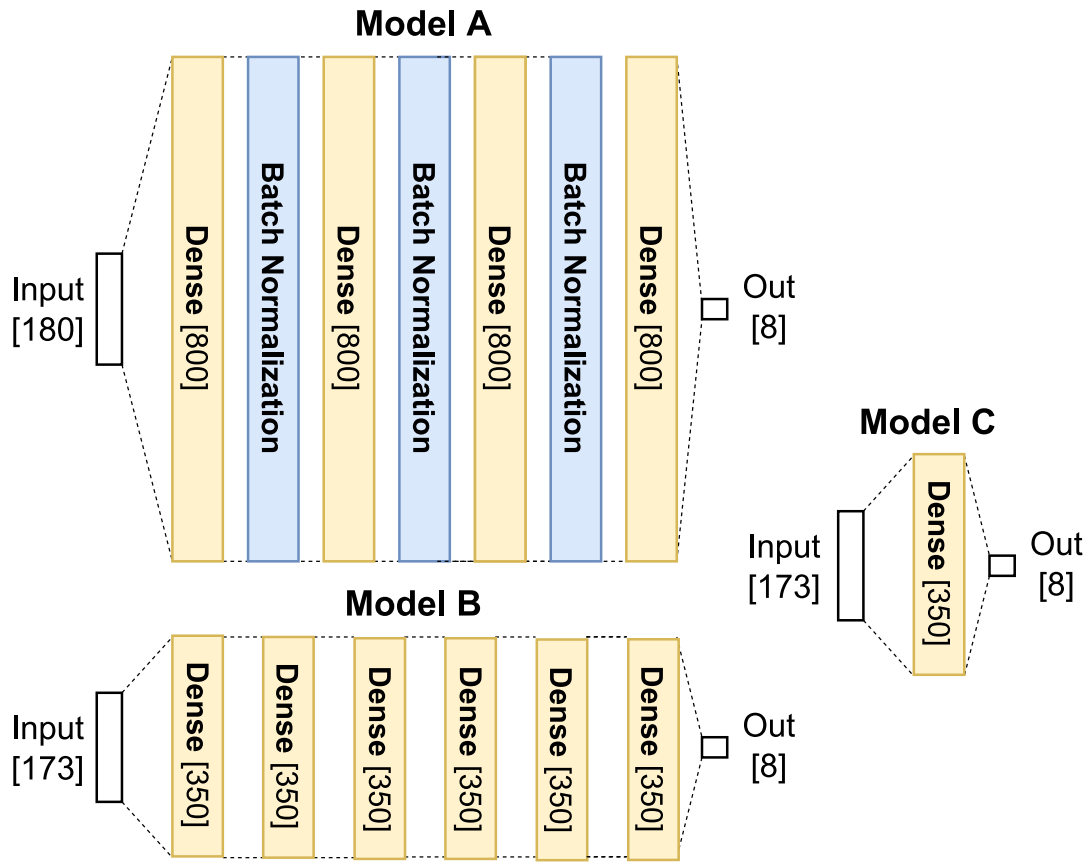


Figure 5.2: *Architecture of the neural network models used for the comparison.*

5.3.4 Metrics

Each IE is compared in terms of the following eight metrics:

1. Real-time safety;
2. Model Execution time;
3. Usage of computation resources (CPU and RAM);
4. Model footprint (file size);
5. Library footprint (library object size);
6. Supported operations;
7. Ease of use;
8. Quality of documentation.

Table 5.1: *Compatible combinations of models and IEs compared in our analysis. The dynamic-model-load mode (**R.load**) and the compile-time-definition mode (**C.def**) of RTNeural are tested separately. The sole Model C is tested by running the IEs in the real-time thread (for the reasons described in Section 5.3.2 and Section 5.3.3), while the remaining are loaded on a separate high-priority thread.*

	TFLite	**TorchScript**	**ONNX Runtime**	**RTNeural**	
				[R.load]	**[C.def]**
MA	✓	✓	✓	✗	✗
MB	✓	✓	✓	✓	✓
MC	✓(RT thread)	✓(RT thread)	✓(RT thread)	✓(RT thread)	✓(RT thread)

First and foremost, our analysis focused on determining whether each IE could perform inference safely on a real-time thread. Safety in this context refers to the absence of code operations that could take an "unbounded" amount of time to complete. Because of the Xenomai real-time kernel used by Elk Audio OS, unsafe operations or system calls performed from the audio thread trigger a mode switch, returning control to the Linux kernel. Mode switches are logged by the system, aiding in the identification of non-safe operations in the real-time thread. Consequently, with Model C, each IE was executed within the real-time thread to assess their safety.

In addition to assessing safety, we also aimed to determine which IE was the most efficient in terms of executing the same models. Efficiency is crucial in a real-time context as it constrains the minimum latency achievable by the system. Execution time was first measured in an isolated context by running each IE from the Linux shell, independently of the classification pipeline. This provided a measure that was unaffected by many parameters of our classification pipeline. However, it did not account for delays that could be present when deploying the complete pipeline. Hence, execution time was also measured within the deployed audio classification plugin. For Model A and Model B, executed on a high-priority thread separate from the real-time execution, the second measurement includes the time required to schedule the inference and any delays stemming from real-time audio processing running in the foreground.

In our evaluation, we also monitored the average CPU and RAM usage during the execution of the audio classification plugin using the process status command (`ps`) in Linux and the Xenomai kernel. This allowed us to estimate the resource consumption associated with each IE. Furthermore, considering that each model needed to be

converted to a specific format for each IE, we measured the size of the resulting model files. This information is crucial, especially in the context of limited storage on some embedded computers, to ensure that the models can be accommodated within the available storage space.

The first three metrics (i.e., **Real-time safety**, **Model Execution time**, and **Usage of computation resources**) were measured on a Raspberry PI 4 single-board computer (4 GB RAM model). The Raspberry PI board runs the Elk Audio OS (v0.9.0), based on the the Xenomai Cobalt Kernel[9]. The computation times for standalone execution were obtained by averaging across 17,604 executions for each model-IE combination. "Deployment" execution times were averaged across 768 executions, triggered on the classification plugin that was deployed on the target embedded system. The 768 executions were triggered from as many guitar notes in a 26-minute audio signal that was streamed to the embedded board in real-time for each combination of model and compatible IE. Due to the time-intensive nature of this process, the number of executions was limited. Additionally, **Model footprint** was simply defined as the file size of the models in the format of each IE. Furthermore, we evaluated four metrics that are specific to the different deep learning IEs and independent of the models.

Library footprint (library object size): Given that memory storage can be a constraint on embedded devices, we measured the total size in MiB of the shared or static library objects needed for each IE. Each library was compiled for the Linux AArch64 architecture (ARM64).

Supported operations: We compiled a list of the most common and widely used neural layer types and assigned a score to each IE reflecting the fraction of operations supported. In Deep Learning, it is crucial that IEs support a wide range of operations, allowing for flexibility in model architecture and training. The list of main layers used for comparison is composed of *Dense*, *Gated Recurrent Unit*, *LSTM*, *1D* and *2D Convolution*, *1D* and *2D MaxPooling*, and *Batch Normalization* layers. Additionally, the list of activation types includes: *TanH*, *Sigmoid*, *Softmax*, *ReLU*, *Leaky ReLU*, and *PReLU* activations.

Ease of use: While challenging to quantify precisely, we aimed to evaluate the ease of use of each IE. This encompassed factors such as the simplicity of converting a pre-trained model to the target format, as well as the usability of the APIs for model loading, retrieval of model properties, and inference execution. Two of the authors

[9] https://xenomai.org/

responsible for the implementation (including the author of this thesis) assigned a score ranging from zero to ten for each category. The scores were then averaged to derive a final score for this metric.

Quality of documentation: This metric evaluates the quantity and quality of documentation available for each deep learning IE. Similar to the ease of use metric, two authors individually assigned a score ranging from zero to ten to each IE with respect to the quality of its documentation. The scores provided by the two authors were then averaged to determine the final score for this metric.

5.4 Results and discussion

This section presents the results of the comparison between the deep learning IEs mentioned in Section 5.3.1, according to the metrics described in Section 5.3.4.

5.4.1 Real-time safety

The real-time safety capabilities of the deep learning engines were evaluated by running the inference of Model C in the real-time audio thread while monitoring the status of the Xenomai Cobalt real-time kernel. Interestingly, all the IEs compared here were able to execute multiple subsequent inference operations without causing audio glitches or a significant increase in mode switches at run-time.

However, it is worth noting that both TorchScript and ONNX Runtime did generate a single mode switch each during the initial execution of the model.

In the case of TorchScript, the identified non-safe operation was the allocation of a `std::vector<c10::IValue>` item, which consistently occurred during the first call of the `forward` function. To address this, a single inference operation was executed during the first call to the real-time audio processing method, acting as a "priming" operation. This priming operation triggers an early allocation of memory in the classifier at the program's startup, where delays that are due to non-safe operations can be tolerable. Consequently, when the first actual classification is needed, non-real-time-safe operations are avoided.

In the case of ONNX Runtime the culprit of the mode-switch was the call to `Ort::Run()`, which was causing memory allocation on its very first call. The same "priming" approach was successfully applied to ONNX Runtime.

5.4.2 Execution time

The execution times of the models varied across different deep learning IEs, even when using identical models. This variation was assessed through both isolated executions and in the context of an audio plugin running on the real-time Elk Audio OS (see Section 5.3.4). The resulting measurements for both scenarios are presented in Figure 5.3.

The results show that, while most of the IEs offer rather comparable performance in terms of execution time (especially with smaller models), TorchScript is consistently slower than the alternatives. Aside from TorchScript, the two alternatives among the popular IEs (i.e., TFLite and ONNX Runtime) averaged comparable times with Model B, while TFLite slightly prevailed on the smaller Model C and ONNX Runtime worked better with the bigger Model A. The difference for Model A was reduced when running the deployment tests, which indicates that ONNX Runtime could be performing better optimizations on larger numbers of operations.

Additionally, RTNeural showed slightly longer average execution times compared to TFLite and ONNX Runtime, but the differences were small and the performance was still comparable, as opposed to TorchScript. Moreover, RTNeural was tested with both of its model loading modalities: run-time dynamic model parsing and compile-time model definition. Interestingly, the two modalities exhibited virtually identical performance in all the tests, contrary to the expectation of quicker inference with the compile-time model definition as indicated in RTNeural's documentation. This may be attributed to the larger size of the models used in the comparison, which generally exceed the sizes RTNeural was specifically designed for. It is important to note that RTNeural was used with the Eigen backend, as suggested by the developer for larger networks. However, RTNeural also supports the use of xsimd or the `C++` STL, which might yield different results and could be explored in future investigations.

As expected, the standard deviation of the execution time is minimal for all standalone execution tests and the execution of Model C in the real-time thread of the deployment application. On the contrary, the deployment setups present some variability in the execution time of Model A and Model B, which is due to the synchronization between the audio processing routine and the inference thread (which are separate for Model A and Model B in the deployment setup). In particular, in a few cases, the results of the classification reach the real-time thread one or two audio interrupts late. These delays represent a maximum of 6.59% of classifications where the results arrive one interrupt later than the average (i.e., 1.33 ms), and a

mere 0.37% that arrive two interrupts late (i.e., 2.67 ms). The consistency of these delays confirms that they are primarily caused by synchronization issues between the real-time execution and the classification thread, rather than being inherent to the deep learning IEs themselves.

5.4.3 Computational resources

The usage of CPU and RAM was monitored during the execution of the audio classification plugin thanks to the process status command (`ps`), the `top` command, and the utilities of the Xenomai kernel. The usage metrics for each configuration were averaged across a test that lasted 25 minutes and 50 seconds, during which 768 inference operations were executed. On average, an inference operation occurred approximately every 2 seconds. The detailed results for each test, showcasing CPU and RAM usage, are presented in Table 5.2.

Table 5.2: *Usage of CPU and RAM for each combination of model and compatible IE. "Avg.CPU" represents the CPU usage of the main Linux system, specifically non-real-time tasks, while "Avg.CpuX" reports the CPU usage by real-time tasks in the Xenomai kernel. The average CPU and memory usage were measured using the* `ps` *and* `top` *commands. The percentage measures are relative to the embedded system described in Section 5.3.4, while Avg. Phys. Mem. and Avg. Virt. Mem. are measured in MiB.*

Model	Runtime	Avg. Cpu	Avg. CpuX	Avg. Mem %	Avg. Phys. Mem. (MiB)	Avg. Virt. Mem. (MiB)
MA	TFLite	8.7 %	6.1 %	5.2 %	196	1158
	TorchScript	9.0 %	5.8 %	8.8 %	333	1551
	OnnxRuntime	10.7 %	6.1 %	6.5 %	246	1335
MB	TFLite	8.0 %	5.9 %	5.0 %	191	1217
	TorchScript	8.6 %	5.9 %	8.6 %	327	1545
	OnnxRuntime	9.6 %	5.9 %	5.8 %	222	1310
	RTNeural(R.time)	8.4 %	5.8 %	5.7 %	217	1242
	RTNeural(C.time)	8.6 %	5.8 %	5.7 %	215	1241
MC	TFLite	6.1 %	6.2 %	4.7 %	180	1139
	TorchScript	5.3 %	5.8 %	7.7 %	292	1254
	OnnxRuntime	5.1 %	5.8 %	5.2 %	198	1224
	RTNeural(R.time)	5.2 %	5.8 %	4.8 %	181	1141
	RTNeural(C.time)	5.1 %	5.8 %	4.8 %	182	1142

The low frequency of classification operations indeed results in a low average utilization of computational resources. However, it is important to note that differences in resource usage between different IEs and models could become more significant at

higher operation frequencies, such as those encountered in audio processing. In terms of memory consumption, TorchScript consistently exhibits the highest memory usage percentage, which aligns with the inference time results (see Section 5.4.2). Additionally, for both Model A and Model B, ONNX Runtime demonstrates higher CPU usage on the standard Linux kernel compared to the alternatives. It is notable that all IEs exhibit higher CPU usage from operations running in the non-hard-real-time domain for both Model A and Model B, as expected since inference for these models is executed on a non-real-time thread. Conversely, CPU usage for all IEs on Model C is higher on the Xenomai Kernel operations, given that Model C is executed in the audio processing thread. These variations emphasize the importance of considering the specific use case and operational context when evaluating the performance of IEs.

Overall, the average use of CPU is demonstrated to not be a highly informative metric for low-rate audio classification, as it does not capture the nuances in performance that can be better highlighted by measuring execution times, as demonstrated in the previous section. The relative increase in RAM consumption from Model C to Model B and Model A is minimal despite the substantial difference in model sizes. This is to be attributed to the ample amount of RAM available in the latest iteration of the Raspberry PI single-board computer (4 GB), reflecting the significant technological advancements in modern embedded computers. It emphasizes the progress in hardware capabilities and how these advancements can influence the efficiency of handling larger models and computations even in resource-constrained environments. Absolute memory consumption metrics in MiB also show a rather small increase with larger models. Average memory usage increase between Model C and Model B is 14% with a maximum of 20%, and between Model C and Model A is on average 16% with a maximum of 24%. On the contrary, the number of parameters of Model B is 10.6 times larger than Model C, and Model A has 32.7 more weights than Model C, which indicates that, for these model sizes and at this rate of executions, most of the memory consumption could be overhead that is not to be attributed to model size.

It is worth noting that the CPU and RAM usage results obtained with the `ps` and `top` commands can be subject to measurement error, as they rely on averaging over large time windows. In this case, very sizable but brief consumption spikes will go undetected, while still hindering the performance of the hard real-time system. The use of more precise measurements like the Elk Audio OS' Sushi internal timings is crucial for reliable results.

5.4.4 Model footprint

When converting the original Keras models to the formats accepted by the various IEs, we noticed differences in the final file sizes. The file size of the models after conversion is an important consideration, particularly for devices with limited memory, as it impacts the storage and loading of models into memory during inference. In the case of TFLite, TorchScript, and ONNX Runtime, which use compressed model formats, the resulting file sizes are quite similar. On the other hand, RTNeural utilizes the JSON format, resulting in significantly larger file sizes for each model. This difference in file size is a trade-off between human readability (as JSON is easily human-readable) and storage efficiency. The results are shown in Figure 5.4.

Additionally, a test conversion of only the supported operations of Model A (i.e., no batch normalization) results in a final model file with a size of 94.9 MiB. Nevertheless, since RTNeural is an open-source project, any developer could integrate serialization and deserialization primitives to obtain lightweight models.

5.4.5 Model-independent metrics

The final four metrics are *Library size*, *Supported operations*, *Ease of use*, and *Quality of documentation* (See Section 5.3.4). These metrics are independent of the use of either Model A, B, or C. The scores obtained for these metrics are shown in Table 5.3 and visually represented in Figure 5.5, and their discussion follows in subsequent sections.

Table 5.3: *Model-independent metric results. All the scores are expressed on a scale from zero to ten, with ten being the most desirable score. The Library size score is inversely proportional to the actual size, with ten being the smallest library (RTNeural) and zero being the largest (TorchScript). The highest score for each metric is reported in bold.*

Inference Engine	Library size (score)	Supported Operations	Ease of use	Quality of doc.
TFLite	9.25	**10**	**9.75**	**9.5**
TorchScript	0	**10**	7.75	4.5
ONNX Runtime	8.98	**10**	7	8
RTNeural	**10**	5.7*	6.5	3.5

* With recent additions to the RTNeural library that followed the contribution presented in this chapter [109], this score would be 7.9. Considering PReLU as a manual substitute for LeakyReLU, the score would be 8.6.

Library Size

Library Size refers to the overall size of the C++ libraries for each IE. The corresponding sizes can be found in Table 5.4. The results reveal that TorchScript has a notably large code library, TFLite and ONNX Runtime are relatively similar in size, while RTNeural is significantly smaller by several orders of magnitude. However, it is essential to consider these measures in conjunction with the sizes of the models mentioned earlier.

Table 5.4: *Size of the C++ library objects for each IE. The version of each IE is specified in Section 5.3.1.*

Inference Engine	Library	Size (MiB)	Total (MiB)
TFLite	`libtensorflow-lite.a`	8.8	8.8
TorchScript	`libtorch_cpu.so`	116.8	117.3
	`libc10.so`	0.5	
ONNX Runtime	`libonnxruntime.so`	12.0	12.0
RTNeural	`libRTNeural.a`	0.026	**0.026**

From Figure 5.6, we observe that the relatively compact size of TorchScript models is offset by its substantial code library. Additionally, although the RTNeural library is extremely small to the point of being practically invisible on the plot, its large uncompressed JSON models pose a significant disadvantage. Moreover, if we project the size of Model A (which is currently unsupported), it exceeds 94 MiB, aligning the total size more with the significantly large TorchScript rather than the other options. However, the implementation of new serialization and deserialization functions for RTNeural (refer to Section 5.4.4) would significantly reduce the overall footprint since, at its current state, it is heavily influenced by the model size.

Supported Operations

The amount of supported operations is presented as a percentage representing the availability of various neural layer types and activations in each IE from a list of the most common types of neural layers and activations (see Section 5.3.4). As per their documentation, the more advanced TFLite, TorchScript, and ONNX Runtime each cover 100% of the listed operations. Conversely, RTNeural had more limited support, including only some operations such as *Dense*, *Gated Recurrent Units*, *LSTM* cells, *1D Convolutions*, and certain activations: *TanH*, *Sigmoid*, *Softmax*, and *ReLU*.

Hence, RTNeural achieves coverage for 57% of the commonly used neural network operations. Notably, at the time of writing this contribution [109], RTNeural lacked support for key operations like *2D convolutions*, *MaxPooling*, and *Batch Normalization*. From the time of writing this contribution [109], the development of RTNeural continued with the addition of more operators by the community. Notably, it now includes 2D convolutions, Batch Normalization, and more activation types, bringing the supported operations at 79%[10].

Despite supporting only a part of the most common operations, the code library of RTNeural is more than 300 times smaller than that of TFLite and over 450 times smaller than ONNX Runtime. Coupled with comparable execution-time results to the most popular alternatives, this underscores a significant emphasis on simplicity and avoidance of code bloat within RTNeural.

Ease of Use

Ease of use was assessed by the first two authors of the contribution this chapter is based on ([109]). It was assessed based on the perceived complexity of converting a neural model and utilizing the APIs of each IE to load the model, retrieve its properties (e.g., input and output sizes), and perform inference. It is important to note that the scores for ease of model conversion were influenced by the starting point being a standard Keras/TensorFlow model.

The conversion to a TFLite model is a straightforward process facilitated by a Python tool provided by the developers, namely TFLiteConverter. This tool allows the user to convert a SavedModel, a Keras model, or concrete functions easily. On the other hand, generating a TorchScript model from a TensorFlow model necessitates a custom implementation of a TensorFlow-PyTorch converter. Fortunately, both frameworks represent the fundamental layers in a relatively similar way, enabling a relatively simple conversion of data types from one library to the other. One detail to note is that PyTorch uses a different orientation standard for weight matrices, requiring the transposition of the data stored in the TensorFlow classes.

After obtaining a PyTorch model, the JIT API offers two methods to generate a TorchScript model: tracing and scripting. Tracing involves inferring the computational graph by recording the operations performed on a sample input. On the other hand, scripting creates the TorchScript model by analyzing the source code, making

[10]If we consider PReLU as a substitute for LeakyReLU, the percentage of operations supported by RTNeural would now 86%.

it a better choice for more complex models, particularly those involving conditional statements and more intricate logic. Although these models were simple enough to be converted through tracing, we still used scripting to have the guarantee of a complete conversion. Similarly, in PyTorch, users can opt for tracing to generate an ONNX model, which is a straightforward process. This is a relatively simple operation, but it does not provide ONNX Runtime a high score for model conversion since it requires the custom transformation of the original TensorFlow model into a PyTorch model. Lastly, for RTNeural, a Python script is available to export TensorFlow model weights to a JSON file. However, some refinement was necessary to discard layers not essential for inference, such as dropout layers.

The utility developed to convert TensorFlow models for each IE is available in the project's repository[11], along with wrappers for each IE, maintaining a consistent API and enabling seamless interchangeability between them.

Quality of documentation

The quality of documentation metric comprises the clarity and quantity of guides, tutorials, and formal API documentation. In this aspect, TFLite stood out with the best documentation, featuring user-friendly guides and comprehensive API documentation. ONNX Runtime followed closely, offering detailed API documentation and a reasonable number of examples, though it lacked the extensive tutorial resources for TFLite. On the other hand, TorchScript had relatively limited technical documentation for a project of its scale, relying mostly on a few incomplete guides. Lastly, RTNeural understandably had a minimal amount of information, which is to be expected by a project of its size, but its intuitive usability is reflected in the scores assigned in the previous section.

Notes on the sustainability of projects that use inference engines

It is worth noticing that the reliability of project dependencies throughout time, in terms of updates and patches, is a criticality of any sustainable and durable project [177]. In this context, this is particularly relevant for RTNeural, which is the effort of a small group of people. It is in fact a very well-known tendency for small projects to easily stop development and remain unupdated. Therefore adopting RTNeural as an inference engine, compared to more popular projects could be a risk. Table 5.5

[11] https://github.com/domenicostefani/deep-classf-runtime-wrappers

shows a simple comparison between the four IEs in terms of the start date, number of contributors, and number of commits. This highlights the rather small nature of the RTNeural project in comparison to the others.

Project	Start Date	Contributors	Commits
TFLite	2019 (Demo in 2017)	560 (3,480 for TensorFlow)	14,196
TorchScript	2018	N/D (3,080 for PyTorch)	67,825
ONNX Runtime	2018	579	10,224
RTNeural	2020	13	154

Table 5.5: *Comparison of project histories between the 4 inference engines compares (accessed on 29/12/2023).*

5.4.6 Comparison Results and Key Takeaways

Each IE proved to be safe for real-time inference when implemented with appropriate coding practices. Additionally, regarding model execution speed, TFLite, ONNX Runtime, and RTNeural displayed the fastest performance with largely comparable results, while TorchScript exhibited considerably slower performance. Notably, we observed that the compile-time definition mode of RTNeural did not yield a substantial speedup with the tested models.

The average CPU and memory usage proved to be less informative for a relatively low-frequency task. However, they did confirm the variations in resource consumption across different neural network models. Additionally, it is worth noting that, except for RTNeural, all the IEs use a compressed format for neural networks, resulting in considerably smaller model files compared to the human-readable representation used by RTNeural. Notably, TorchScript exhibited a significantly larger code library compared to all the alternatives.

All the popular IEs examined provide comprehensive support for a wide range of neural layers and activation functions. In contrast, RTNeural lacked, at the time this comparison was performed [109], some critical types of neural layers such as *Batch Normalization*, *MaxPooling*, and *2D Convolutions*. However, despite supporting 57% of the most common neural operators, the code library of RTNeural was found to be remarkably compact, differing by several orders of magnitude compared to the competition. Lastly, in terms of ease of use and detailed documentation, TFLite and ONNX Runtime were identified as the most user-friendly and well-documented options.

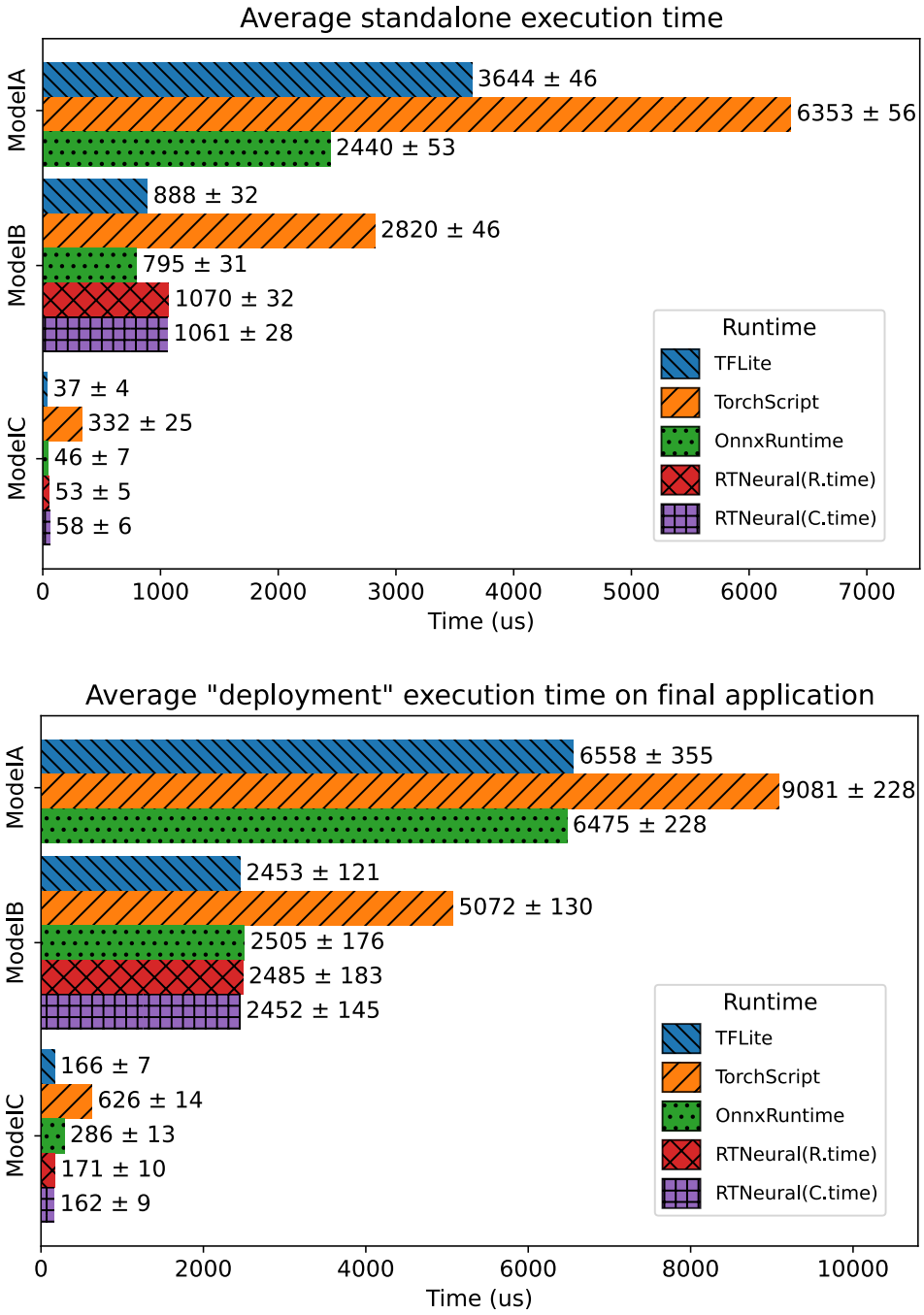


Figure 5.3: *Mean and standard deviation of model execution time in microseconds for each combination of model and compatible IE. The top image illustrates measurements obtained in an isolated context, while the image on the bottom encompasses the "deployment" execution time. These metrics shed light on the variability and performance of different IEs in handling the same models (see Section 5.3.4).*

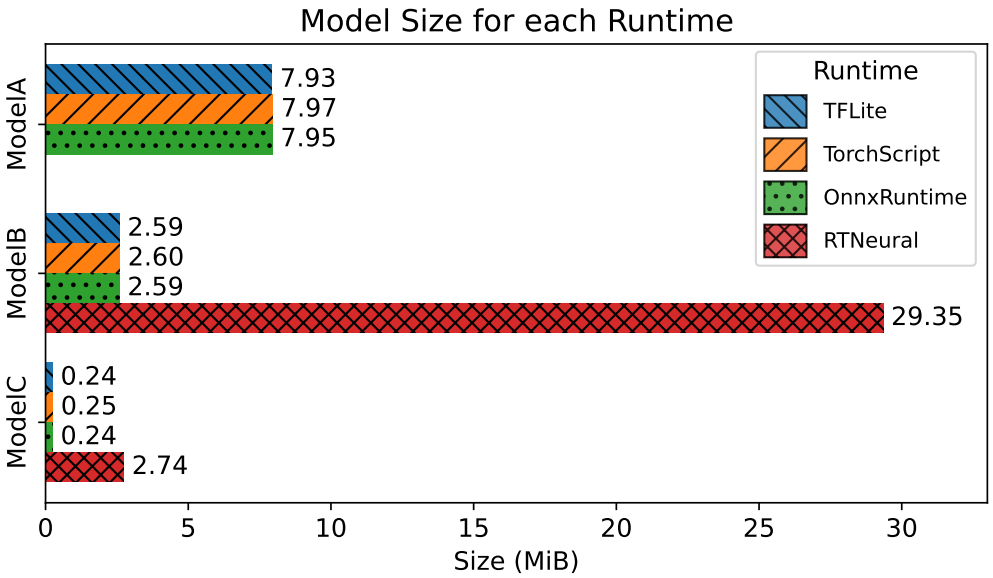


Figure 5.4: *Size of Model A, Model B, and Model C when converted for each IE.*

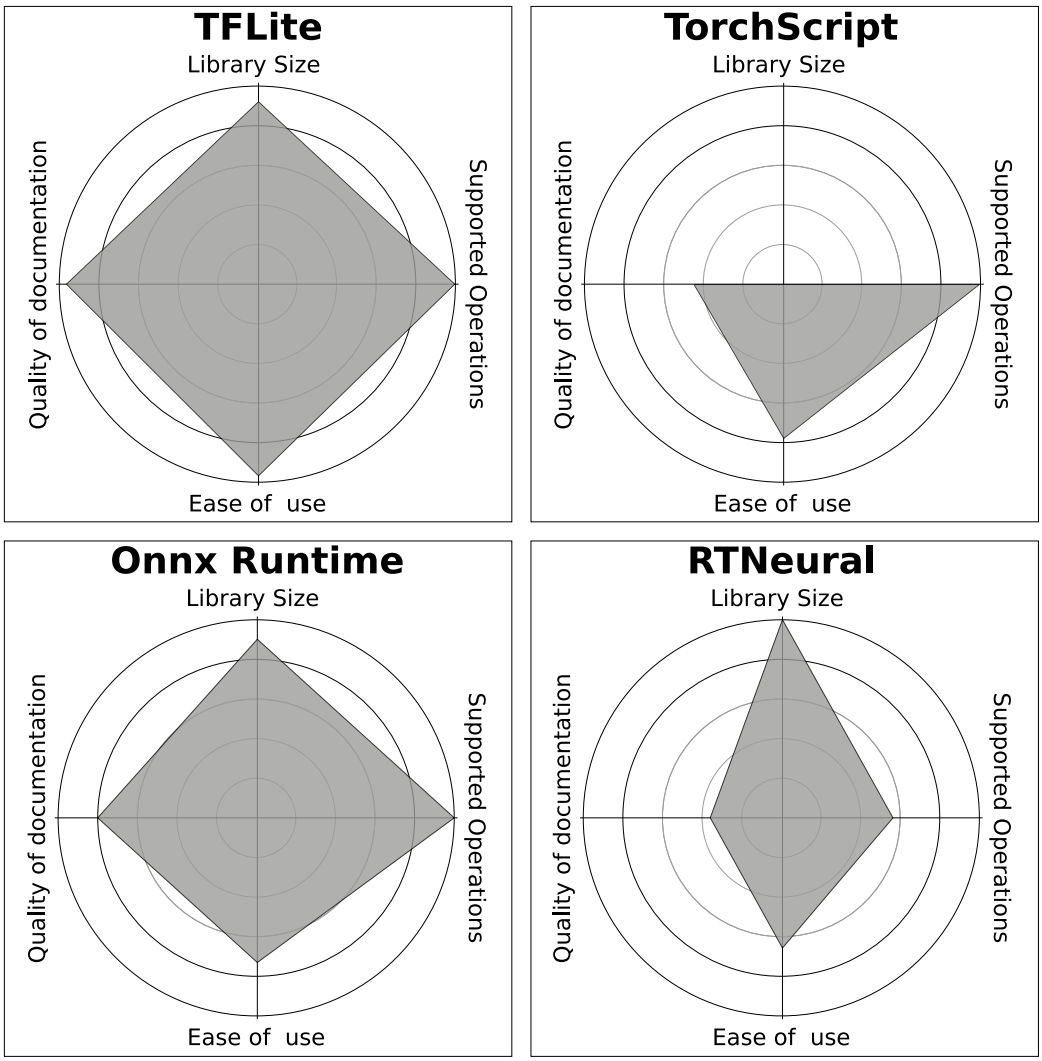


Figure 5.5: *Representation of the model independent scores for each IE. The four spokes on each graph represent respectively the score assigned to the size of the IE library, the number of supported operations, the perceived ease of use, and the quality of documentation.*

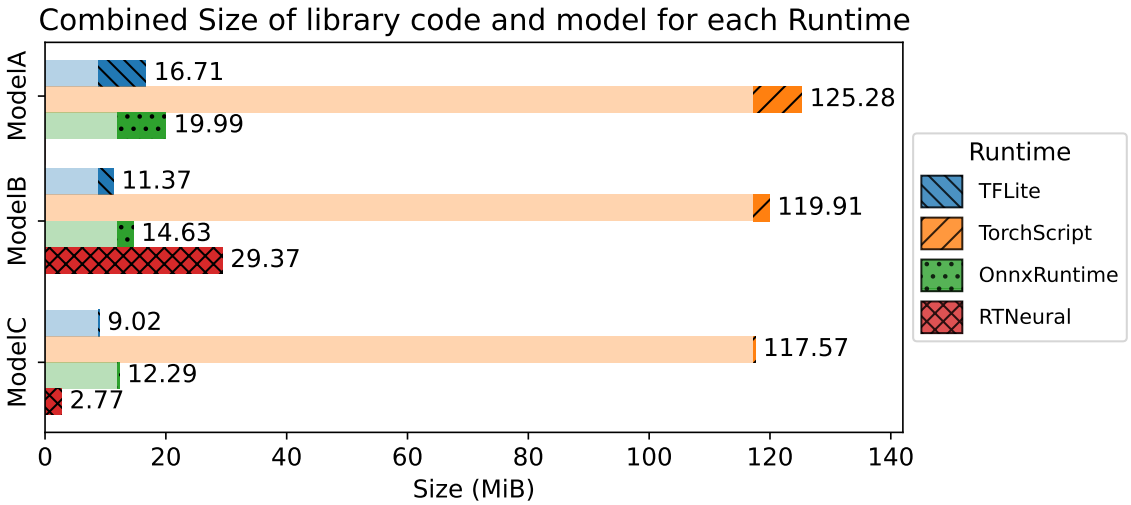


Figure 5.6: *Combined sizes of models and C++ code libraries for each compatible combination. The lighter-color part of each bar shows the size of each code library, while the remaining part indicates the size of the model.*

5.5 Summary

In this chapter, we presented a comparison of four distinct inference engines focusing on real-time audio classification using embedded CPUs. Our objective was to provide insights into optimized inference engines for efficient deep learning inference, especially in the context of real-time audio classification. For our study, we chose models geared towards classifying expressive guitar techniques in real-time. Our findings demonstrated that many well-known deep learning inference engines are well-suited for real-time audio classification, without needing to resort to specialized and more limited solutions. Conversely, some specialized solutions like RTNeural can serve as lightweight and minimalist alternatives, particularly in scenarios where flexibility is not a primary concern. Although the focus of this comparison was on embedded computers and audio classification, most results are likely to translate or scale to audio plugins for desktop computers, and audio processing. The limitations of this study are in the choice of restricting the comparison to Feed-Forward Neural Networks and only four deep learning inference engines. Besides exploring more inference engines, future work should also investigate performance differences with a wider range of deep learning models, such as recurrent and convolutional neural networks. Other possibilities would be to extend this comparison to slower CPUs and to test with quantized neural network models.

Chapter 6

Embedded Real-Time Expressive Guitar Technique Recognition

Following the various approaches presented in the previous two chapters, we set to improve the expressive guitar technique classifier presented as a demonstrative example in Chapter 3. In this chapter, we present a flexible-latency approach to embedded real-time expressive guitar technique recognition. We set to classify four pitched and four percussive techniques on seven acoustic guitars. We conducted three experiments to get a broader understanding of the problem and the capabilities of the classifier. We observed how relaxing onset-to-result latency constraints can greatly benefit accuracy with pitched techniques, while more attack-based percussive techniques can suffer from longer feature windows. Additionally, we observed a "GuitarPlayer" effect that deeply influences the reliability of performance metrics and model generalization capabilities. This was successfully addressed with grouped k-fold cross-validation, achieving reliable accuracy results between 60.1% and 81.7% with total latency between 14.2 and 101 ms on a Raspberry PI implementation. We achieved a 17% accuracy increase when focusing on one guitar and player, and measured a non-negligible impact of the guitarist's touch on recognition results.

This chapter discusses a contribution submitted to IEEE/ACM Transactions on Audio, Speech, and Language Processing.

6.1 Introduction

In this chapter, we present a real-time embedded classifier that harnesses timbre nuances to identify pitched and percussive techniques on a smart acoustic guitar. We considered four main pitched playing styles (i.e., sounds produced on the strings) and four different classes of percussive hits (i.e., non-pitched sounds produced by hitting the guitar's body). The recognition system detects note onsets, analyzes a brief excerpt of the signal (i.e., attack phase) that is captured by the guitar's internal transducers, and outputs a prediction of the expressive technique used. Moreover, we conducted three experiments to investigate the impact of target latency and context information on recognition accuracy, generalization performance, and specialization performance on one guitar.

During the process, we observed the existence of a specific *Guitar/player effect*. This effect is similar to the well-known "album effect" in the artist recognition problem [178]. The original album effect relates to the similarity between songs in the same record and is described as the risk of having a model learn to classify the producer traits and mastering chain of a record when instead trying to learn the higher level characteristics of the artist who made it. When a record is present in both the train and test datasets with different songs, learning models can show misleading high-performance metrics while lacking the ability to generalize to new data, since they mostly learn to classify album-specific similarities. We observed a similar effect even within very small analysis windows in expressive guitar technique recognition, where different guitars and players in a dataset can have very specific traits (e.g., transducer qualities, resonating characteristics, preamplifier equalization, playing style), and models that are trained and tested with data from the same guitar/player pair can show extremely high recognition accuracy even on 5 or 10 fold cross-validation, but fail to generalize to new data. While differences in timbre between musicians and instruments are a well-known fact and have even been used for player identification [20], the impact of these differences in the very first milliseconds of played notes has not been assessed yet. In particular, studies like that of Zhao *et al.* [20] perform player identification by analyzing specific features, e.g., vibrato, over long periods.

The remainder of the chapter is organized as follows. In Section 6.2 we describe the experimental setup, including the data, hardware, and software used, along with the three different experiments conducted. In Section 6.3 we present the results of the experiments and discuss them. Finally, we summarize the work and draw our

conclusions in Section 6.4.

6.2 Experimental Setup and Motivation

The aims of this study are twofold: to describe the approaches that achieved the best results for embedded real-time expressive guitar technique recognition, but also to present a broader overview of the problem. The latter derives from acknowledging the limited amount and variety of expressive guitar data available to us (clean, well-separated, labeled, and varied data) and how it can affect the performance of expressive technique recognition. In particular, we created a novel dataset as well as devised three experiments aimed at investigating the following aspects:

- The recognition accuracy with respect to different latency constraints;
- The *Guitar/player effect* and the generalization performance with more or less variety in the training data;
- The specialization performance of a single instrument, and the relevance of the *guitarist's touch.*

The remainder of this section is organized as follows: Section 6.2.1 describes the data used for the experiments. Section 6.2.2 provides information about the hardware choice while Section 6.2.3 describes the software implementation common to the three experiments. Finally, Sections 6.2.4, 6.2.5 and 6.2.6 describe the three experiments.

6.2.1 Data

The dataset used for the experiments was an updated version of the *expressive guitar technique dataset* mentioned in Chapter 3 and [80]. The dataset is a work in progress with recordings of pairs of guitars and players being added rather regularly. The dataset is currently composed of 15 hours and 55 minutes of recordings of over 35,000 individual notes captured with the internal pickups of 7 acoustic guitars. These are played with 12 distinct expressive techniques (including percussive techniques), and three different dynamics (i.e., piano, mezzo forte, forte) by 6 experienced guitarists. The dataset was made freely available at [28]. Of the 12 techniques, 8 attack-based techniques were selected for these experiments, where four are percussive techniques (percussive hits on different parts of the soundboard), and 4 are pitched. The techniques considered for this study are the following:

1. "Kick" technique: hit on the lower right part of the guitar top;
2. "Snare-1" technique: hit on the lower side of the guitar body;
3. "Tom" technique: hit on the upper guitar body near the top of the fretboard end;
4. "Snare-2" technique: hit on the muted strings over the fretboard;
5. "Natural Harmonics": plucking a string and stopping it to let harmonic overtones ring;
6. "Palm Mute": partially muting a string with the palm of the picking hand;
7. "Pick Near Bridge": plucking a string near the saddle, or bridge;
8. "Pick Over the Soundhole": plucking a string over the soundhole.

A more detailed description of the techniques and dataset can be found in Chapter 3 and [80]. It is worth noticing that we consider any note played by a guitarist as being played with one playing technique. In this context, "Pick over the soundhole" can be considered the *ordinario* technique. Some of the other techniques in the dataset (e.g., bending, hammer-on, or vibrato) were left out for this particular study as we opted to perform short-time (i.e., attack-timbre based) technique recognition, while these will be addressed through pitch tracking on longer time scales in future studies. The reasons for this choice are described more in detail at the end of Chapter 3, in Sections 3.6.4 and 3.7.

With respect to the data used for the work inChapter 3, two more guitar/player pairs were added. Furthermore, the slice of the dataset with the eight playing techniques selected here was composed of 31,911 notes and as many onsets. Differently from the previous study, each onset was labeled at the millisecond level in order to precisely measure the onset detection latency, accuracy, and total recognition latency.

Additionally, for *Experiment 3* (see Section 6.2.6), we recorded a small additional test dataset. This *extra-test* dataset is composed of 362 notes recorded by the pair Guitar#**1**/Player-**A** (which is already present in the main dataset), and 380 notes recorded by the new pair Guitar#**1**/Player-**B**, where guitar Guitar#**1** is the same as in the main dataset, but Player-**B** is a different guitarist playing the same guitar. The extra-test dataset is used to measure differences in the recognition accuracy between the two players on the same guitar, which could suggest a non-negligible effect of

the *guitarist's touch* on the recognition system. We consider as guitarist's touch the specific way a guitarist conceives and plays any expressive technique (e.g., the amount of force applied in string muting, hand positioning or pick positioning).

6.2.2 Hardware and embedded implementation

The classifiers trained for these studies were deployed on a Raspberry PI4 (RPI4) 4 Gb single board computer running the Elk Audio OS [24]. The choice of a single-board computer is justified by the compact size of the hardware and reasonable power consumption, along with the rather satisfactory computational power of this most recent model. The input and output audio signals were handled with the Elk-PI Hat development board, which performs AD/DA conversion. However, given the requirements of these experiments, similar results can be reproduced with just a RPI4 Board and an Elk audio OS compatible ADC/DAC board (e.g., HifiBerry DAC+ ADC Pro).

To measure the onset-to-results latency, test data recordings were played on a computer and fed to the board through the output of a USB audio interface (Focusrite Scarlett 2i2). At the same time, the stereo output from the board was recorded through two inputs of the audio interface. The left channel contained a passed-through version of the guitar signal fed to the classifier, while the right contained a set of reference spikes in correspondence with the time at which the classifier produced its prediction for each note. Measuring the interval between each note onset in the left channel and the relative spike in the right channel allowed us to measure the total "software" latency of the system in a more reliable way than previous methods [80]. The measurement excluded any hardware or buffering latencies that can differ based on the specific implementation and DAC / ADC hardware used, as they were not relevant to the study.

6.2.3 Software

The basic intuition behind the classification pipeline used for these experiments is not dissimilar to the work presented in [80]. In particular, expressive guitar technique recognition is performed on a per-note basis, and the classification is composed of three separate steps (i.e., onset detection, feature extraction, and classification), which is preferable to an end-to-end neural approach in the embedded real-time context (See Chapter 3). At the end of the pipeline, the extracted features are fed to

a neural classifier that can predict a higher-level property such as expressive guitar technique starting from timbral features. Onset detection is used to trigger feature extraction, which is delayed to gather a sufficient amount of audio signal. While recent approaches in offline MIR employ end-to-end networks that operate directly on the raw audio signal and learn their own internal features, this can be challenging for a real-time embedded approach that tries to limit computation requirements and latency. On the contrary, timbral features such as MFCC can be extracted in real-time with a reasonable amount of computational power.

With respect to our previous research, numerous improvements were made to the classification pipeline. The main improvements are the following:

- Onset detection was optimized for both accuracy and latency with an evolutionary algorithm, as described in Chapter 4. The onset detection accuracy and F1-score achieved are respectively 90.98% and 95.28%.

- Instead of extracting 1D feature vectors from a single, small signal window, now 2D matrices of features are extracted from overlapping windows. This allows the classifier to receive more information about timbre and its temporal evolution in the very first milliseconds of the note.

- The classifier was changed from a 1D-input fully connected neural network to small 2D convolutional neural networks. These proved to be more accurate and efficient than the previous approach. The source code of a 2D-input wrapper for the TFLite library was made available online [1].

Other than improvements to the software pipeline, the conceptual differences with the previous studies are highlighted in Sections 6.2.4, 6.2.5, and 6.2.6.

Onset detection

For onset detection, we used the Aubio library [128] implementation of the Modified Kullback-Leibler (MKL) distance function with Adaptive Whitening [123] disabled, whose parameters were optimized for both accuracy and latency with an evolutionary algorithm (see [135]). As a result, we measured an accuracy of 90.98% and an F1-score of 95.28% on the main dataset. On the smaller extra test dataset, the accuracy and F1-score of the onset detector were 95.72% and 97.81%, respectively.

[1] https://github.com/CIMIL/cpp-deep-inference-wrappers

Feature extraction

For the feature extraction step, we used a set of timbral extractors from the TimbreID Pure Data (Pd) library [27], which were converted to C++ classes and made compatible with the Juce framework for audio plugins. The extractors used for the classification pipeline are MFCC, BFCC, Bark Spectrum, Bark Spectral Brightness, Real Cepstrum, Peak Sample, Attack Time, and Zero Crossing Rate. For each note, features are extracted throughout the *feature window* interval. The feature window is composed of a number of overlapping sub-windows, each of which is 256 samples long and overlaps by 50% with the neighboring sub-windows. The first sub-window is overlapped by 50% with the signal that comes before the beginning of the feature window, while the last overlaps with zero padding. The number of sub-windows is determined by the overall feature window length. Figure 6.1 shows the case of the 704 samples feature window (11 blocks of 64 samples), which contains 6 sub-windows. For each overlapping sub-window, the aforementioned extractors produce a vector of 271 feature values.

To preserve the characteristics of the feature extraction process, the first part of the classification pipeline (i.e., onset detection and feature extraction) was also compiled as a separate VST plugin and executed on a computer to extract features from notes in the training dataset. This ensures that the feature extraction process is the same between the training and the deployment phases. However, in the extraction phase, the plugin can be executed considerably faster than real-time, to reduce the time needed to process the dataset. The feature extraction plugin was provided with the ability to save the extracted matrices as flat vectors in Comma-separated values (CSV) text format. The source code for both the C++ version of part of the TimbreID library and the feature extraction plugin is available as open source on GitHub [2].

Classifier Training and testing

The neural models used for each experiment were trained and tested using Keras and TensorFlow on a Jupyter Notebook environment. Later, final models were converted to the TFLite format and deployed on the embedded board, where latency was measured and the accuracy of the system was compared with the test accuracy measured on the notebook. This ensured the correctness of the conversion process and cross-validation procedure. In particular, this part of the process allowed us to identify the

[2] https://github.com/CIMIL/cpp-timbreID

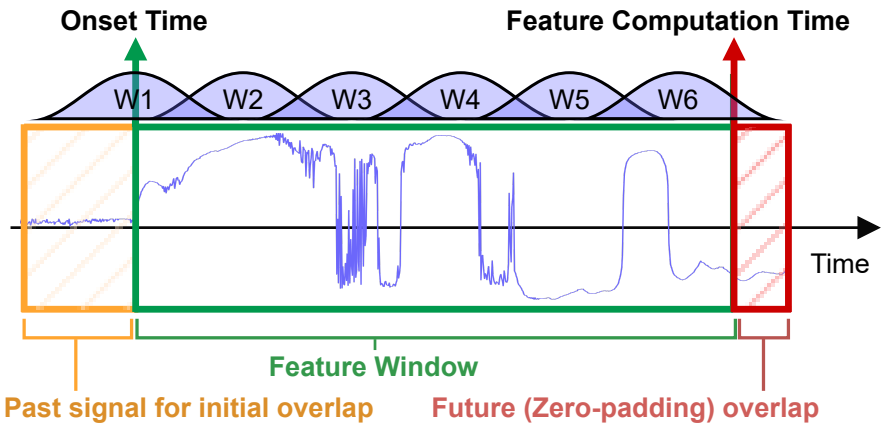


Figure 6.1: *Overlapped feature extraction for the 704-samples feature window. Here six sub-windows of 256 samples each (four audio blocks), with 50% overlap are spaced across the entire feature extraction window (704 samples) plus a part of the pre-onset signal and zero-pad blocks for the part on the right of the feature computation instant. Hann windowing is used for the subwindows.*

Guitar/player effect (see Section 6.1), when models that previously showed over 95% accuracy on regular 5-Fold cross-validation, resulted in poor performances on new data. To find the optimal model for each experiment condition, the Python training notebook was converted to a script, and several hyperparameters were exposed as command-line arguments. The script was used to perform a grid search over a range of values for each hyperparameter, and the best model was selected based on the recognition accuracy. The hyperparameters optimized are the following: learning rate, batch size, training epochs, number of input features for automatic selection, number of convolutional layers, kernel size per layer, stride per layer, number of filters per layer, layer activations, pooling layer type, number of fully-connected layers, number of neurons per Dense (i.e., fully-connected) layer, and dropout rate. Multiple rounds of grid search were performed for each experiment condition, starting from a coarse grid and progressively refining the search space. The code and data used to train the classifiers is available on GitHub as well[3].

[3] https://github.com/CIMIL/ExpressiveGuitar-TechniqueClassifier

6.2.4 Experiment 1: Recognition accuracy with respect to Latency constraints

The target "repurposing" application of a real-time expressive technique recognition system, i.e., the way that the classification results are used, determines the maximum tolerable latency for the user. In turn, this affects how long the system can analyze the audio signal from the onset of a note before having to produce a classification on the currently available context. In a sound-to-sound application, (i.e., where the classification made on the incoming audio controls sound generation or processing) latencies as little as 10 milliseconds can be the maximum tolerable delay between action and sound for players [116]. However, there are application scenarios where recognition results are used for less time-critical control parameters of a sound synthesis engine, while note triggering is performed through quicker pitch trackers and envelope followers [53]. In this case, the latency of the recognition system can be increased to allow for more accurate classification results, without affecting the perceptual quality of the instrument. Other applications can include sound-to-video systems, where the use of different expressive techniques can trigger or affect live visuals, in conjunction with continuous measures such as signal amplitude or pitch. In such applications, video artists might desire a system with a latency that matches the frame rate of the video output, which is typically 30 or 60 frames per second (i.e., 33 or 16 ms). However, this would be a requirement only when using fast-paced video animations or effects. For more relaxed musical styles, musicians and visual artists may steer towards slowly evolving visuals, in which case they may be willing to trade off some latency for better recognition accuracy. This can also apply to applications that control stage equipment over the network, such as lighting, smoke machines, or other stage effects [25].

These requirements indicate that the end-users or the developers of these repurposing applications could benefit from real-time recognition systems that offer a flexible choice of the tradeoff between how accurate and how reactive the system is. In light of this, we optimized and trained four classifiers with respectively four target latencies of 15, 45, 75, and 100 milliseconds. We observed how 15 milliseconds is the minimum latency that can provide reasonable accuracy with our approaches and data, while 100 milliseconds was a reasonably large delay that we consider to be greater than the maximum tolerable latency for most musical tasks, even under the most relaxed constraints. The target latencies of 45 and 75 milliseconds were chosen

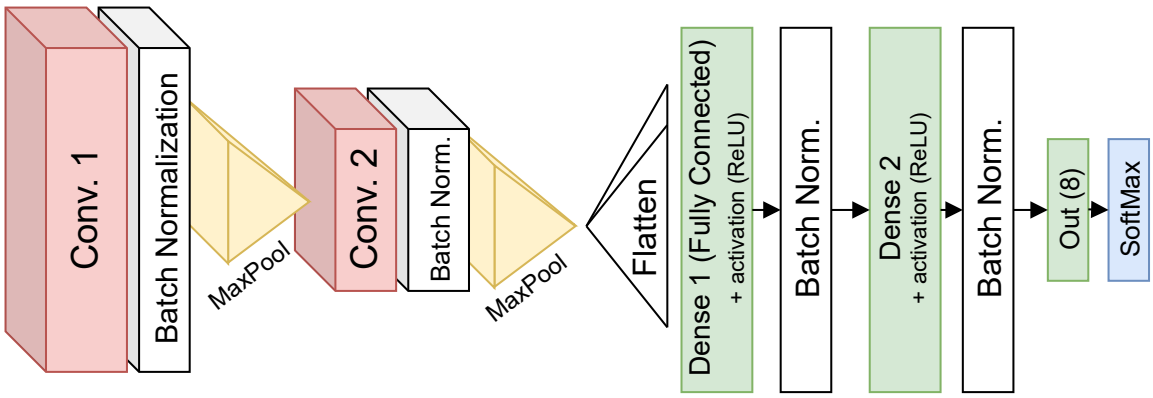


Figure 6.2: *General structure of the classification networks. First, a set number of 2D convolutional layers process the input feature matrix, and they are interleaved with batch normalization and pooling layers. Then the data is flattened and passed through a set number of fully connected layers, which are also interleaved with batch normalization, and dropout is used for further regularization. Finally, the last layer applies the softmax function to the output. The numbers and types of hidden layers are part of the hyperparameters that were found via grid search (see Table 6.1). For example, only one parametrization out of the four latency configurations included two convolutional layers.*

by dividing the range between the two extremes and rounding the result. For each condition, the main effect of a different target latency is on the length of the feature extraction window. The longer the window, the more information is available to the classifier, but this also increases the latency. In this case, for the four conditions, the window length was set to 704, 2112, 3456, and 4800 samples respectively, with a sample rate of 48kHz. Each value is an integer multiple of the audio block size used (i.e., 64 samples). With the windowing procedure described in Section 6.2.3, the four experiment settings correspond to extracted feature matrices of sizes (271×6), (271×17), (271×28) and (271×38) for each note respectively.

For each condition, the classifier was optimized through grid search resulting in different CNN architectures. The best-performing model for each condition is shown in Table 6.1. Additionally, Figure 6.2 shows the model structure. The accuracy was evaluated through Guitar/Player cross-validation (see Section 6.2.5), where the data was split into 7 folds corresponding to data from the 7 guitar/player pairs in the dataset. For each fold, the model was trained on the other 6 folds and tested on the selected data, so that the test guitar/player pair was never used for training. Resulting metrics are averaged over the 7 folds.

Once the best model for each configuration was found, these were converted to

Table 6.1: *Optimal values found for the hyperparameters of each classifier configuration for Experiment 1 and the resulting number of weights.*

	Feature Window Size			
Parameter	**704**	**2112**	**3456**	**4800**
Learning-rate	8.0e-05	1.0e-04	1.0e-05	1.0e-04
Batch-size	64	64	128	128
Training Epochs	500	600	600	300
Features (per subwindow)	271	200*	200*	200*
Tot features	*1626*	*3400*	*5600*	*7600*
Num. of Conv layers	2	1	1	1
Conv. kernel sizes	3x3,3x3	5x5	5x5	5x5
Conv. strides	1,1	2	2	2
Conv. Num. filters	4,4	32	64	32
Conv. Activations	relu,relu	relu	relu	relu
Pooling layer types	Max,Max	Avg	Avg	Max
Num. of Dense layers	2	None	None	None
Width of Dense layers	32	16	16	16
Dropout rate	0.5	0.5	0.5	0.5
Model weights	10,404	52,168	181,128	116,168

* The number of features *computed* for each subwindow is 271 and is determined by the size of the subwindows and filter spacing for different extractors. However, feature selection was found to be successful, via hyperparameter optimization, for the three configurations **2112**, **3456**, and **4800**. In particular, a value of 200 features performed the best.

the TFLite format and deployed on the embedded board. The latency of the whole pipeline was measured by feeding guitar recordings to the input of the system while recording the output of the board. The output is composed of a copy of the clean input signal, and a reference signal, which allowed us to accurately measure the total latency (see Section 6.2.2). Furthermore, software probes that employed a steady clock allowed us to break down the total delay between the note onset and the classification results into its main components, namely the onset detection latency (τ_{OD}), the post-onset delay (τ_{POD}) introduced to properly delay feature computation, the feature extraction or computation latency (τ_{FE}) and the classification latency (τ_{DNN}). A graphical representation of the time instants and corresponding intervals is shown in Figure 6.3.

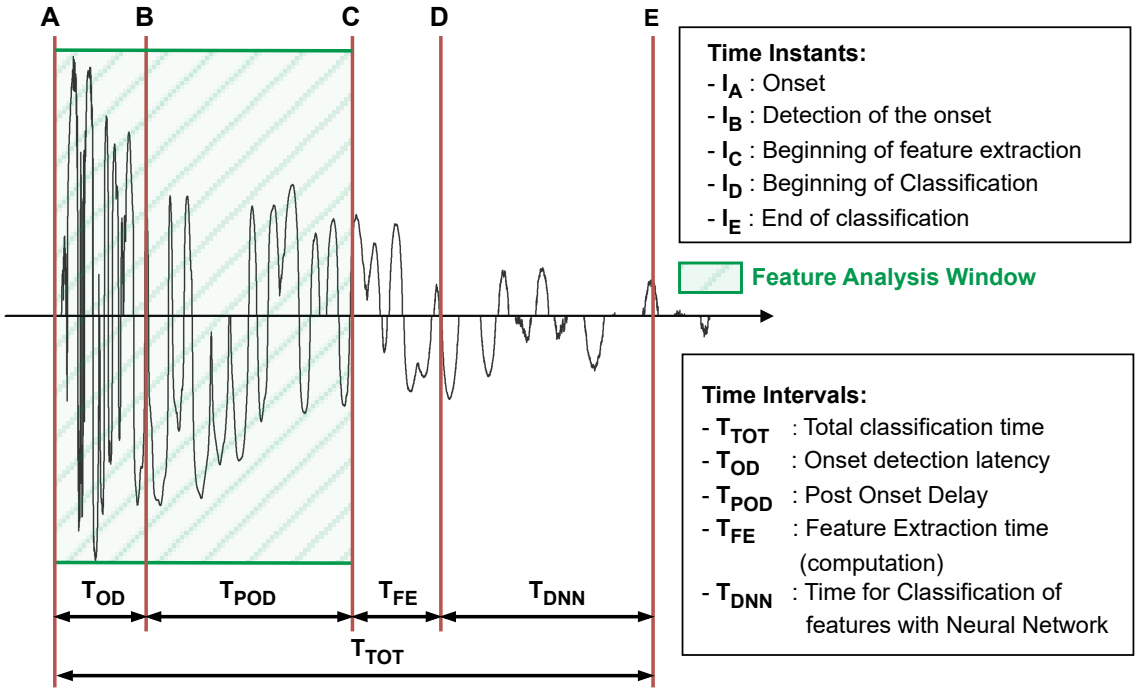


Figure 6.3: *Representation of the time intervals that compose the total latency of the system. The total onset-to-results latency corresponds to the sum of the Onset detection latency, the post-onset delay used to align the feature window, the feature computation latency, and finally the classifier inference time.*

6.2.5 Experiment 2: Guitar/Player effect and generalization performance

This experiment focuses on the generalization performance of the system with respect to the amount of data available. For this, we started from one of the configurations of Experiment 1 (i.e., the lowest latency, 704 samples configuration) and measured the performance of the system with both regular *stratified 5-fold cross-validation* and a different *Guitar/Player cross-validation*. Moreover, we trained and tested the classifier starting from a single guitar/player pair and progressively adding the remaining pairs one at a time. This was done to assess whether the amount of data available to us is a limiting factor for the performance of the system and to roughly estimate the amount of data required to achieve a certain level of performance.

For the regular *stratified k-fold cross-validation* measurement, the sci-kit learn `StratifiedKFold` utility was used to create 5 folds where the percentage of samples from each class was preserved from the original dataset. Instead, for our *Guitar/Player cross-validation*, the data is split into 7 folds corresponding to data from the 7 guitar/player pairs in the dataset. In this sense, Guitar/Player cross-validation is a special case of Group K-Fold cross-validation, where each fold contains exactly one guitar/player pair. This way, the model is not only tested on data that was not seen during the training, but also that was played by different a guitar/player pair than those in the training set.

Regular 5-fold cross-validation was repeated starting from a single guitar/player pair and up to all 7 pairs. For the Guitar/Player cross-validation instead, the minimum number of guitarists in the dataset is two, to always have at least one pair for testing only.

We choose to use the quickest and lowest scoring configuration because Experiment 2 requires a large number of measurements and training sessions, and this configuration could be trained on a single Laptop GPU (i.e., Nvidia GeForce GTX 1650 Ti) with low energy consumption and carbon footprint. On the contrary, the two configurations with the largest feature windows for Experiment 1 needed to be trained with a more powerful GPU (i.e., Nvidia GeForce RTX 3090) because of the larger memory requirement.

6.2.6 Experiment 3: Specialization performance and the "Guitarist's Touch"

Once we assessed the generalization capabilities of the system, we set to measure its performance on a specific single instrument. It is not unreasonable in fact, for an embedded recognition system for an SMI to target a specific instrument rather than focusing on performance across different instruments. In this context, we set to verify whether adding data from different guitar/player pairs would affect recognition accuracy. This can be a useful result because, in the case that the data from a single instrument is limited, it can inform whether it is possible to integrate data from different conditions, or if it is necessary to collect more data. In order to do this we recorded extra data (see Section 6.2.1) for Guitar#**1** and a player that we will refer to as Player-**A** from now on. The pair Guitar#**1**/Player-**A** was already present in the main dataset. The final model for the chosen configuration (i.e., 704 samples feature window) was first trained on the main dataset recordings for only the pair Guitar#**1**/Player-**A**, and tested on the extra data for the same pair. Then, the model was trained and tested six more times by progressively adding the six remaining pairs to the training set only.

Additionally, we set to evaluate the performance of these trained models on the same instrument (i.e., Guitar#**1**) but with a different player (i.e., Player-**B**) that was not present in any recording of the main dataset. Although limited in its extent, this experiment can help to understand whether the performance of a real-time expressive guitar technique recognition system can be affected by a certain degree of "guitarist's Touch". Guitarist's touch is a term used to describe the subtle differences in the way different musicians play the instrument [179]. While the existence of the guitarist's touch is undoubted, it is yet to be verified whether the differences in the way a musician intends and plays certain expressive techniques can affect the sound of the instrument in the very first milliseconds of played notes. This can help to inform a future in-depth study on the matter that could, in turn, drive relevant choices in how to integrate new recordings into our dataset, or the creation of new datasets that are specifically targeted to a single instrument.

6.3 Results and Discussion

6.3.1 Experiment 1: Recognition accuracy with respect to Latency constraints

The results of Experiment 1 are shown in Figures 6.4 and 6.5. The first plot shows the recognition accuracy across the eight techniques and the latency of the different configurations of the system, while the second shows the F1-score for each technique. Accuracy and F1-score values reported are averaged over 7-fold Guitar/Player cross-validation (see Section 6.2.5). The total latency is broken down into its components, as described in Figure 6.3.

The accuracy results show a clear increasing trend with the increase of the feature window (therefore latency). However, The greatest increase in accuracy is observed between the 704 and 2112 window configurations, with marginal improvements with longer windows. Figure 6.5 offers better insight into why this is the case. In particular, we see how the performance for all techniques increases between the first two configurations, indicating that 704 samples at 48kHz may be a very limited context to capture most of the expressive techniques, but the most sizable performance jumps are seen with pitched techniques and natural harmonics in particular. This can be attributed to the more complex nature of pitched techniques and the pitch variations in the data.

Moreover, natural harmonics are played by plucking the muted string and immediately releasing the finger, which is a fast action that often is detected as two onsets, making it difficult to align the feature window. While the first action is always detected as the main onset, and the second suppressed by a debouncing mechanism, the more interesting timbre content happens only after the second onset. This can explain how longer windows, which include this additional context, can largely improve the performance with natural harmonics. A more in-depth analysis of the Natural Harmonic false negative notes shows that these are generally confused with other pitched techniques, and to a lesser degree with the Snare-1 technique. This seems to change depending on the guitar/player pairs.

With longer feature windows, the performance of pitched techniques tends to increase less and stagnates in some cases, but the performance of percussive techniques tends to decrease. This can be due to percussive techniques generating sounds with

shorter decay and more information around the early attack phase of the signal. While adding more context should not affect the performance of percussive techniques, trying to classify pitched sounds may cause the network to favor focusing on the whole feature window instead of the sole attack.

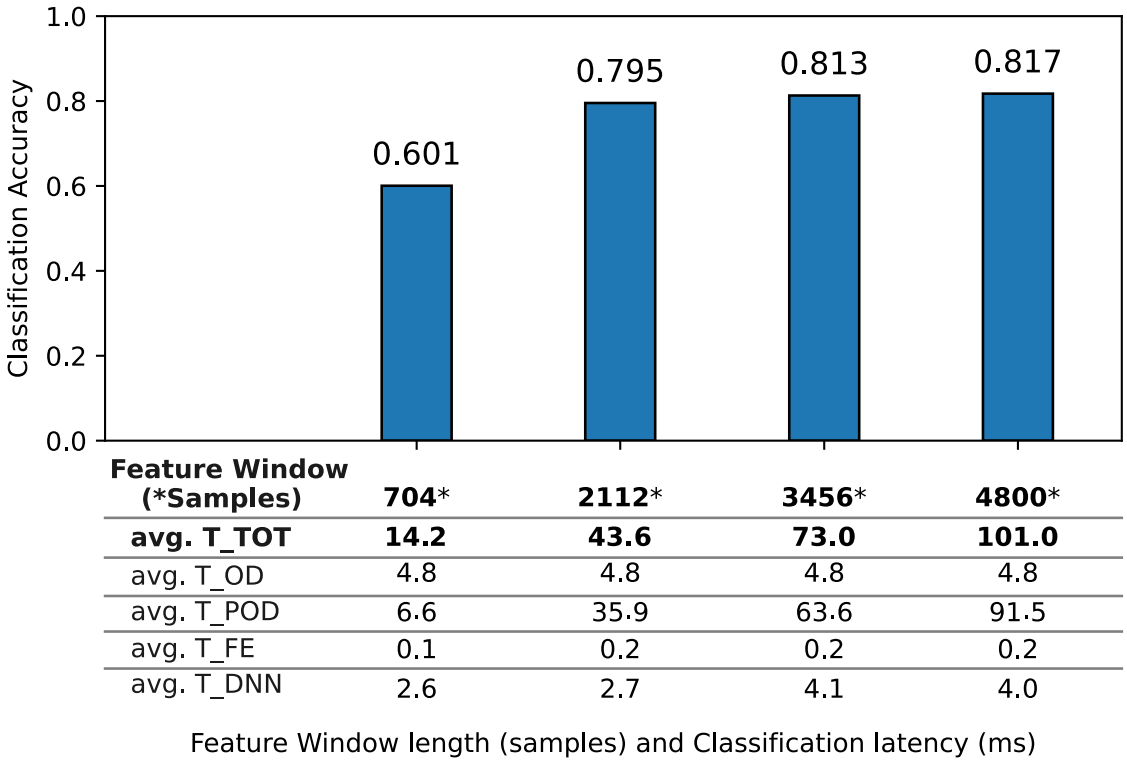
In terms of the latency results measured on the embedded implementation, the impact of the classifier inference (T_{DNN}) is less relevant with greater feature windows. Moreover, most of the latency in the configurations with 2112, 3456, and 4800 is constituted by the post-onset delay (T_{POD}), which is used to align the feature windows with the onset and depends on both its size and T_{OD}. The onset detection delay (T_{OD}) remains the same as the configuration of the detector was not changed, but the composition of the total latency shows how we could have different onset detector settings with more latency and greater accuracy and reduce T_{POD} without affecting the total latency. Finally, the time required by the actual computation of the feature matrices (T_{FE}) is negligible, as most of the computations are simple and performed when each audio block of 64 samples is added to the buffer.

6.3.2 Experiment 2: Generalization performance and the "Guitarist/Player effect"

The results of Experiment 2 are shown in Figure 6.6. The plot shows the recognition accuracy of the 704 sample feature window configuration, trained and tested on a progressively larger dataset, where guitar/player pairs are added one at a time. Striped bars represent the average results obtained with regular Stratified 5-fold cross-validation, where all the data from the selected guitar/player pairs is mixed and each sample is eligible to become part of any of the 5-folds. Conversely, blue bars represent the results of Guitar/Player cross-validation, where data from each guitar/player pair constitute a separate fold. As a consequence, the rightmost blue bar corresponds to the first bar of Figure 6.4, while the remaining bars represent the performance of the model trained on fewer data. The takeaways from the results of Experiment 2 are the following:

1. Generalization performance increases with the addition of data from more guitars and players, but the current size of the dataset is limiting;

2. The performance averaged over regular 5-fold cross-validation is misleading as it decreases with the increase of the real generalization capabilities of the classifier.



Feature Window (*Samples)	704*	2112*	3456*	4800*
avg. T_TOT	**14.2**	**43.6**	**73.0**	**101.0**
avg. T_OD	4.8	4.8	4.8	4.8
avg. T_POD	6.6	35.9	63.6	91.5
avg. T_FE	0.1	0.2	0.2	0.2
avg. T_DNN	2.6	2.7	4.1	4.0

Figure 6.4: *Recognition accuracy and latency of the four configurations for Experiment 1. For each configuration, a different feature extraction window length is used, impacting both the accuracy and latency of the system. The total latency is reported along with a breakdown of its different components, as described in Figure 6.3.*

Takeaway 1 highlights an expected increase in the performance of the model, but it indicates that the task of generalization over multiple guitars and players is complex, and having only seven guitar/player pairs in the dataset is limiting the potential recognition performance. Furthermore, the unstable increase in performance might suggest that, for this task, a satisfactory number of guitar/player pairs would be considerably higher than seven. These results suggest how future efforts should either be directed at increasing the size of the dataset, or at investigating the possibilities of specializing the trained models on a single instrument with techniques such as layer-freezing and fine-tuning, or domain adaptation. While doing network fine-tuning with a whole new set of data from the target instrument would defeat the purpose of a generalization experiment, using a small and possibly unlabeled set recorded on the fly would be a reasonable way to adapt the pre-trained model in the real-world scenario.

Takeaway 2 highlights how using a cross-validation procedure that is not grouped

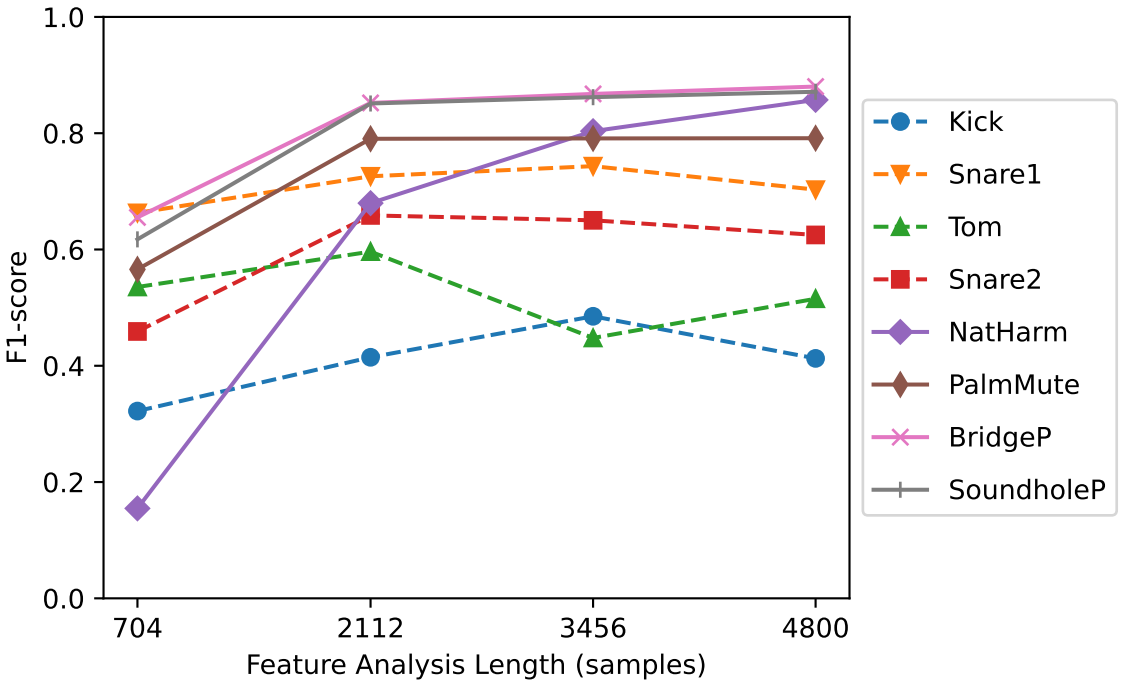


Figure 6.5: *F1-score for each expressive technique and each configuration of Experiment 1. Dotted lines represent percussive techniques, while solid lines represent pitched techniques (see Section 6.2.1).*

by guitar/player pair can be misleading, as it shows a high recognition accuracy when the model overfits the specific characteristics of the guitars and players in the dataset. This becomes detrimental to the real generalization performance of the classifier when recognition accuracy measured with non-grouped cross-validation is used to drive hyperparameter optimization and model selection. In this case, even on the smallest analysis window, we observed the insurgence of a *Guitar/player effect* that fosters the overfitting of the model to the specific characteristics of the known guitars and players rather than learning the general properties of the expressive techniques.

6.3.3 Experiment 3: Specialization performance and the "Guitarist's Touch"

The results of Experiment 3 are shown in Figure 6.7. The plot shows the recognition accuracy of the 704 sample feature window configuration, trained starting from a specific guitar/player pair (i.e., Guitar#**1**/Player-**A**) and tested on extra data from the same instrument but two players separately (i.e., Player-**A** and Player-**B**). Furthermore, the classifier is progressively trained on data from the other guitar/player

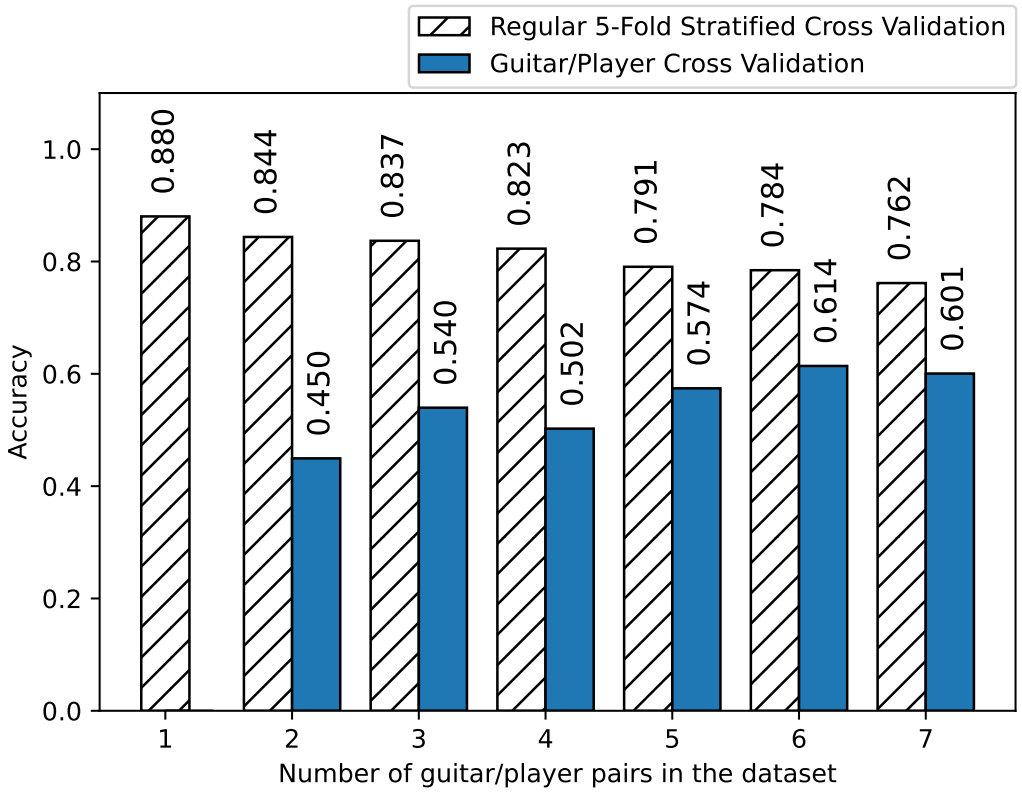


Figure 6.6: *Average recognition accuracy of the 704-samples configuration of the classifier depending on the number of guitar/player pairs in the dataset used to train and test the model. Striped bars represent the accuracy for 5-fold cross-validation, while blue bars represent Guitar/Player group cross-validation. Guitar/Player cross-validation is not performed for one pair since it requires at least two pairs, which are kept separate between training and test.*

pairs in the main dataset. While Player-**A** is always in the training set for each model, Player-**B** did not record any data for the main dataset. The main results of Experiment 3 are the following:

1. The model trained only on data from Guitar#**1**/Player-**A** performs considerably better on extra data from the same pair than on new guitar/player pairs (see Experiment 1 results in Section 6.3.1);

2. Adding data from other guitar/player pairs to the training set is detrimental to the performance of the model on the specific Guitar#**1**/Player-**A** pair. This indicates that specific guitar/player data is preferable over data quantity;

3. Most importantly, the model trained on Guitar#**1**/Player-**A** performs consistently worse when a new (unknown) Player-**B** plays the same instrument. This

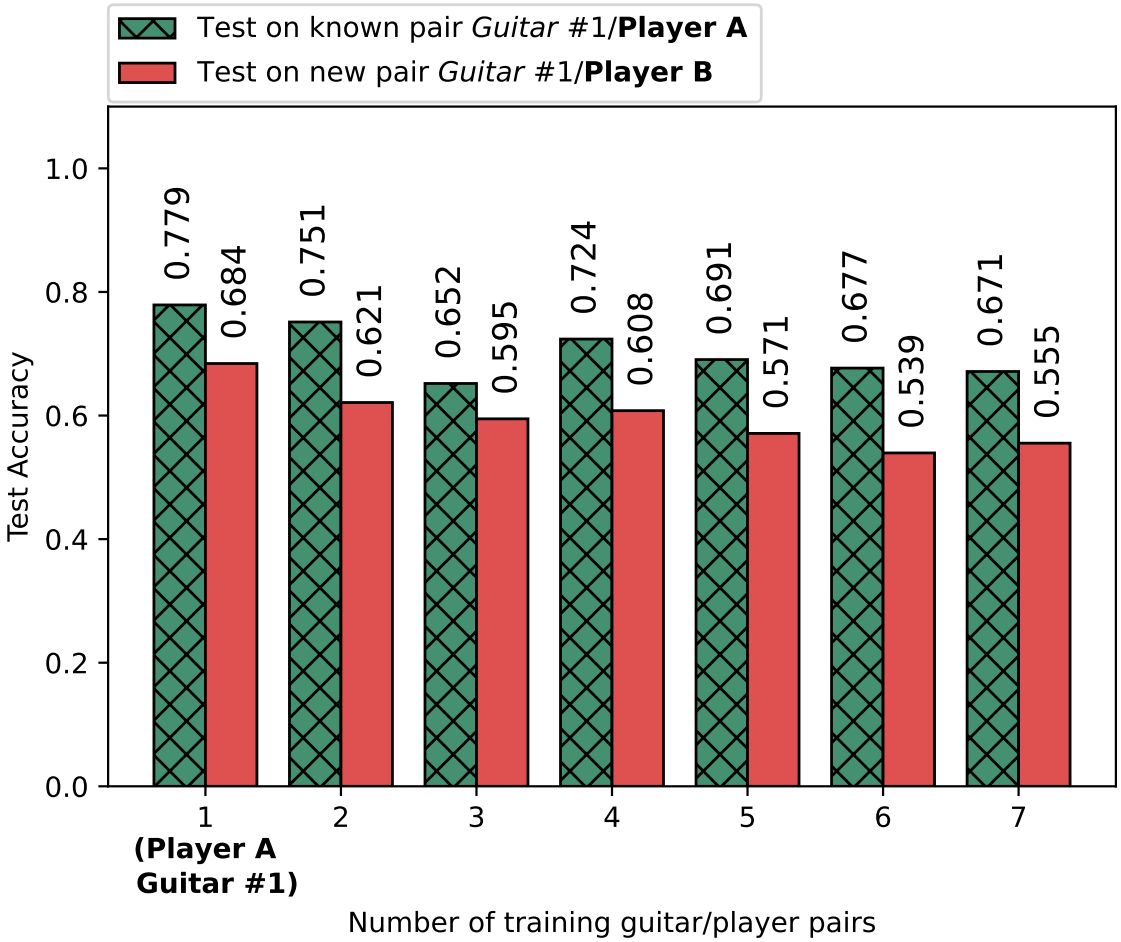


Figure 6.7: *Test accuracy for Guitar#**1** with respect to the number of guitars in the training set. Green bars represent the accuracy of the system on the extra data recorded by the known **Player-A**, while red bars indicate the performance for the new **Player-B**.*

suggests that the "Guitarist's Touch" could be a relevant factor for the performance of an expressive guitar technique classifier.

The first result is to be expected, but it is still useful to verify that the classifier is not overfitting on the specific data samples from the main set, since the extra test set was recorded months later, with a different pick, cables, and slightly different guitar settings (e.g., neck bow, volume setting). Independence on these factors is a key characteristic of a proper playing technique classifier. Additionally, this can highlight how the generalization metrics of Experiment 1 are not relevant when designing a classifier for a specific instrument and player. Takeaway 2 instead highlights how adding training data from new guitar/player pairs cannot help the specialization

performance for a target pair. Finally, Takeaway 3 suggests how differences between the guitars used are not the only factor that can hinder recognition performances. In particular, the ways that different musicians play the same techniques differently (i.e., the "Guitarist's Touch") seem to affect the recognition performance of a real-time expressive guitar technique classifier, even with the very short context considered. While the limited extra data used for Experiment 3 does not allow for a conclusive analysis of the Guitarist's Touch impact on the task, the consistency of the results suggests that this factor is relevant and should be investigated in-depth in future work.

6.4 Summary

In this chapter, we presented a flexible-latency embedded real-time expressive guitar technique classifier. Moreover, we investigated the impact of the task requirements and data characteristics on recognition performance. We found that relaxing the latency constraints, especially between 15 and 45 milliseconds, can benefit the recognition accuracy for pitched and percussive techniques, while the performance for percussive techniques is mostly unaffected and even slightly degraded with larger feature windows. Additionally, we observed a tendency for models to overfit specific guitar and player characteristics even on very small signal windows, limiting their ability to learn the broader properties of expressive techniques and the generalization performance on new instruments and players. We successfully addressed the issue by employing grouped k-fold cross-validation to ensure that guitar and player pairs remain separate during training and testing, which provides a reliable accuracy metric to drive hyperparameter optimization. On the contrary, the accuracy metrics obtained with simple k-fold cross-validation were revealed to be misleading as they increased with the decrease in real generalization performance. Finally, we focused on a single instrument and found how this can expectedly lead to better performance, but also how the different touch, or style, of different guitarists, is likely to affect the recognition performance. The classifier was designed for low latency on resource-constrained embedded devices, and it was implemented and successfully deployed to a single-board computer. The limitations of this study are in the limited size of the additional dataset used for the experiment on the guitarist's touch, and the focus on the sole recognition of attack-based techniques on monophonic signals. Polyphony can be addressed with the use of a special hexaphonic pickup. Furthermore, the

embedded real-time recognition of more techniques could be enabled through the integration of more features on multiple time scales. Finally, observing the effect of the "Guitarist's Touch" on recognition is going to inform a more thorough investigation and the integration of new data in the expressive guitar technique dataset.

Chapter 7

Real-Time Embedded Deep Learning on Elk Audio OS

As mentioned in previous chapters, notable progress has been made, in recent times, in deep learning architectures tailored for music. This coincides with the recent availability of increasingly powerful embedded computing platforms designed for low-latency audio processing tasks. These advancements have opened promising avenues for the creation of innovative Smart Musical Instruments (SMIs) and audio devices that harness deep learning models executed on compact embedded computers. Despite these promising prospects, we found a lack of instructions on how to effectively deploy neural networks to many promising embedded audio platforms, including the real-time Elk Audio operating system. Within this chapter, we define and present a procedure for deploying deep learning models on embedded systems leveraging the Elk Audio OS. This procedure encompasses the entire deployment process, from establishing a compatible code project to its execution and diagnostic evaluation on a Raspberry Pi. Additionally, we discuss various approaches for executing real-time deep learning inference on embedded devices and offer viable options for managing larger neural network models. To streamline implementation and facilitate future updates, we provide an online repository featuring an elaborate guide, code templates, functional examples, and precompiled library binaries optimized for the TensorFlow Lite and ONNX Runtime inference engines (IEs). This work aims to bridge the

existing gap between the development of deep learning models for audio and their practical deployment on embedded systems, thereby promoting the development of self-contained digital musical instruments and audio devices with real-time deep learning capabilities.

This chapter describes the contribution presented at the 4th International Symposium on the Internet of Sounds [180].

7.1 Introduction

In recent years, notable advancements have been observed in deep learning architectures for audio processing [181, 182] and low-latency embedded computing platforms [23, 24, 126, 175]. Deep Learning has proved to be successful in modeling audio effects [183], manipulating tone and timbre in new ways [182], and recognizing in real-time high-level attributes of sound sources, such as for expressive playing techniques recognition [80] or beat tracking [110].

Moreover, the growth in compute capabilities of embedded computers has spurred the creation of various embedded audio platforms, including Elk Audio OS [24], Bela [23], Prynth [101], Satellite CCRMA [102], and Axoloti[1]. Nevertheless, developing deep learning models for audio and deploying them on embedded platforms require distinct skill sets with limited overlap. Specifically, deep learning requires the following skills:

- Proficiency in high-level programming languages;
- Advanced understanding of the mathematical and probabilistic principles underlying layers, activations, and various operators;
- In-depth domain knowledge concerning data and preprocessing, encompassing relevant software and libraries for preprocessing tasks.

Conversely, deploying a model necessitates the following skills:

- Advanced knowledge of lower-level programming languages (typically `C++` and `C`);
- Advanced understanding of compilation and cross-compilation procedures;

[1] http://www.axoloti.com/, https://github.com/axoloti/axoloti

- Familiarity with audio processing and feature extraction libraries specific to the programming language used. Alternatively, an advanced understanding of DSP concepts and programming to effectively implement the required processing routines.

In this context, we believe that any effort aimed at narrowing the gap between the development and deployment phases in audio and music deep learning can foster the creation of novel self-contained DMIs, such as SMIs [25] and other AI-equipped audio devices. The emphasis on embedded platforms stems from the necessity to provide SMIs with artificial intelligence capabilities through computing devices that can be placed inside these instruments. SMIs represent an emerging category of musical instruments that are a central component of the IoMusT, which is an extension of the Internet of Things (IoT) paradigm to the musical domain [184]. As IoMusT devices, SMIs are envisioned to be capable of communicating and becoming part of a "network of interoperable devices" in order to share and receive musical content.

The capacity to conduct deep learning inference within SMIs constitutes a form of "embedded intelligence" [25]. This embedded intelligence can be leveraged for real-time audio processing, manipulation of sensor data, and extraction of high-level audio properties or features, which in turn can be shared with similar instruments within a IoMusT network [185]. In this interconnected scenario, embedded inference holds particular significance since numerous real-time music applications cannot tolerate the inherent latency introduced by cloud-based deep learning solutions.

Among the mentioned embedded platforms, Elk Audio OS [24] and Bela [23] stand out as the most prominent and versatile systems within the open-source domain. Notably, recent efforts were directed at documenting a pipeline for deploying neural networks specifically for Bela [186]. In contrast, while Elk Audio OS on the Raspberry Pi 4 has proved to be a very capable platform for real-time deep learning [80, 109, 187], a documented procedure for deploying deep learning models on this platform is currently lacking. This absence of documentation poses an obstacle to the development of self-contained DMIs with intelligent functionalities.

In this chapter, we describe the steps required to deploy deep learning models to the Elk Audio OS, specifically targeting a Raspberry Pi. Additionally, we provide an online repository[2] encompassing a comprehensive guide, code templates, and precompiled dependencies to facilitate the execution of deep learning inference with either TensorFlow Lite or the ONNX Runtime IEs. These engines were selected based on

[2] https://github.com/CIMIL/elk-audio-AI-tutorial

the comparison described in the previous Chapter. Furthermore, TensorFlow models can be seamlessly converted to the *Lite* format, and models trained in alternative frameworks such as PyTorch can be converted to the *ONNX* format.

We propose a procedure that uses the open-source framework JUCE to develop VST plugins that can be executed on a Raspberry Pi 4 with the Elk Audio OS. This enables developers and deep learning engineers to run real-time and offline audio models on a SBC that can be embedded into instruments and standalone devices. Moreover, a byproduct of learning the proposed procedure is that the reader will also have the tools to compile deep-learning-equipped VST plugins for desktop and laptop computers. This chapter describes a contribution presented at the 4th International Symposium on the Internet of Sounds [180].

The remainder of this chapter is organized as follows. Section 7.2 discusses the background and presents different works related to deep learning and embedded platforms for audio. In Section 7.3 we discuss the tools required to follow the guide, and the motivation behind these choices. Then, Section 7.4 presents an overview of the deployment procedure to follow in order to create a JUCE project, cross-compile plugins for Elk Audio OS, install the OS, configure its DAW and troubleshoot code issues with real-time execution. Furthermore, Section 7.5 contains a few considerations on the different modes of inference, i.e., offline, audio-rate real-time, and other real-time approaches. Then, in Section 7.6 we present some additional applications of this guide, and more broadly this thesis' work. In particular, we focus on two works in progress, started during a period of research at the Centre for Digital Music, Queen Mary University of London. Finally, we summarize the work and draw our conclusions in Section 7.7.

7.2 Background

The deployment of deep learning models onto embedded devices has gained considerable traction and relevance in recent times. This surge can be attributed to a convergence of factors, including the increase in computational capabilities of embedded devices and SBCs, alongside noteworthy research advancements in Artificial Intelligence (AI) for music [181], with a notable focus on real-time approaches [182]. The increase in the availability of powerful embedded computers has fostered the development of various open-source audio platforms, such as Elk Audio OS [24], Bela [23], Prynth [101], and Satellite CCRMA [102]. Meneses *et al.* [175] conducted a thor-

ough comparison of some of these aforementioned open-source platforms, including Prynth, Bela, and a customized processing unit, offering a comprehensive overview of their respective strengths and limitations, without a clear standout winner. More recently, Elk Audio OS was introduced along with an extensive comparative analysis against similar platforms [24]. Furthermore, Vignati *et al.* [126] conducted a performance comparison between the Xenomai Cobalt kernel (utilized by both Elk Audio OS and Bela) and the more common Preempt RT kernel patch for Linux systems. The results demonstrated superior overall performance of the former, especially under heavy computational loads.

While most of the aforementioned platforms were not originally devised to perform deep learning inference, recent efforts have successfully shown how integration is possible and can be made more accessible for deep learning developers [186]. Moreover, the increased interest in embedding deep learning inference for audio into devices and musical instruments has led to the creation of scientific workshops and events that specifically focused on *embedded AI*. A notable example is the NIME 2022 workshop *Embedded AI for NIME: Challenges and Opportunities* [26].

This work takes inspiration from recent research by Pelinski *et al.* [186], who devised a deployment pipeline for deep learning on the Bela platform [23]. The pipeline presented by the authors encompasses a tool for recording sensor reading datasets and a cross-compilation environment, streamlining the deployment of deep learning models onto Bela. The authors articulated their goal to facilitate rapid prototyping and experimentation with neural networks for real-time embedded musical applications through this approach. Their proposed pipeline utilizes the TensorFlow Lite inference engine, and they provide a Docker container for cross-compiling Bela-compatible deep learning programs from a host computer. Similar efforts in this direction have been made by other researchers, as seen in projects like Flucoma-Bela[3]. However, the work by Pelinski *et al.* bears stronger relevance to our effort due to its clear deployment process, catering to individuals without in-depth knowledge of low-level programming concepts, and offering a prepackaged cross-compilation tool to streamline the deployment process.

The main distinction of our work lies in its emphasis on a significantly different platform: the real-time Elk Audio OS and the Raspberry Pi 4. While the combination of Elk Audio OS and Raspberry Pi 4 has been previously utilized for deep learning deployment [80, 109, 187], the intricate deployment process has not been formally

[3] https://github.com/jarmitage/flucoma-bela

documented. Furthermore, the Raspberry Pi 4 offers superior capabilities compared to Bela's hardware, enabling the execution of more demanding tasks, including AI-based audio processing. Hence, we present code examples that perform inference at the audio rate on the input signal (refer to Section 7.3.4), instead of the sensor data processing example provided by Pelinski *et al.* .

In addition to this, we provided comprehensive templates and code examples for both TensorFlow Lite and ONNX Runtime, aiming to streamline the deployment process for a broader spectrum of deep learning frameworks.

7.3 Tools

7.3.1 JUCE and VST

JUCE is a cross-platform framework for audio plugins and applications. It is a `C++` framework with a dual license (i.e., GPLv3 open-source and commercial). JUCE embeds the VST3 SDK and Elk Audio provides support and instructions on how to build a VST plugin for their OS starting from a JUCE project. The procedure reported in this contribution, and in more detail in the project repository, was tested with JUCE 6 (version 6.0.7).

7.3.2 Elk Audio OS

Elk Audio OS [24] is an embedded operating system designed for low-latency audio processing on embedded hardware. Elk provides an open-source distribution for the Raspberry Pi 4 SBC and a cross-compilation SDK. Elk offers support for additional hardware platforms under a commercial license[4]. Moreover, older versions of Elk Audio OS support the Raspberry Pi 3 (i.e., up to version 0.7.2 of Elk Audio OS). The guidelines presented in this contribution refer to version 0.11.0 of Elk Audio OS. Any variations in the deployment procedure for forthcoming versions of Elk Audio OS will be documented in the project's repository (refer to Section 7.3.4). In particular, after the creation of this guide, version 1.0 of Elk Audio OS was released. Release 1.0 simplified some of the steps, which will be addressed throughout this Chapter.

[4] https://elk-audio.github.io/elk-docs/html/intro/supported_hw.html

7.3.3 Choice of Inference Engine

Inference of deep learning models involves the procedure of providing input data to the network and performing the necessary computations to generate an output prediction. Typically, deep learning models are trained and validated on robust server machines or PCs, utilizing high-level programming languages (e.g., Python) and deep learning frameworks (e.g., PyTorch, TensorFlow). The training process demands significant computational resources, often necessitating specialized acceleration hardware and drivers (e.g., GPUs or TPUs). Conversely, inference is notably less computationally intensive and can be optimized for deployment.

In recent years, companies and developers specializing in deep learning frameworks have directed their attention towards in-device inference for edge and mobile computing. Within the realm of IoT, edge computing offers a significant advantage by processing essential computations closer to the point of data acquisition [188].

Furthermore, embedded in-device inference holds a distinct advantage in terms of action-to-reaction latency. This is achieved by conducting inference computations directly at the location where input data is gathered through sensors, eliminating the latency associated with communication to one or more cloud servers. While in-device computation may yield marginal advantages in certain IoT applications, it is an essential requirement for music performance tools and numerous Internet of Musical Things (IoMusT) systems [116]. This holds true even when the learning task allows for slightly more flexible time constraints compared to audio-rate deadlines [80] (see Section 7.5). For this reason, many deep learning frameworks have made available `C` and `C++` libraries known as inference engines. These engines facilitate efficient and quick inference, particularly catering to resource-constrained embedded devices.

In Chapter 5 ([109]) we compared the performance and suitability of four of these engines engines (TensorFlow Lite, ONNX Runtime, Torch+Torchscript, and RT Neural) for audio deep learning tasks on a Raspberry Pi 4 running Elk Audio OS. While the exact execution time can depend on a specific model and task, ONNX Runtime and TensorFlow Lite were found to be quick, well-documented, and easy to use. For the code templates in the repository that accompanies this contribution, both TensorFlow Lite and ONNX Runtime were included separately. TensorFlow users will find it extremely easy to export their model for the former, while models from most frameworks can also be converted to ONNX, including PyTorch [109, 183, 189, 190]. Support for ONNX Runtime is particularly relevant as a large part of research on black-box audio effect emulation is currently carried out using PyTorch, and PyTorch-

to-TensorFlow model-conversion is not a straightforward process [109].

For TensorFlow developers, we suggest using the TensorFlow Lite template code, while PyTorch and other developers should convert their models to ONNX and use ONNX Runtime.

7.3.4 Project Repository

This contribution defines a procedure to successfully deploy deep learning models to embedded devices running Elk Audio OS and perform inference. While this chapter reports an overview of the deployment procedure, we provide a *detailed guide*, clean source code *templates*, working *examples*, and *inference engine binaries* in order to reduce the effort required for deployment. This substantial addition to the written part of this contribution is contained in the `elk-audio-AI-tutorial` repository on the GitHub page of the Creative, Intelligent & Multisensory Interactions Laboratory (CIMIL): `https://github.com/CIMIL/elk-audio-AI-tutorial/`. The guide available in the project's repository provides a more detailed explanation of the deployment process. It will be regularly updated to address potential changes in the new version of Elk Audio OS or IEs.

7.4 Deployment Procedure

This section outlines the procedure for deploying a deep learning model on a Raspberry Pi with the Elk Audio OS. It is worth noting that this same process can be applied for deploying to a VST plugin (excluding cross-compilation and platform-specific steps) across various platforms, including Windows, MacOS, and Linux. All the necessary code and library binaries essential for following this guide are available in the project repository (refer to Section 7.3.4). These resources allow readers to skip parts of the guide for easier deployment (e.g., library compilation can be skipped if using the provided binaries). Further updates, improved instructions, and additional inference engines will be added to the detailed guide in the repository.

The instructions provided in this guide are based on the assumption of utilizing a Unix-based OS (e.g., Ubuntu Linux). However, they are also applicable for users operating on Windows through the Windows Subsystem for Linux (WSL) (Windows Subsystem for Linux) or a Linux virtual machine. As previously mentioned, the instructions in this chapter are tested with version 0.11.0 of Elk Audio OS. Nonetheless, the project repository will be updated to accommodate new versions of the OS.

Figure 7.1 shows an overview of the entire deployment process.

The next sections will describe the following steps:

1. Creation of a JUCE project for the deployment of deep learning models on Elk Audio OS (Section 7.4.1);
2. Cross-compilation of a plugin and its dependencies (Section 7.4.2);
3. Setup and communication with Elk Audio OS (Section 7.4.3);
4. Configuration of Elk's DAW (Section 7.4.4);
5. Troubleshooting (Section 7.4.5);

7.4.1 Project creation

JUCE plugin projects can be created using the Projucer app, which is a component of any JUCE distribution. The Projucer manages the project's configuration, export formats, build systems, and their respective configurations (known as *exporters*). Furthermore, it automates the creation of build files for each exporter. Additionally, the Projucer can execute a user-defined command each time a project is saved, referred to as the *post-export Shell command*. It is also possible to use CMake to set up JUCE projects, but this will not be covered by the guide.

The following two alternatives can be used to prepare a JUCE project to compile VST plugins that will be compatible with Elk Audio OS:

- Use of one of the provided templates;
- Manual project creation.

Here we explain the two methods.

Templates

The project repository (refer to Section 7.3.4) contains two template projects, tailored respectively for ONNX Runtime and TensorFlow Lite. These templates include the precompiled dependencies and the project configuration (`.jucer` file). The project configuration file ensures that the relative inference library is correctly linked, headers are included and it creates a cross-compilation script for Elk Audio OS.

Users should open the `.jucer` file using the Projucer and save the project, thereby creating the essential build folder structure. Subsequently, for any alterations in the project's configuration (e.g., renaming the project), the `.jucer` file should be saved from the Projucer app. Furthermore, users are expected to modify the template code responsible for loading an inference model and executing inference to align with their specific requirements. This customization should be done in accordance with the documentation provided for each engine[5].

Importantly, while Elk Audio OS exclusively manages headless plugins (i.e., without a graphical user interface (GUI)) to optimize processing latency, there is no imperative to remove any GUI code from the plugin editor, as the graphic routines will simply not be called by the Sushi DAW.

Manual Creation

The manual steps required to create a JUCE project for a VST plugin compatible with Elk Audio OS are as follows:

1. Creation of a JUCE project for an audio plugin;
2. Editing of project settings for compatibility with Elk Audio OS;
3. Addition of external library binaries and headers (e.g., TensorFlow Lite or ONNX Runtime);
4. Creation of a Linux exporter;
5. Creation of a cross-compilation script.

Comprehensive details regarding the manual project creation steps can be found in the up-to-date guide available within the project repository.

7.4.2 Cross-compilation for Elk Audio OS

Deploying a plugin on Elk Audio OS, particularly for resource-constrained devices like Raspberry Pi, typically involves cross-compilation. Cross-compilation is a technique where source code is compiled using a cross-compiler on a **host** computer, producing a binary file executable on a **target** computer with a distinct architecture from that

[5] https://www.tensorflow.org/lite/guide/inference#load_and_run_a_model_in_c https://onnxruntime.ai/docs/get-started/with-cpp.html

of the host. This enables the generation of binary executables for embedded devices from a host machine with more powerful hardware. Although native compilation on the Raspberry Pi is feasible, it is not recommended except as a last resort, primarily due to the limited available resources which result in extended computation times (i.e., tens of hours for most engines).

We achieved successful cross-compilation using an x86-64 Linux computer as the **host**, while the target was an ARM 64bit SBC (aarch64 architecture) running the Linux-based real-time Elk Audio OS. To conduct cross-compilation for any program intended for a Raspberry Pi with Elk Audio OS, users should install and use the appropriate Elk-PI SDK corresponding to the chosen OS version[6].

It is not necessary to proceed with the next step, i.e., *Dependencies Compilation* when compiling a project that solely relies on the chosen inference library, as the pre-compiled binaries are included in the project repository (Section 7.3.4).

Two additional possibilities would have been to prepare a Docker image for cross-compilation or a dedicated Yocto recipe. Both alternatives were reportedly made easier with version 1.0 of Elk Audio OS.

Dependencies Compilation

To perform cross-compilation of a plugin, it is necessary to compile all its external dependencies. In the case of a basic plugin incorporating deep learning inference, the only direct dependency is the inference engine library, such as TensorFlow Lite or ONNX Runtime. However, more complex projects may require additional libraries for STFT computation, feature extraction, and more. Moreover, each direct dependency might have sub-dependencies that must be included for compilation (See the TensorFlow Lite template in the project's repository).

Cross-compiling dependencies can however be intricate and may not always be feasible. Setting up cross-compilation, especially for libraries lacking appropriate cross-compilation support, inadequate documentation, or utilizing less recognized build systems, can be challenging. Libraries that rely on the CMake[7] build automation system can generally be integrated effectively with Elk's toolchain. The key steps for cross-compiling such libraries are as follows:

1. Downloading and installing the Elk-PI SDK;

[6] https://github.com/elk-audio/elkpi-sdk/releases
[7] https://cmake.org/

2. Downloading the library source code for the desired version;
3. Creating a `build` folder;
4. Resetting the `LD_LIBRARY_PATH` variable and sourcing the Elk-PI SDK;
5. Executing CMake from the build directory:
   `cmake path/to/CMakeLists/dir/`;
6. Compiling with `make`.

An example demonstrating these concepts for the TensorFlow Lite is available in the project's repository (refer to Section 7.3.4). This example serves to showcase how the actual compilation procedure may deviate from the expected ideal process, often due to unique characteristics of certain sub-dependencies or encountered bugs. In particular, for TensorFlow Lite 2.11.0, specific adjustments are necessary. These include modifying the version of the dependency `FlatBuffers` from 2.0.6 to 2.0.8 and overriding the `CMAKE_SYSTEM_PROCESSOR` variable. The Elk toolchain initially sets this build variable to `cortexa72`, but the `Abseil` dependency requires it to be `aarch64` to prevent incorrect library linking. These corrections were informed by insights gathered from the comments within the Issues section of the Abseil and TensorFlow GitHub repositories. Such posts were found by searching for specific error messages resulting from failed compilation runs. A similar informed trial-and-error approach can be applied when dealing with other libraries. As an example, Figure A.1 shows the compilation script for TensorFlow Lite.

In cases where cross-compilation is not feasible, native compilation can be conducted directly on the Raspberry Pi, albeit at the cost of significantly prolonged completion times. In this case, the Linux build instructions for each library should be followed. This may require separate compilation of additional sub-dependencies if they are not already included in Elk Audio OS or fetched automatically during compilation setup. This was the case for ONNX Runtime, where the compilation process required a few tens of hours on the board. Consequently, a pre-compiled binary is made available in this project's repository.

When a dependency compiles to a dynamic library (i.e., `*.so` files), the binary is necessary both during the linking phase (on the host computer) and the execution phase (on the board) when the library is dynamically loaded. Consequently, the compiled `.so` binary for any dependency must be transferred to the board and placed in one of the system library paths. To determine the dynamic loading paths,

run `echo $LD_LIBRARY_PATH` on the device. As an alternative, if the binary is located in a different directory, the folder containing the binary should be appended to the `LD_LIBRARY_PATH` system variable after each reboot as follows:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/libpath/
```

This can be done manually or automatically. For automatic execution, the export line can be added to either the `~/.bashrc` or `/etc/profile`.

On the contrary, static libraries (i.e., `*.a` files) are automatically included in the plugin binary and do not need to be copied to the board.

Plugins Compilation

Once the IE and other necessary libraries are compiled and appropriately added to the compilation exporter (refer to Projucer configuration in the project repository, Section 7.3.4), the plugin can undergo multiple compilations without the need to re-compile the dependencies.

Assuming a proper Projucer setup, the steps for plugin compilations are as follows:

1. Saving the project from the Projucer app, to create the build structure;
2. Opening the terminal in the `/build/Linux-aarch64` folder;
3. Resetting `LD_LIBRARY_PATH`;
4. Sourcing the Elk-PI SDK;
5. Compiling with `make`, specifying `JUCE_HEADLESS_PLUGIN_CLIENT=1`;
6. For VST3 plugins, renaming the `PluginName.vst3/Contents/arm64-Linux` folder to `aarch64-Linux`.

Additionally, prior to the `make` command, the user can specify additional optimization flags such as the following:

```
export CXXFLAGS="-O3 -pipe -ffast-math -feliminate-unused-debug-types -
    funroll-loops"
```

The following is a simplified compilation script for any plugin project.

```
unset LD_LIBRARY_PATH

source /opt/elk/0.11.0/environment-setup-cortexa72-elk-Linux

export CXXFLAGS="-O3 -pipe -ffast-math -feliminate-unused-debug-types -
    funroll-loops"

AR=aarch64-elk-Linux-ar make \
-j$(nproc) CONFIG=Release\
CFLAGS="-DJUCE_HEADLESS_PLUGIN_CLIENT=1 -Wno-psabi"\
TARGET_ARCH="-mcpu=cortex-a72 -mtune=cortex-a72"
```

7.4.3 Elk Audio OS on the Raspberry

Elk Audio OS v0.11.0 is available as open source for the Raspberry Pi 4 board[8]. Previous versions also support the Raspberry Pi 3. Additionally, support for more SBCs is available under a commercial license. To set up the OS, the OS image should be downloaded from the GitHub repository and then flashed to a high-quality SD card. The Raspberry Pi should be paired with an audio *"hat"* board that is compatible with the OS, such as the HiFiBerry DAC+ ADC and HiFiBerry DAC+ ADC Pro boards.

Once an audio hat board is connected to the Raspberry Pi and the OS SD is inserted, the board should be powered on. The user can access the terminal either using a monitor connected via HDMI or through a remote Secure SHell (SSH) connection. Elk Audio OS operates without a GUI and necessitates control through the terminal or via network protocols like Google Remote Procedure Calls (gRPC) or Open Sound Control (OSC). For remote terminal access, the board can be connected via an ethernet cable to either a network router or directly to a computer. Subsequently, the terminal can be used to connect the board to a Wi-Fi network if desired. The default hostname for Elk-Pi boards is `elk-pi.local`, enabling easy identification of the board within a local network. To confirm the board's connectivity, utilize the `ping` command in the following manner and await a positive reply:

```
ping elk-pi.local
```

The `arp -a` command on a Linux terminal can prove useful in determining the board's IP address if the hostname cannot be resolved. Subsequently, the SSH

[8] https://github.com/elk-audio/elk-pi/releases

protocol can be used to access the terminal remotely. The `ssh` command is available on Linux, MacOS, and the latest Windows 10 and 11 terminals (PowerShell). In the case of earlier versions of Windows or PowerShell, SSH clients like Putty[9] can be used to replace remote terminal and copy functions. To access the board via the terminal, utilize the following command:

```
ssh mind@elk-pi.local
```

The default password is `elk`.

Once a connection is established, files such as the compiled plugin, configuration files, and dynamic libraries can be transferred to the board using `scp`[10] from the host computer:

```
scp -r /path/to/PluginName.vst3 mind@elk-pi.local:~/
scp libonnxruntime.so mind@elk-pi.local:~/
```

7.4.4 DAW configuration: Sushi

Once a VST plugin is copied to the board, it can be loaded into Elk Audio OS's DAW Sushi. Similar to other DAWs, Sushi allows the creation of multiple tracks, each capable of having one or more audio and MIDI inputs and outputs. Every track can feature a chain of plugins, loaded as dynamic libraries during runtime. However, unlike most other DAWs, Sushi does not display or invoke the GUI code of the hosted plugins, and the audio processing callback operates on a hard real-time Xenomai thread to ensure low latency processing. For these reasons, executing the plugin prepared in the previous steps requires the configuration and execution of Sushi. Sushi can be configured in different ways, which include JSON static configurations, the use of the SUSHI gRPC API[11], the elkpi module[12] and the Sushi GUI app[13].

Here we will present the process to configure Sushi through a JSON text file. Below is a configuration file that instructs Sushi to create a mono track with a single audio input and output, and to load the VST3 plugin named "PluginName":

[9] https://www.putty.org/
[10] https://Linux.die.net/man/1/scp
[11] https://github.com/elk-audio/sushi-grpc-api
[12] https://github.com/elk-audio/elkpy
[13] https://github.com/elk-audio/sushi-gui

```
{
  "host_config":{ "samplerate":48000 },
  "tracks":[
    {
      "name":"main",
      "mode":"mono",
      "inputs":[
        {
          "engine_channel":1,
          "track_channel":0
        }
      ],
      "outputs":[
        {
          "engine_channel":1,
          "track_channel":1
        }
      ],
      "plugins":[
        {
          "uid":"PluginName",
          "path":"/path/to/vst/PluginName.vst3",
          "name":"arbitrary_plugin_name",
          "type":"vst3x"
        }
      ]
    }
  ],
  "midi":{
    "cc_mappings":[]
  }
}
```

In particular, the *uid* field should match exactly the VST3 unique identifier, which for JUCE corresponds to the project name (e.g., PluginName for PluginName.vst3). Alternatively, for VST2 plugins, the plugin block in the JSON configuration should contain only the following:

```
  "path":"/path/to/vst/PluginName.so",
  "name":"arbitrary_plugin_name",
  "type":"vst2x"
```

Alternatively, the configuration for a stereo track should use the following `mode`, `inputs`, and `outputs` field values:

```
"mode":"stereo",
"inputs":[ {
    "engine_bus":0,
    "track_bus":0
  } ],
"outputs":[ {
    "engine_bus":0,
    "track_bus":0
  } ]
```

More configuration file examples are provided in the project repository (Section 7.3.4).

Once a configuration file is prepared, Sushi must be executed via the terminal by providing the audio driver type and the configuration file path:

```
sushi -r -c "/path/to/config.json"
```

In this command, the `-r` option is used to specify Elk's RASPA low-latency front-end. Additional options can be appended, such as `-multicore-processing=2` which enables Sushi to use multiple cores. Additionally, adding `&` at the end of the command runs Sushi in the background, allowing continued use of the terminal. To halt background execution later, employ `pkill sushi`. Should Sushi encounter start-up issues, users should inspect the log file `/tmp/sushi.log` for insights. Subsequently, the following section offers a concise overview of diagnostic tools.

7.4.5 Diagnostic tools

By default, Sushi records runtime events and errors in the file `/tmp/sushi.log`. Adjusting the logging level is possible through the `-l` flag. The log can provide insights into errors like mismatched plugin uids or incorrect configuration formats. Furthermore, utilizing the `-timing-statistics` flag prompts Sushi to log the proportion of CPU time utilized for processing. This is particularly significant for real-time execution of deep learning models, where one or more inference operations occur per audio block. It helps determine if the plugin remains within the designated time budget for each call or exceeds it, causing unwanted glitches in the audio output. The subsequent section will offer a concise overview of various execution modes, along with considerations for managing real-time tradeoffs and handling large models.

Elk Audio OS offers diagnostic tools for identifying real-time execution issues. This is crucial as any portion of the code intended for real-time execution must adhere to specific programming principles to ensure real-time safety [127]. These principles encompass avoiding dynamic memory allocation within the audio thread, refraining from using locking mechanisms for concurrent memory access, and not awaiting lower-priority threads (e.g., querying system timers). In essence, these rules can be distilled to the directive of not executing operations on the audio thread that have unbounded or uncertain completion times. While the inference libraries included have been tested and deemed real-time safe [109], it is essential for any user-added code in a plugin to also adhere to these established rules.

Elk Audio OS operates on a dual-kernel system, utilizing Xenomai real-time threads for audio processing. The assessment of real-time safety and troubleshooting of potential violations in user code becomes relatively straight-forward within this framework. Specifically, in Elk Audio OS, non-safe operations detected during the audio callback trigger a **mode switch**, i.e., the system will give control back to the regular Linux kernel to handle unsafe operations. Subsequently, control is returned to the Xenomai kernel. The entire operation is particularly time-consuming. In Elk Audio OS versions up to 0.11.0, monitoring mode switches involves regularly inspecting the `MSW` column in the `/proc/xenomai/sched/stat` file. The following command, for instance, updates its output every three seconds, aiding in this monitoring process:

```
watch -n 3 cat /proc/xenomai/sched/stat
```

Version 1.0.0, which was released while this procedure was being finalized, requires using the `evl ps -s` command instead and looking at the `ISW` counter.

The number of mode switches must remain stable after plugin startup, with a minor allowance of one or two mode switches at startup if not perceptible as artifacts. However, in the case of repeated mode switches, their origin can be traced back to the source code using the gdb debugger as follows:

1. Running the GNU Debugger `gdb` on the sushi executable for the current block size (default 64):

   ```
   gdb sushi_b64
   ```

2. Setting `gdb` to stop whenever the `SIGXCPU` signal is sent by the program:

```
catch signal SIGXCPU
```

3. Running Sushi with the debug-mode-sw flag:

```
r -r --debug-mode-sw -c config.json
```

Lastly, the Elk Audio forum[14] serves as a valuable resource for obtaining support and resolving similar issues.

[14] https://forum.elk.audio/

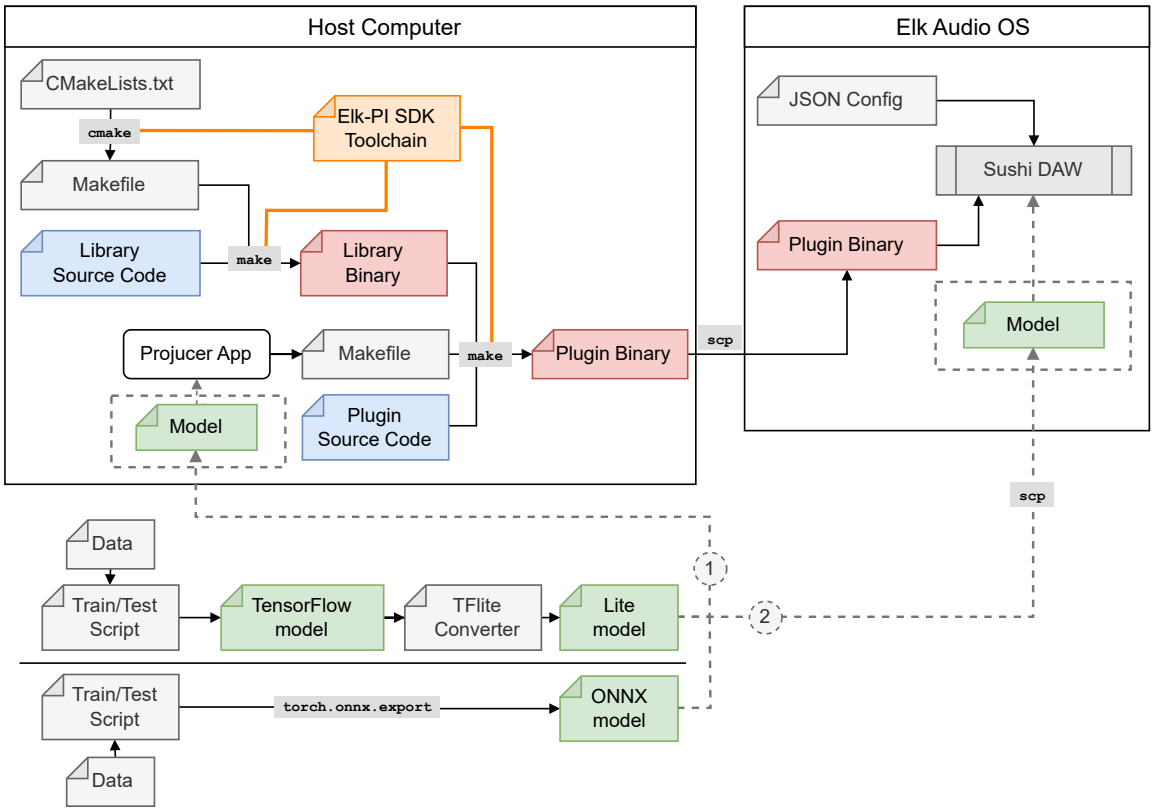


Figure 7.1: *Diagram depicting the deployment process of a deep learning model onto an embedded device running Elk Audio OS. The process starts on a Host computer (top left), where both plugin and dependency compilation take place. This involves utilizing the Elk-PI toolchain to cross-compile the source code. During plugin compilation, the necessary library binaries are linked, resulting in the generation of a binary file for a VST plugin capable of execution on the target device (right). Once the plugin is compiled, it needs to be transferred to the embedded computer running Elk Audio OS. Elk's DAW Sushi allows configuring the loading of the plugin, enabling real-time audio processing. In the lower part of the diagram, the training, testing, and model export phases essential for TensorFlow or frameworks exporting to the ONNX format, such as PyTorch, are depicted. Dashed arrows labeled as 1 and 2 signify two distinct options for model integration: in option 1, the deep learning model is integrated as JUCE BinaryData into the plugin's binary. Conversely, option 2 involves simply copying the model to the target device. For the latter option, the plugin code must be designed to load the model from a path relative to the target's folder structure.*

7.5 Considerations on real-time inference

Applications of machine learning and deep learning in the domain of audio can be categorized into real-time and non-real-time (or offline) cases [191]. It is important to clarify that this categorization is specific to audio tasks and should not be confused with the terms "online" and "offline" used in deep learning research to refer to single-sample and batch learning.

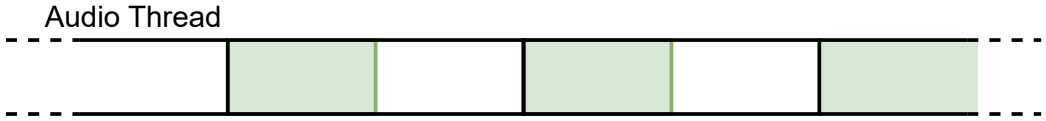


Figure 7.2: *Representation of repeated invocations of the audio processing routine on the real-time thread. The black vertical lines demarcate the time-budget slots allocated for processing each incoming audio buffer. Within this framework, the green boxes signify computations conducted on the input audio buffers. These computations are safely executed within the allotted time budget, ensuring completion before the consumption of the current output buffer and the subsequent reading of the next input buffer.*

In particular, real-time audio analysis and processing algorithms must be designed to continuously process audio data and produce a result (i.e., audio or other data) before pre-defined problem-specific deadlines. An example is a *real-time audio effect*, where the audio processing must occur at a faster pace than the output consumption rate to prevent buffer underruns [126]. Since digital audio is typically buffered into short audio blocks and processed at a fixed rate, this implies that an input buffer consisting of X samples must be processed and transferred to the output within $\frac{X}{\text{samplerate}}$ seconds.

Figure 7.2 illustrates a scenario where computations are carried out safely within the allocated time frame for each invocation of the audio processing routine. This corresponds to the examples found in the project's repository, where inference of a small neural network that models stateless audio saturation is conducted for each sample in the audio buffer. These examples are configured with a default of 64 samples per block and a *samplerate* of 48 kHz. They demonstrate efficient utilization, utilizing approximately 15% (on the Raspberry Pi 4) of the available 1.33 ms for each processing routine call (i.e., $\frac{64}{48{,}000}$).

Nonetheless, the model's execution time depends on the CPU in use (or acceleration hardware like GPUs, if accessible) and the optimizations conducted by the IE (refer to [109]). Consequently, a model that operates proficiently within the designated time limit on a laptop or desktop computer might face challenges on a resource-limited device such as the Raspberry Pi. Figure 7.3 illustrates such a scenario, where a specific set of computations is unable to finalize before the output buffer is consumed and converted to analog, coinciding with the read operation for the new input buffer. Consequently, this leads to noticeable audible artifacts in the output signal due to the corrupted buffers.

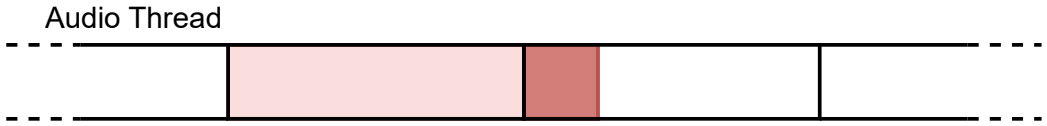


Figure 7.3: *Representation of a set of computations (executed on the real-time audio thread) that exceed the designated time budget, as indicated by the darker shaded area. The darker area represents the overlap with the subsequent call to the audio processing routine. When this occurs, it results in the production of audible artifacts in the output signal.*

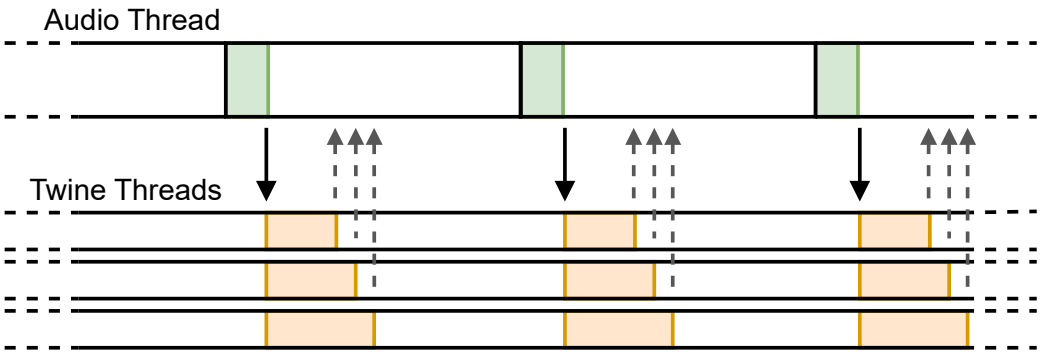


Figure 7.4: *Representation of an audio system where a computationally intensive series of operations must be conducted at the audio callback rate. However, these operations can be fragmented into distinct and independent tasks. In this scenario, various independent sets of computations can be executed in parallel on separate threads. Elk Audio OS facilitates this through the TWINE library, which enables precise control of the parallel execution of threads and the gathering of results of multiple working threads from the audio callback.*

In the instance of a deep learning model that **must execute for each audio routine call** (e.g., for effect modeling), striking a balance between latency and result quality becomes imperative. Potential solutions, often encountered by deep learning developers, include:

1. **Increasing the time budget**. This can be accomplished by increasing the audio block size or diminishing the sample rate. Such decisions unavoidably increase the latency between input and output, yet concurrently expand the processing time budget. A larger audio block size signifies a greater number of samples to process for each call, but it will also diminish function call overhead. It is worth noticing that processors can operate more efficiently by executing a larger batch of operations at once.

2. **Optimizing on the model**. A primary option involves training smaller models for the same task, which can mean having to find a balance between model size and "result quality" (e.g., error or accuracy). Techniques such as transfer learning and knowledge distillation can assist in maintaining satisfactory result quality while utilizing smaller models. Additionally, there are options that do not alter the model's fundamental design and structure, including "quantization" and "pruning". Quantization involves decreasing the weight resolution of the network (e.g., from 32-bit float values to 8-bit integers). On the other hand, pruning gradually sets the weights of the network to zero, reducing the number of multiplications to execute during, if the engine of choice supports sparse execution. It is important to note that these alternatives may potentially lead to reduced accuracy or increased test error in the results.

3. **Parallel execution**. Finally, in rare cases where the set of computations that exceed the time budget can be subdivided into more manageable and independent subsets, these can be assigned to multiple real-time threads. The quad-core (4 cores) Cortex-A72 processor of the Raspberry Pi4 allows the concurrent execution of multiple threads. However, this is only possible if the tasks to run in parallel are completely independent of the results of each other. This situation is depicted in Figure 7.4.

However, not all real-time audio analysis or classification systems need to run in their entirety for every call of the audio processing routine. This is particularly

[14] https://github.com/elk-audio/twine

relevant in event-based systems, where deep signal analysis is required only when a specific event occurs. An example is a note-based real-time guitar technique classifier [80], where the prediction model is executed only upon onset detection. In cases where events are expected to occur less frequently than processing routine calls (e.g., less frequently than once every 1.33 ms for 64 sample blocks and a 48 kHz sample rate), only the detection stage (e.g., rapid onset detection using DSP methods) needs to be executed for each block. In this scenario, the in-depth analysis, potentially involving deep inference, is triggered by detection and can take longer to complete, depending on the minimum inter-event time allowed. However, it is essential to offload the classification to a high-priority thread outside of the real-time kernel to allow inference to take longer than a block without impacting the audio output. If necessary, the results can be moved to the real-time thread after inference has been completed (refer to Figure 7.5). In this case, it is crucial to move input data and results without using locking data structures or unsafe operations. An example illustrating this approach are the expressive guitar technique classifies presented in Chapter 3 and Chapter 6.

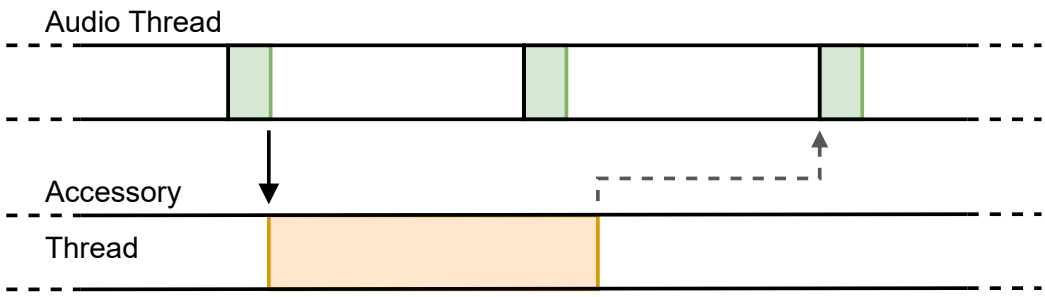


Figure 7.5: *Representation of a digital audio system with a more relaxed real-time constraint compared to those in Fig. 7.2 (i.e., not audio-rate deadlines), where some computationally intensive operations need to be conducted less frequently than calls to the audio callback and can be executed on a separate thread. This situation can arise in event-based deep learning inference [109], where signal analysis occurs only when events are detected. In these cases, event detection can be a less computationally demanding operation performed for each audio block, while the more substantial analysis is allowed to exceed the time budget for a single audio callback call.*

7.6 Other Application and work in progress

Other applications of this guide, and more broadly of the general results obtained in this thesis work, consist of a wide range of tasks that require the execution of deep learning models for audio on embedded devices. For instance, one of the code examples in the project's repository consists of a real-time audio effect that uses a very simple neural network to emulate an audio saturation function. This code sample is provided as a mere example, as it is far from a high-quality neural audio effect, but there are many examples of neural networks used for analog effect modeling that could be executed in a similar way.

Additionally, the procedure described in this guide and previous chapters supported the work done during the 6-months research period spent at the Centre for Digital Music (C4DM) at the Queen Mary University of London. This culminated in the development of an offline embedded classifier for intended emotion from guitar and piano improvisation excerpts. The emotion recognition pipeline is composed of three stages: a recorder, a set of feature extractors, and an instrument-specific deep classifier. The recognition system first records a short emotional piece from the audio input (monaural). The recording is then downsampled to 16 kHz and a wide range of timbral features are extracted using the library *Essentia* [148]. The emotion classifiers used were based on MusiCNN, fitted with additional layers, and trained with transfer-learning [192]. TensorFlow Lite 2.11 was used to execute the emotion classifier in the embedded platform. The recognition system was paired with a mechanism to query a musical database for brief pieces with similar emotions to those that were recognized.

The embedded emotion recognition system is now complete, and it has been undergoing a phase of user studies, to assess the perceived recognition quality. The system has been now tested with 4 musicians: a picture of the user-study setup for a piano player is in Figure 7.6. Figure 7.7 shows an example of the emotional recognition results when analyzed on a laptop after the studies.

After the completion of this offline classifier, the development of a real-time version has started. In this system, emotions will be retrieved from the sound signal every 6 to 9 seconds. Emotional information retrieved in this way will be sent with OSC messages via the network for varied applications for live performance.

Figure 7.6: *User-study for the embedded emotion recognition system, for a piano player.*

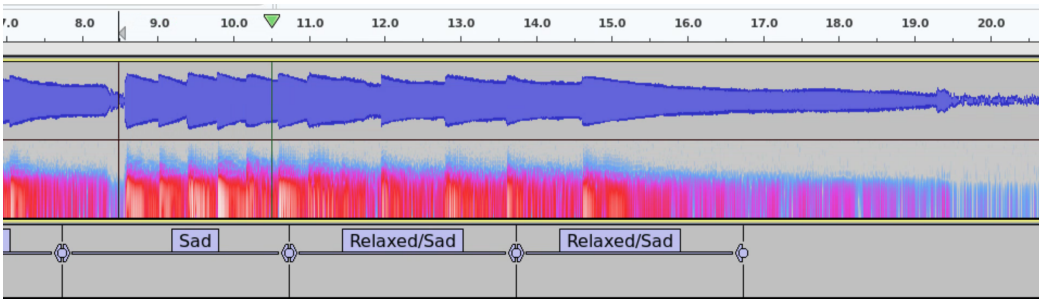


Figure 7.7: *Example of emotion classification results for 3-second audio chunks logged by the embedded recognition system.*

7.7 Summary

This chapter outlined a comprehensive procedure for deploying real-time deep-learning inference on an embedded computer running Elk Audio OS. The procedure covers the steps from creating a compatible code project to executing and diagnosing a VST plugin on a Raspberry Pi. Additionally, we discussed various approaches for achieving real-time execution of deep learning inference on embedded devices and presented alternatives and strategies applicable to larger neural network models.

To accompany this chapter, we have provided an online repository containing a detailed and up-to-date guide, code templates, functional examples, and library binaries for the two supported IEs. The code repository serves the purpose of helping the reader to deploy their models but also to provide updates in case the process is subject to change with future versions of Elk Audio OS and the IEs. This work enables developers and machine learning engineers to start using audio deep learning models on compact embedded computers.

In the short period between its creation and the writing of this thesis, this guide has already been used by a few students and researchers for theses, instrument prototypes, and forthcoming scientific publications.

A limitation of this study is that library cross-compilation can follow a very different process for other libraries, while this study focused mostly on providing a procedure for the TensorFlow Lite and ONNX Runtime libraries. This excludes some of the processing libraries that a developer could need. Lastly, some details of the deployment procedure may become outdated due to updates in Elk Audio OS and IEs. As a result, we have provided a general overview in this chapter, directing readers to a more detailed and up-to-date guide available in the project's repository.

Nevertheless, we believe the outlined procedure represents a coherent process that remains fundamentally similar despite variations in the details. The accompanying code repository will stand as a valuable resource as systems and libraries continue to evolve.

Chapter 8

Conclusions

In the initial stages of our research, we found a lack of studies detailing the real-time execution of deep-learning inference for audio on embedded devices. Conversely, deep-learning methods had gained significant traction in various MIR tasks, including expressive guitar technique recognition. Moreover, we observed that online/real-time execution was rarely considered as a metric in studies that investigated guitar technique recognition. These gaps proved challenging in realizing our vision for a smart guitar, but it became clear that these limitations had broader implications, affecting the development of smart musical instruments and deep-learning-powered audio devices as a whole.

In this thesis, several methods to support the development of a smart guitar have been proposed. In particular, we focused on the task of expressive guitar technique recognition targeting real-time execution on an embedded computer. In addition, the challenges of embedded real-time Music Information Retrieval were investigated and discussed along with potential solutions and tradeoffs. Furthermore, we proposed a real-time-aware optimization method for parametric onset detectors, which takes into account both detection accuracy and latency. Moreover, we compared several tools of embedded real-time deep learning inferences, helping shed light on the different real-time performance granted by different software tools when performing inference of a neural network. Finally, we proposed a procedure for the deployment of real-time deep learning for audio on embedded computers with Elk Audio OS.

We believe the implications of this doctoral research and its results extend past

expressive guitar technique recognition and smart guitars, as the proposed approaches can be applied to many other instruments, and the technical tools investigated and created can help the development of deep-learning-equipped audio devices in general.

The majority of the work presented in this thesis has been presented in international peer-reviewed conferences, as shown in Section 1.2.1 and presented with demos (Section 1.2.3). Additionally, the software output of the studies presented in this thesis, along with the dataset of acoustic guitar playing techniques created, have been made available as open-source (Section 1.2.4). In this chapter, the main contributions of the thesis are summarized and directions for future work are presented.

8.1 Challenges of Embedded Real-time Music Information Retrieval

Chapter 3 presented the challenges of embedded real-time Music Information Retrieval, which include access to the sole past and current input data, tuning the tradeoff between system accuracy and latency, real-time audio deadlines, real-time-safe programming rules, and the limitations of embedded hardware and low-level software. Furthermore, we presented a demonstrative implementation of a real-time embedded expressive guitar technique classifier to illustrate the proposed solutions.

The classifier achieved an accuracy of 99.2% in distinguishing pitched and percussive techniques, along with an average accuracy of 99.1% in distinguishing four distinct percussive techniques, alongside a fifth category dedicated to pitched sounds. Conversely, the task of classifying the full set of twelve different techniques proved to be more difficult, and our proposed approach did not obtain satisfactory results.

Nevertheless, the classification pipeline with the models for the first two tasks was successfully integrated into a Raspberry Pi 4 with the real-time Elk Audio OS, and the classification outcomes were generated with an average latency of roughly 30.7 ms from the note onset. The classification delay was found to be hardly perceptible.

The experiment showcased solutions such as dividing the classification task into a stepwise pipeline with different operating rates, precise tuning of system latency, utilizing a real-time embedded operating system for optimal performance on devices with limited resources, and implementing coding practices that ensure real-time safety.

Limitations and Future Work

The presented classifier, however, has its limitations. First of all, it primarily focused on an attack-based real-time classification approach, and we constructed the classifier as a minimum viable example to showcase the discussed solutions, allowing room for more refined technical choices and tools. Additionally, we employed a simple but limited approach using a Feed-Forward Neural Network to classify 1D feature vectors extracted from the feature window. In summary, future studies could explore two potential directions: 1. Redesigning the classification framework and pipeline to include non-attack-based techniques. 2. Refining the classification pipeline for attack-based techniques, given the considerable room for improvement in the simple approach used here. We opted to pursue the second alternative, as we observed that many non-attack-based techniques can be detected through pitch and envelope tracking (e.g., pitch bending, vibrato).

Moreover, refining the current pipeline presented opportunities for addressing interesting challenges. This includes defining real-time-aware optimizations for onset detectors, fine-tuning feature selection, enhancing the neural classifier, exploring the possibilities and performance of different inference libraries for neural networks on embedded computers, and ultimately facilitating the deployment of deep audio models on embedded devices. These possibilities were further investigated in the following chapters.

The work described Chapter 3 is a contribution presented at the 25$^{\text{th}}$ International Conference on Digital Audio Effects [109].

8.2 Bio-inspired Optimization of Parametric Onset Detectors

In Chapter 4 we introduced a methodology for optimizing the performance of parametric onset detectors. Our approach focused on enhancing detection accuracy and minimizing the time delay, i.e., latency, between the actual onset and its notification. The proposed technique was effectively applied to the onset detectors within the Aubio library. The proposed technique involved an Evolutionary Computation algorithm with which we automated the optimization of input parameters for each detector, utilizing the Pareto front to identify the best solutions. Compared to the commonly used manual optimization, the proposed method decreased the human

effort needed and yielded comparable results. The efficiency of our approach, combined with its short execution time, makes it feasible to apply the method to larger datasets.

When comparing the F1-score values obtained through the evolutionary computation algorithm to those achieved via manual optimization, we observed an average difference of 1.4×10^{-3} F1-score points, with a standard deviation of 1.2×10^{-2}. Additionally, the automated algorithm required 13 hours and 34 minutes to compute the best results, while the manual procedure demanded over two working days of human effort.

Limitations and Future Work

A limitation of the experiments present in our work was the use of only a subset of an audio dataset. The suggested method could derive greater benefits from utilizing the entire dataset of interest. Moreover, the number of parallel optimization instances could be increased, especially with a more powerful computer. A higher degree of parallelism could also be exploited by allowing the evolutionary algorithm to compute the fitness of more than one individual simultaneously, on multiple processing threads.

Furthermore, we did not compare the proposed method with grid-search algorithms as they require scanning a large number of combinations of parameters, without the possibility of automatically focusing the search on potential optima of the search space. Nevertheless, a comparison between the proposed approach and grid search would help further define the strengths and drawbacks of the proposed method. Finally, the optimization performance may be further improved by using fully multi-objective EC algorithms for even less human supervision over the optimizer.

The work described Chapter 4 is a contribution presented at the 24$^{\text{th}}$ International Conference on Digital Audio Effects [135].

8.3 Comparison of Deep Learning Inference Engines for Embedded Real-time Audio Classification

In Chapter 5 we conducted a comparison of four distinct inference engines focusing on real-time audio classification on the CPU of an embedded computer. Our goal was to provide insights into optimized inference engines for efficient deep learning

inference, particularly in the context of real-time audio classification. For our study, we selected models designed for classifying expressive guitar techniques in real-time.

Our findings revealed that many well-known deep learning inference engines are well-suited for real-time audio classification, eliminating the need to resort to specialized and more limited solutions. However, specialized solutions can serve as lightweight and minimal alternatives, especially in scenarios where flexibility is not a primary concern.

While the focus of this comparison was on embedded computers and audio classification, most results are likely to translate or scale to audio plugins for desktop computers and audio processing.

Limitations and Future Work

The limitations of this study lie in the choice of restricting the comparison to feed-forward neural networks and only four inference engines. Future work should explore more inference engines and investigate performance differences with a wider range of deep learning models, such as recurrent and convolutional neural networks. Additionally, extending this comparison to slower CPUs and testing with quantized neural network models could offer valuable insights.

Chapter 5 discussed a contribution presented at the 25th International Conference on Digital Audio Effects [109].

8.4 Embedded Real-Time Expressive Guitar Technique Recognition

In Chapter 6 we introduced a flexible-latency embedded real-time expressive guitar technique classifier. Furthermore, we explored the influence of task requirements and data characteristics on recognition performance.

Our findings indicate that easing the latency constraints, particularly in the range of 15 to 45 milliseconds, can enhance recognition accuracy for both pitched and percussive techniques. Interestingly, the performance for percussive techniques is mostly unaffected and may even experience slight degradation with larger feature windows.

Additionally, we observed a trend where models tend to overfit specific guitar and player characteristics, limiting their ability to learn broader properties of expressive

techniques and hampering generalization performance across different instruments and players. This effect was observed even within the very first $\tilde{1}4$ ms of musical notes in the signal.

We effectively tackled this challenge by implementing grouped k-fold cross validation, ensuring the separation of guitar and player pairs during training and testing. We showed how this can yield accurate metrics. In contrast, the accuracy metrics obtained through simple k-fold cross-validation proved to be misleading, as they increased with reduced real generalization performance. Furthermore, our emphasis on a single instrument revealed that while it could lead to improved performance, the distinctive touch or style of different guitarists is likely to influence recognition performance. The classifier was successfully implemented and deployed to a single-board computer.

Limitations and Future Work

Some of the limitations of the study include the relatively small size of the additional dataset used for the guitarist's touch experiment and the focus solely on recognizing techniques in monophonic signals. To address this, future work should handle polyphony through the use of a hexaphonic pickup. Additionally, future endeavors should extend the classification task to more techniques, such as hammer-on, pull-off, and bending, which could be integrated thanks to a real-time pitch tracker. Moreover, future studies should focus on a more comprehensive investigation into the impact of the "Guitarist's Touch" on recognition, informing the integration of new data into the expressive guitar technique dataset.

Finally, this study presented a review of the performance of an expressive guitar technique classifier for different latency constraints and under different characteristics, while the perceptual effect of recognition latency on musicians was outside of the scope of the study. In fact, the perception of latency would depend on whether the playing technique information is repurposed as audio, video, or other media. A future study will focus on the development of an application that repurposes the technique information for sound synthesis, and different degrees of latency will be tested with musicians in order to understand their reactions.

Chapter 6 discussed a contribution submitted to IEEE/ACM Transactions on Audio, Speech, and Language Processing.

8.5 Real-Time Embedded Deep Learning on Elk Audio OS

In Chapter 7 we presented a detailed procedure for deploying real-time deep learning inference on an embedded computer running Elk Audio OS. The outlined process encompasses the entire workflow, from creating a compatible code project to executing and diagnosing a VST plugin on a Raspberry Pi.

Furthermore, the chapter discussed different approaches for achieving real-time execution of deep learning inference on embedded devices. Additionally, we discussed alternatives and strategies applicable to larger neural network models.

To complement the discussion in Chapter 7, we provided an online repository with a comprehensive and updated guide, code templates, functional examples, and library binaries for the two supported inference engines. The repository serves not only as a practical resource for readers to deploy their models but also as a hub for updates, ensuring adaptability to potential changes in the deployment process with future versions of Elk Audio OS and the inference engines. This work enables developers and machine learning engineers to start deploying audio deep learning models on compact embedded computers.

Notably, in the brief period since its inception and the writing of this thesis, several students and researchers have already used this guide for their theses, instrument prototypes, and upcoming scientific publications.

Limitations and Future Work

A limitation of this study is that the process of library cross-compilation can vary significantly across different libraries, while this study predominantly focused on providing a procedure for the TensorFlow Lite and ONNX Runtime libraries. This excludes certain processing libraries that developers might require for Music Information Retrieval or applications based on audio spectrograms.

Additionally, some details of the deployment procedure may become outdated due to updates in Elk Audio OS and inference engines. As a solution, we have offered a general overview in this chapter, guiding readers to a more detailed and continuously updated guide available in the project's repository. Nevertheless, we believe the outlined procedure represents a coherent process that remains fundamentally similar despite variations in the details. The accompanying code repository will serve as a

valuable resource as systems and libraries continue to evolve.

Chapter 7 described a contribution presented at the 4th International Symposium on the Internet of Sounds [180].

8.6 Concluding Remarks

I am grateful to have had the possibility to contribute to scientific knowledge in the research fields I delved into during my PhD, which were relatively unexplored at its beginning. I am also grateful to have had the possibility to put my best efforts and time into research work that produced results that are not purely theoretical, and that could help the creation of tools and musical instruments and, in turn, be used to create music.

At the same time, I am aware of the entity of the research efforts that still need to be put into smart musical instruments to achieve precise and quick tracking of musical properties on resource-constrained devices, as well as the refinement and improvement necessary for the tools that enable deep learning inference for audio on embedded devices. Finally, I hope that the contribution presented with this thesis can inspire and help researchers and developers to improve the technology that supports smart musical instruments and the execution of deep-learning inference for real-time audio on embedded devices.

Bibliography

[1] Joseph A Paradiso, "Electronic music: new ways to play," *IEEE Spectrum*, vol. 34, no. 12, pp. 18–30, 1997.

[2] Noboru Suenaga, "GUITAR SYNTHESIZER," USA Patent US4 357 852A, May 15, 1980, assignee: Roland Corporation.

[3] David Friend, "An integrated guitar synthesizer for live performance," in *Audio Engineering Society Convention 58*. Audio Engineering Society, 1977.

[4] Matrixsynth, "360 System Spectre Oberheim SEM Synthesizer," https://www.matrixsynth.com/2023/04/360-system-spectre-oberheim-sem.html, Aug. 2023.

[5] Tonehome.de, "Jen GS-3000 Syntar," https://www.tonehome.de/jen/gs-3000-syntar/, 2023.

[6] Fabián Esqueda, Otso Lähdeoja, and Vesa Välimäki, "Algorithms for guitar-driven synthesis: Application to an augmented guitar," in *Proceedings of the 15th Sound and Music Computing Conference: Sonic Crossings, SMC 2018*, 2018, pp. 444–451.

[7] Steve Hackett, "Roland GR-500," in *Sound International Magazine*, Dec. 1978. [Online]. Available: https://www.joness.com/gr300/pdf/steve-hackett-roland-gr-500.pdf

[8] Tom Mullen, "Interview with Robert Fripp," *Guitar Player Magazine*, Jun. 1986. [Online]. Available: https://www.joness.com/gr300/fripp.htm

[9] William A. Aitken, Anthony J. Sedivy, and Michael S. Dixon, "Electronic Musical Instrument," USA Patent USD289 900S, Aug 7, 1984.

[10] Jacob Harrison, Robert H Jack, Fabio Morreale, and Andrew P. McPherson, "When is a Guitar not a Guitar? Cultural Form, Input Modality and Expertise," in *Proceedings of the International Conference on New Interfaces for Musical Expression*. Zenodo, Jul. 2018, pp. 299–304. [Online]. Available: https://doi.org/10.5281/zenodo.1302589

[11] "Sonuus website - G2M midi converter," https://www.sonuus.com/products_g2m.html, accessed: 2023-12-29.

[12] "Fishman website - TriplePlay MIDI Guitar Controller," https://www.fishman.com/tripleplay/, accessed: 2023-12-29.

[13] "Jam Origin - MIDI Guitar - website," https://www.jamorigin.com/, accessed: 2023-12-29.

[14] Robert H Jack, Tony Stockman, and Andrew McPherson, "Rich gesture, reduced control: the influence of constrained mappings on performance technique," in *Proceedings of the 4th International Conference on Movement Computing*, 2017, pp. 1–8.

[15] Duncan MacConnell, Shawn Trail, George Tzanetakis, Peter Driessen, Wyatt Page, and N Wellington, "Reconfigurable autonomous novel guitar effects (range)," in *Proceedings of the international conference on sound and music computing*, 2013.

[16] Markus Schedl, Emilia Gómez, and Julián Urbano, "Music information retrieval: Recent developments and applications," *Foundations and Trends® in Information Retrieval*, vol. 8, no. 2-3, pp. 127–261, 2014.

[17] Adam R Tindale, Ajay Kapur, George Tzanetakis, and Ichiro Fujinaga, "Retrieval of percussion gestures using timbre classification techniques." in *ISMIR*, 2004.

[18] Esteban Maestre, "Modeling instrumental gestures: an analysis/synthesis framework for violin bowing," Ph.D. dissertation, Department of Information and Communication Technologies, Universitat Pompeu Fabra, 2009.

[19] Caroline Traube, Philippe Depalle, and Marcelo Wanderley, "Indirect acquisition of instrumental gesture based on signal, physical and perceptual information," in *Proceedings of the 2003 conference on New interfaces for musical expression*, 2003, pp. 42–47.

[20] Yudong Zhao, Changhong Wang, György Fazekas, Emmanouil Benetos, and Mark Sandler, "Violinist identification based on vibrato features," in *2021 29th European Signal Processing Conference (EUSIPCO)*. IEEE, 2021, pp. 381–385.

[21] Brian CJ Moore, *An introduction to the psychology of hearing*. Brill, 2012.

[22] L. Turchet, C. Fischione, G. Essl, D. Keller, and M. Barthet, "Internet of Musical Things: Vision and Challenges," *IEEE Access*, vol. 6, pp. 61 994–62 017, 2018. [Online]. Available: https://doi.org/10.1109/ACCESS.2018.2872625

[23] Andrew McPherson and Victor Zappi, "An environment for submillisecond-latency audio and sensor processing on BeagleBone Black," in *Proceedings of the AES 138th Convention, Warsaw, Poland*, 2015.

[24] Luca Turchet and Carlo Fischione, "Elk Audio OS: an open source operating system for the Internet of Musical Things," *ACM Transactions on the Internet of Things*, vol. 2, no. 2, pp. 1–18, 2021.

[25] Luca Turchet, "Smart Musical Instruments: Vision, Design Principles, and Future Directions," *IEEE Access*, vol. 7, pp. 8944–8963, 2019.

[26] Teresa Pelinski, Victor Shepardson, Steve Symons, Franco Santiago Caspe, Adan L Benito Temprano, Jack Armitage, Chris Kiefer, Rebecca Fiebrink, Thor Magnusson, and Andrew McPherson, "Embedded AI for NIME: Challenges and Opportunities," *International Conference on New Interfaces for Musical Expression*, Jun. 2022. [Online]. Available: https://nime.pubpub.org/pub/rwr2c3zs

[27] William Brent, "A Timbre Analysis And Classification Toolkit For Pure Data," in *Proceedings of the 2010 International Computer Music Conference, ICMC 2010, New York, USA, 2010.* Michigan Publishing, 2010.

[28] Domenico Stefani and Luca Turchet, "aGPTset (acoustic Guitar Playing Technique dataset)," Nov. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.10159492

[29] Luca Turchet, Michele Benincaso, and Carlo Fischione, "Examples of use cases with smart instruments," in *Proceedings of the 12th international audio mostly conference on augmented and participatory sound and music experiences*, 2017, pp. 1–5.

[30] Luca Turchet, Andrew McPherson, and Carlo Fischione, "Smart instruments: Towards an ecosystem of interoperable devices connecting performers and audiences," in *Proceedings of Sound and Music Computing Conference*. http://www. smcnetwork. org/, 2016, pp. 498–505.

[31] Eduardo Reck Miranda and Marcelo M Wanderley, *New digital musical instruments: control and interaction beyond the keyboard.* AR Editions, Inc., 2006, vol. 21.

[32] Alex Mulder, "Towards a choice of gestural constraints for instrumental performers," *Trends in gestural control of music*, vol. 315, p. 335, 2000.

[33] Caroline Hummels, Gerda Smets, and Kees Overbeeke, "An intuitive two-handed gestural interface for computer supported product design," in *I. Wachsmuth and M. Frohlich (Eds.), Gesture and Sign Language in Human-computer Interaction: Proceedings of the II Gesture Workshop.* Springer-Verlag, 1997, pp. 197–208.

[34] Pierre Feyereisen and Jacques-Dominique De Lannoy, *Gestures and speech: Psychological investigations.* Cambridge University Press, 1991.

[35] Rolf Inge Godøy and Marc Leman, *Musical gestures: Sound, movement, and meaning.* Routledge, 2010.

[36] Ralf Steinmetz, "Human perception of jitter and media synchronization," *IEEE Journal on selected Areas in Communications*, vol. 14, no. 1, pp. 61–72, 1996.

[37] Tod Machover and Joe Chung, "Hyperinstruments: Musically Intelligent and Interactive Performance and Creativity Systems," in *Proceedings of the 1989 International Computer Music Conference (ICMC'89)*, 1989, pp. 186–190.

[38] Tod Machover, *Hyperinstruments: A Progress Report, 1987-1991*. MIT Media Laboratory, 1992.

[39] Dan Overholt, Edgar Berdahl, and Robert Hamilton, "Advancements in actuated musical instruments," *Organised Sound*, vol. 16, no. 2, pp. 154–165, 2011.

[40] Luca Turchet, "Some Reflections on the Relation between Augmented and Smart Musical Instruments," in *Proceedings of the Audio Mostly 2018 on Sound in Immersion and Emotion*, ser. AM '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3243274.3243281

[41] Andrew McPherson, "The Magnetic Resonator Piano: Electronic Augmentation of an Acoustic Grand Piano," *Journal of New Music Research*, vol. 39, no. 3, pp. 189–202, 2010. [Online]. Available: https://doi.org/10.1080/09298211003695587

[42] ——, "Buttons, Handles, and Keys: Advances in Continuous-Control Keyboard Instruments," *Computer Music Journal*, vol. 39, no. 2, pp. 28–46, 06 2015. [Online]. Available: https://doi.org/10.1162/COMJ_a_00297

[43] Frédéric Bevilacqua, Nicolas H Rasamimanana, Emmanuel Fléty, Serge Lemouton, and Florence Baschet, "The augmented violin project: research, composition and performance report," in *6th International Conference on New Interfaces for Musical Expression (NIME 06)*, 2006, pp. 402–406.

[44] Daniel Overholt, "The overtone fiddle: an actuated acoustic instrument," in *New Interfaces for Musical Expression*. University of Oslo, 2011, pp. 4–7.

[45] Adrian Freed, David Wessel, Michael Zbyszynski, and Frances Marie Uitti, "Augmenting the cello," in *Proceedings of the 2006 conference on new interfaces for musical expression*, 2006, pp. 409–413.

[46] Joseph Thibodeau and Marcelo M. Wanderley, "Trumpet Augmentation and Technological Symbiosis," *Computer Music Journal*, vol. 37, no. 3, pp. 12–25, 09 2013. [Online]. Available: https://doi.org/10.1162/COMJ_a_00185

[47] Cléo Palacio-Quintin, "The hyper-flute," in *Proceedings of the International Conference on New Interfaces for Musical Expression*. Zenodo, 2003, pp. 206–207. [Online]. Available: https://doi.org/10.5281/zenodo.1176549

[48] Luca Turchet, "The hyper-hurdy-gurdy," in *Proceedings of the Sound and Music Computing Conference*, 2016, pp. 491–497.

[49] ——, "The hyper-zampogna." in *Proceedings of Sound and Music Computing Conference*, 2016, pp. 485–490.

[50] Duncan Menzies and Andrew McPherson, "An Electronic Bagpipe Chanter for Automatic Recognition of Highland Piping Ornamentation," in *NIME*, 2012.

[51] Luca Turchet, "The Hyper-Mandolin," in *Proceedings of the 12th International Audio Mostly Conference on Augmented and Participatory Sound and Music Experiences*, ser. AM '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3123514.3123539

[52] Teemu Maki-Patola, Perttu Hämäläinen, and Aki Kanerva, "The augmented djembe drum: sculpting rhythms," in *Proceedings of the 2006 conference on New interfaces for musical expression*, 2006, pp. 364–369.

[53] Miller Puckette, "Patch for guitar," in *Pure Data Convention, Montreal*, Feb. 2007, pp. 1–5.

[54] Nicolas Bouillot and M Wozniewski, "A Mobile Wireless Augmented Guitar," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, no. October 2014, 2008, pp. 189–192.

[55] Loïc Reboursière, Christian Frisson, Otso Lähdeoja, John A Mills, Cécile Picard-Limpens, and Todor Todoroff, "Multimodal Guitar: A Toolbox For Augmented Guitar Performances." in *Proceedings of the 2010 Conference on New Interfaces for Musical Expression (NIME)*, 2010, pp. 415–418.

[56] Iñigo Angulo, Sergio Giraldo, and Rafael Ramirez, "Hexaphonic guitar transcription and visualization," in *Proceedings of the International Conference on Technologies for Music Notation and Representation (TENOR)*, 2016, pp. 187 – 192.

[57] Andrea Martelloni, Andrew McPherson, and Mathieu Barthet, "Guitar augmentation for Percussive Fingerstyle: Combining self-reflexive practice and user-centred design," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, Shanghai, China, 2021.

[58] ——, "Real-time percussive technique recognition and embedding learning for the acoustic guitar," in *Proceedings of the 24th International Society for Music Information Retrieval Conference*. ISMIR, Nov. 2023.

[59] Otso Lähdeoja, "An augmented guitar with active acoustics," in *Proceedings of the 12th International Conference in Sound and Music Computing (SMC-15)*, 2015, pp. 85–89.

[60] Eduardo AL Meneses and José Fornari, "Guitarami: um instrumento musical aumentado que transpoe restriçoes intrınsecas do violao," in *Proceedings of the 15th Brazilian Symposium on Computer Music, Campinas/SP*, 2015.

[61] Eduardo AL Meneses and Marcelo M Wanderley, "New developments on the augmentation of a classical guitar: Addition of embedded sound synthesis and osc communication over network," in *Proc. 16th Brazilian Symposium on Computer Music (SBCM), Sao Paulo, Brazil*, 2017.

[62] Eduardo AL Meneses, Sérgio Freire, and Marcelo M Wanderley, "GuitarAMI and GuiaRT: two independent yet complementary augmented nylon guitar projects," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, 2018, pp. 222–227.

[63] Sérgio Freire, Augusto Armondes, and Rubens Silva, "Real-Time Symbolic Transcription and Interactive Transformation Using a Hexaphonic Nylon-String Guitar," *Computer Music Journal*, vol. 45, no. 4, pp. 20–39, 12 2021. [Online]. Available: https://doi.org/10.1162/comj_a_00625

[64] Ricky Graham and Brian Bridges, "Gesture and Embodied Metaphor in Spatial Music Performance Systems Design." in *Proceedings of the International Conference on New Interfaces for Musical Expression*, 2014, pp. 581–584.

[65] Amit Amit Shlomo Zoran, "Chameleon Guitar - a physical heart in a digital instrument," (Master thesis) MIT Media Lab, USA, 2009.

[66] "Elk Audio website," https://www.elk.audio/, accessed: 2023-09-29.

[67] Pranay Dighe, Parul Agrawal, Harish Karnick, Siddartha Thota, and Bhiksha Raj, "Scale independent raga identification using chromagram patterns and swara based features," in *2013 IEEE International Conference on Multimedia and Expo Workshops (ICMEW)*, 2013, pp. 1–4.

[68] Sergio Freire and Lucas Nézio, "Study of the tremolo technique on the acoustic guitar: Experimental setup and preliminary results on regularity," in *Proc. Int. Conf. Sound and Music Computing, Stockholm*, 2013, pp. 329–334.

[69] Sérgio Freire and Pedro Cambraia, "Analysis of musical textures played on the guitar by means of real-time extraction of mid-level descriptors," in *12th International Conference on Sound and Music Computing. Proceedings... Maynooth: Maynooth University*, vol. 1, 2015, pp. 509–514.

[70] Miller Puckette, "The Patcher," in *Proceedings of the International Computer Music Conference (ICMC)*. International Computer Music Association, 1988, pp. 420–429.

[71] Otso Lähdeoja, Marcelo M Wanderley, and Joseph Malloch, "Instrument augmentation using ancillary gestures for subtle sonic effects," *Proc. SMC*, pp. 327–330, 2009.

[72] Amit Zoran and Joseph A. Paradiso, "The chameleon guitar-guitar with a replaceable resonator," *Journal of New Music Research*, vol. 40, no. 1, pp. 59–74, 2011.

[73] "Line6 website - Variax Modeling Guitars," https://line6.com/variax-modeling-guitars/, accessed: 2023-09-30.

[74] "Roland website - VG Stratocaster Guitar," https://www.roland.com/products/g-5/, accessed: 2023-09-30.

[75] Roy Vanegas, "The MIDI pick: Trigger serial data, samples, and MIDI from a guitar pick," in *Proceedings of the 7th international conference on New interfaces for musical expression*, 2007, pp. 330–332.

[76] Luca Turchet, Andrew McPherson, and Mathieu Barthet, "Co-design of a smart cajón," *Journal of the Audio Engineering Society*, vol. 66, no. 4, pp. 220–230, 2018.

[77] ——, "Real-time hit classification in a smart cajón," *Frontiers in ICT*, vol. 5, p. 16, 2018.

[78] Luca Turchet, "Smart mandolin: autobiographical design, implementation, use cases, and lessons learned," in *Proceedings of the Audio Mostly 2018 on Sound in Immersion and Emotion*, 2018, pp. 1–7.

[79] David Wessel and Matthew Wright, "Problems and prospects for intimate musical control of computers," *Computer music journal*, vol. 26, no. 3, pp. 11–22, 2002.

[80] Domenico Stefani and Luca Turchet, "On the Challenges of Embedded Real-Time Music Information Retrieval," in *Proceedings of the 25-th International Conference on Digital Audio Effects (DAFx20in22)*, vol. 3, Sep. 2022, pp. 177–184.

[81] Claude Cadoz and Marcelo M Wanderley, "Gesture-music," in *Trends in Gestural Control of Music*. IRCAM, 2000, pp. 71—-94.

[82] Vincent Lostanlen, Joakim Andén, and Mathieu Lagrange, "Extended playing techniques: the next milestone in musical instrument recognition," in *Proceedings of the 5th International Conference on digital libraries for musicology*, 2018, pp. 1–10.

[83] Michel Bernays and Caroline Traube, "Expressive production of piano timbre: touch and playing techniques for timbre control in piano performance," in *Proceedings of the 10th Sound and Music Computing Conference (SMC2013)*. KTH Royal Institute of Technology Stockholm, Sweden, 2013, pp. 341–346.

[84] Mauricio Alves Loureiro, Hugo Bastos de Paula, and Hani C Yehia, "Timbre classification of a single musical instrument." in *ISMIR*. Barcelona, 2004.

[85] Diana Young, "Classification of common violin bowing techniques using gesture data from a playable measurement system." in *NIME*, 2008, pp. 44–48.

[86] Luwei Yang, Elaine Chew, and Sayid-Khalid Rajab, "Cross-cultural comparisons of experssivity in recorded erhu and violin music: Performer vibrato styles," in *the 4th International Workshop on Folk Music Analysis*, 2014.

[87] Tan Hakan Özaslan, Enric Guaus, Eric Palacios, and Josep Lluis Arcos, "Attack based articulation analysis of nylon string guitar," in *Proceedings of the 7th International Symposium on Computer Music Modeling and Retrieval (CMMR)*, 2010.

[88] Loïc Reboursière, Otso Lähdeoja, Ricardo Chesini Bose, Thomas Drugman, Stéphane Dupont, Cécile Picard-Limpens, and Nicolas Riche, "Guitar as Controller," *Numediart Quartely Progress Scientific Report*, vol. 4(3), no. 3, 2011.

[89] Otso Lähdeoja, Loïc Reboursière, Thomas Drugman, Stéphane Dupont, Cécile Picard-Limpens, and Nicolas Riche, "Détection Des Techniques De Jeu De La Guitare," *Journées d'Informatique Musicale*, 2012.

[90] Raphael Foulon, Pierre Roy, and François Pachet, "Automatic classification of guitar playing modes," in *International Symposium on Computer Music Multidisciplinary Research*. Springer, 2013, pp. 58–71.

[91] Jakob Abeßer, Hanna Lukashevich, and Gerald Schuller, "Feature-based extraction of plucking and expression styles of the electric bass guitar," in *2010 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2010, pp. 2290–2293.

[92] Caroline Traube and Philippe Depalle, "Extraction of the excitation point location on a string using weighted least-square estimation of a comb filter delay," in *Proceedings of the 6th International Conference on Digital Audio Effects (DAFx-03)*, 2003.

[93] Henri Penttinen and Vesa Välimäki, "A time-domain approach to estimating the plucking point of guitar tones obtained with an under-saddle pickup," *Applied Acoustics*, vol. 65, no. 12, pp. 1207–1220, 2004.

[94] Christian Kehling, Jakob Abeßer, Christian Dittmar, and Gerald Schuller, "Automatic Tablature Transcription of Electric Guitar Recordings by Estimation of Score-and Instrument-Related Parameters," in *DAFx*, 2014, pp. 219–226.

[95] Ana M. Barbancho, Anssi Klapuri, Lorenzo J. Tardon, and Isabel Barbancho, "Automatic transcription of guitar chords and fingering from audio," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 20, no. 3, pp. 915–921, 2012.

[96] Isabel Barbancho, George Tzanetakis, Ana M. Barbancho, and Lorenzo J. Tardón, "Discrimination Between Ascending/Descending Pitch Arpeggios," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 26, no. 11, pp. 2194–2203, 2018.

[97] Yuan-Ping Chen, Li Su, and Yi-Hsuan Yang, "Electric Guitar Playing Technique Detection in Real-World Recording Based on F0 Sequence Pattern Recognition," in *ISMIR*, 2015, pp. 708–714.

[98] Li Su, Li-fan Yu, and Yi-hsuan Yang, "Sparse Cepstral and Phase Codes for Guitar Playing Technique Classification," in *Proceedings of the 15th International Society for Music Information Retrieval Conference (ISMIR)*, 2014, pp. 9–14.

[99] Ting-Wei Su, Yuan-Ping Chen, Li Su, and Yi-Hsuan Yang, "TENT: Technique-Embedded Note Tracking for Real-World Guitar Solo Recordings," *Transactions of the International Society for Music Information Retrieval*, vol. 2, no. 1, pp. 15–28, 2019.

[100] Andrea Martelloni, Andrew McPherson, and Mathieu Barthet, "Percussive Fingerstyle Guitar through the Lens of NIME: an Interview Study," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, Jun. 2020, pp. 440–445. [Online]. Available: https://doi.org/10.5281/zenodo.4813463

[101] Ivan Franco and Marcelo M. Wanderley, "Prynth: A framework for self-contained digital music instruments," in *In Proceedings of the 12th International Symposium on Computer Music Multidisciplinary Research (CMMR)*, 2016, pp. 357–370.

[102] Edgar Berdahl and Wendy Ju, "Satellite CCRMA: A musical interaction and sound synthesis platform," in *Proceedings of the Conference on New Interfaces for Musical Expression (NIME)*, 2011, pp. 173–178.

[103] Andrew McPherson, "Bela: An embedded platform for low-latency feedback control of sound," *The Journal of the Acoustical Society of America*, vol. 141, no. 5 Supplement, pp. 3618–3618, May 2017. [Online]. Available: https://doi.org/10.1121/1.4987761

[104] Alec Wright, Eero-Pekka Damskägg, Lauri Juvela, and Vesa Välimäki, "Real-Time Guitar Amplifier Emulation with Deep Learning," *Applied Sciences*, vol. 10, no. 3, 2020. [Online]. Available: https://www.mdpi.com/2076-3417/10/3/766

[105] "TensorFlow lite - website," https://www.tensorflow.org/lite, accessed: 2023-09-30.

[106] "TorchScript - website," https://pytorch.org/docs/stable/jit.html, accessed: 2023-09-30.

[107] "ONNX RUntime - website," https://onnxruntime.ai/, accessed: 2023-09-30.

[108] Jatin Chowdhury, "RTNeural: Fast Neural Inferencing for Real-Time Systems," *arXiv preprint arXiv:2106.03037*, 2021.

[109] Domenico Stefani, Simone Peroni, and Luca Turchet, "A Comparison of Deep Learning Inference Engines for Embedded Real-Time Audio Classification," in *Proceedings of the 25-th International Conference on Digital Audio Effects (DAFx20in22)*, vol. 3, Sept. 2022, pp. 256–263.

[110] Sebastian Böck and Markus Schedl, "Enhanced beat tracking with context-aware neural networks," in *Proceedings of the 14th International Conference on Digital Audio Effects (DAFx-11)*, 2011, pp. 135–139.

[111] Juan Pablo Bello, Laurent Daudet, Samer Abdallah, Chris Duxbury, Mike Davies, and Mark B Sandler, "A tutorial on onset detection in music signals," *IEEE Transactions on Speech and Audio Processing*, vol. 13, pp. 1035–1047, 2005.

[112] Siddharth Sigtia, Emmanouil Benetos, and Simon Dixon, "An End-to-End Neural Network for Polyphonic Piano Music Transcription," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 24, pp. 927–939, 2016.

[113] George Tzanetakis and Perry Cook, "Musical genre classification of audio signals," *IEEE Transactions on Speech and Audio Processing*, vol. 10, no. 5, pp. 293–302, 2002.

[114] Edgar Berdahl, "How to make embedded acoustic instruments," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*. London, United Kingdom: Goldsmiths, University of London, Jun. 2014, pp. 140–143. [Online]. Available: http://www.nime.org/proceedings/2014/nime2014_551.pdf

[115] Giulio Moro, Astrid Bin, Robert H Jack, Christian Heinrichs, and Andrew McPherson, "Making high-performance embedded instruments with bela and pure data," in *Proceedings of the International Conference on Live Interfaces (ICLI)*. University of Sussex, 2016.

[116] Andrew McPherson, Robert Jack, and Giulio Moro, "Action-Sound Latency: Are Our Tools Fast Enough?" in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*. Brisbane, Australia: Queensland Conservatorium Griffith University, 2016, pp. 20–25.

[117] Jurgen Vandendriessche, Nick Wouters, Bruno da Silva, Mimoun Lamrini, Mohamed Yassin Chkouri, and Abdellah Touhafi, "Environmental Sound Recognition on Embedded Systems: From FPGAs to TPUs," *Electronics*, vol. 10, no. 21, 2021. [Online]. Available: https://www.mdpi.com/2079-9292/10/21/2622

[118] Romain Michon, Yann Orlarey, Stéphane Letz, and Dominique Fober, "Real Time Audio Digital Signal Processing With Faust and the Teensy," in *Sound and Music Computing Conference (SMC-19)*, Malaga, Spain, May 2019. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03153709

[119] Tanguy Risset, Romain Michon, Yann Orlarey, Stéphane Letz, Gero Müller, and Adeyemi Gbadamosi, "Faust2FPGA for Ultra-Low Audio Latency: Preliminary work in the Syfala project," in *Second International Faust Conference*, Paris, France, Dec. 2020, pp. 1–9. [Online]. Available: https://hal.inria.fr/hal-03116958

[120] Bhuwan Bhattarai and Joonwhoan Lee, "Automatic music mood detection using transfer learning and multilayer perceptron," *The International Journal of Fuzzy Logic and Intelligent Systems*, vol. 19, no. 2, pp. 88–96, Jun 2019. [Online]. Available: https://doi.org/10.5391/IJFIS.2019.19.2.88

[121] Yang Yu, Sen Luo, Shenglan Liu, Hong Qiao, Yang Liu, and Lin Feng, "Deep attention based music genre classification," *Neurocomputing*, vol. 372, pp. 84–91, 1 2020.

[122] Florian Eyben, Sebastian Böck, Björn Schuller, and Alex Graves, "Universal onset detection with bidirectional long-short term memory neural networks," in *Proceedings of the 11th International Society for Music Information Retrieval Conference (ISMIR), Utrecht, The Netherlands*, 2010, pp. 589–594.

[123] Dan Stowell and Mark Plumbley, "Adaptive whitening for improved real-time audio onset detection," in *Proceedings of the 2007 International Computer Music Conference, ICMC 2007*, 2007, pp. 312–319.

[124] Sebastian Böck, Florian Krebs, and Markus Schedl, "Evaluating the Online Capabilities of Onset Detection Methods." in *Proc. of the 13th Int. Society for Music Information Retrieval Conf.* Porto, Portugal: ISMIR, Oct. 2012, pp. 49–54. [Online]. Available: https://doi.org/10.5281/zenodo.1416036

[125] Sebastian Böck, Andreas Arzt, Florian Krebs, and Markus Schedl, "Online real-time onset detection with recurrent neural networks," in *Proceedings of the 15th International Conference on Digital Audio Effects (DAFx-12), York, UK*, 2012.

[126] Luca Vignati, Stefano Zambon, and Luca Turchet, "A comparison of real-time Linux-based architectures for embedded musical applications," *Journal of the Audio Engineering Society*, vol. 70, no. 1/2, pp. 83–93, 2022.

[127] Ross Bencina, "Interfacing real-time audio and file I/O," in *Proceedings of the of the Australasian Computer Music Conference (ACMC)*, 2014, pp. 21–28.

[128] Paul M Brossier, "Automatic annotation of musical audio for interactive applications," Ph.D. dissertation, Centre for Digital Music, Queen Mary University of London, London, UK, 2006.

[129] Siddharth Sigtia, Emmanouil Benetos, and Simon Dixon, "An End-to-End Neural Network for Polyphonic Piano Music Transcription," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 24, no. 5, pp. 927–939, 2016.

[130] Luca Turchet, Johan Pauwels, Carlo Fischione, and György Fazekas, "Cloud-Smart Musical Instrument Interactions: Querying a Large Music Collection with a Smart Guitar," *ACM Transactions on the Internet Things*, vol. 1, no. 3, jun 2020. [Online]. Available: https://doi.org/10.1145/3377881

[131] Clemens Wegener, Sebastian Stang, and Max Neupert, "FPGA-accelerated real-time audio in pure data," in *Proceedings of the International Conference in Sound and Music Computing, SMC-22*, 2022.

[132] Domenico Stefani and Luca Turchet, "Demo of the TimbreID-VST Plugin for Embedded Real-Time Classification of Individual Musical Instruments Timbres," in *Proceedings of the 27th Conference of Open Innovations Association (FRUCT)*, vol. 2, 2020, pp. 412–413.

[133] Joseph Turian, Jordie Shier, Humair Raj Khan, Bhiksha Raj, Björn W Schuller, Christian J Steinmetz, Colin Malloy, George Tzanetakis, Gissel Velarde, Kirk McNally *et al.*, "Hear 2021: Holistic evaluation of audio representations," *arXiv preprint arXiv:2203.03022*, 2022.

[134] Siddharth Sigtia and Simon Dixon, "Improved music feature learning with deep neural networks," *ICASSP, IEEE Int. Conf. on Acoustics, Speech and Signal Processing - Proceedings*, pp. 6959–6963, 2014.

[135] Domenico Stefani and Luca Turchet, "Bio-Inspired Optimization of Parametric Onset Detectors," in *Proceedings of the 24th International Conference on Digital Audio Effects (DAFx20in21)*, vol. 2, Sept. 2021, pp. 268–275.

[136] Luca Turchet, A. McPherson, and M. Barthet, "Real-time hit classification in a Smart Cajón," *Frontiers in ICT*, vol. 5, no. 16, 2018. [Online]. Available: https://doi.org/10.3389/fict.2018.00016

[137] Jan Schlüter and Sebastian Böck, "Musical onset detection with convolutional neural networks," in *Proceedings of the 6th international workshop on machine learning and music (MML), Prague, Czech Republic*, 2013, pp. 79–82.

[138] Axel Roebel, Céline Jacques, and Achille Aknin, "MIREX 2018: Training CNN onset detectors with artificially augmented datasets," accompanying abstract for two submissions to MIREX2018-Audio Onset Detection (AR3, AR4), https://www.music-ir.org/mirex/abstracts/2018/AR4.pdf.

[139] Luca Turchet, "Hard real time onset detection for percussive sounds," in *Proceedings of the 21st International Conference on Digital Audio Effects (DAFx-18)*, 2018, pp. 349–356.

[140] Paul M. Brossier, "Aubio, a library for audio labelling," accessed September 23, 2023, http://aubio.piem.org.

[141] Leonardo Gabrielli, Francesco Piazza, and Stefano Squartini, "Adaptive linear prediction filtering in dwt domain for real-time musical onset detection," *EURASIP Journal on Advances in Signal Processing*, vol. 2011, pp. 1–10, 2011. [Online]. Available: https://doi.org/10.1155/2011/650204

[142] Paul Masri, "Computer modeling of Sound for Transformation and Synthesis of Musical Signal," Ph.D. dissertation, University of Bristol, UK, 1996.

[143] Chris Duxbury, Juan Pablo Bello, Mike Davies, and Mark Sandler, "Complex domain onset detection for musical signals," in *Proceedings of the 6th International Conference on Digital Audio Effects (DAFx03)*, vol. 1, Sep. 2003, pp. 6–9.

[144] Juan Pablo Bello and Mark Sandler, "Phase-based note onset detection for music signals," *2003 IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003. Proceedings. (ICASSP '03).*, vol. 5, pp. V–441, 2003.

[145] Jonathan Foote and Shingo Uchihashi, "The beat spectrum: a new approach to rhythm analysis," *IEEE International Conference on Multimedia and Expo (ICME)*, pp. 881–884, 2001.

[146] Stephen Hainsworth and Malcolm D Macleod, "Onset Detection in Musical Audio Signals," in *Proceedings of the International Computer Music Conference (ICMC)*, 2003.

[147] Simon Dixon, "Onset detection revisited," in *Proceedings of the 9th International Conference on Digital Audio Effects (DAFx-06)*, vol. 120, 2006, pp. 133–137.

[148] Dmitry Bogdanov, Nicolas Wack, Emilia Gómez Gutiérrez, Sankalp Gulati, Herrera Boyer, Oscar Mayor, Gerard Roma Trepat, Justin Salamon, José Ricardo Zapata González, and Xavier Serra, "ESSENTIA: an Audio Analysis Library for Music Information Retrieval," in *International Society for Music Information Retrieval Conference (ISMIR'13)*, Curitiba, Brazil, 04/11/2013 2013, pp. 493–498. [Online]. Available: http://hdl.handle.net/10230/32252

[149] Brian McFee, Colin Raffel, Dawen Liang, Daniel P Ellis, Matt McVicar, Eric Battenberg, and Oriol Nieto, "librosa: Audio and music signal analysis in python," in *Proceedings of the 14th Python in Science conference (SciPy)*, vol. 8, 2015, pp. 18–25.

[150] Sebastian Böck, Filip Korzeniowski, Jan Schlüter, Florian Krebs, and Gerhard Widmer, "madmom: A New Python Audio and Music Signal Processing Library," *Proceedings of the 24th ACM International Conference on Multimedia*, 2016.

[151] Sebastian Böck and Gerhard Widmer, "Maximum filter vibrato suppression for onset detection," in *Proceedings of the 16th International Conference on Digital Audio Effects (DAFx-13). Maynooth, Ireland*, vol. 7, 2013.

[152] Kamer Ali Yuksel, Aytul Ercil, and Batuhan Bozkurt, "Digital sound synthesis via parallel evolutionary optimization," in *2012 20th Signal Processing and Communications Applications Conference (SIU)*, 2012, pp. 1–4.

[153] Jônatas Manzolli, Adolfo Maia Jr, Jose Fornari, and Furio Damiani, "The Evolutionary Sound Synthesis Method," in *Proceedings of the 9th ACM International Conference on Multimedia*, ser. MULTIMEDIA '01. New York, NY, USA: Association for Computing Machinery, 2001, pp. 585–587. [Online]. Available: https://doi.org/10.1145/500141.500248

[154] Ricardo A Garcia, "Growing sound synthesizers using evolutionary methods," in *Proceedings of the Workshop on Artificial Life Models for Musical Applications (ALMMA)*, 2001, pp. 99–107.

[155] Colin G Johnson, "Exploring sound-space with interactive genetic algorithms," *Leonardo*, vol. 36, no. 1, pp. 51–54, 2003.

[156] Artemis Moroni, Jônatas Manzolli, Fernando Von Zuben, and Ricardo Gudwin, "Vox populi: An interactive evolutionary system for algorithmic music composition," *Leonardo Music Journal*, pp. 49–54, 2000.

[157] Marco Scirea, Julian Togelius, Peter Eklund, and Sebastian Risi, "Metacompose: A compositional evolutionary music composer," in *International Confer-*

ence on Computational Intelligence in Music, Sound, Art and Design. Springer, 2016, pp. 202–217.

[158] Viriato M Marques, Cecília Reis, and JA Tenreiro Machado, "Interactive evolutionary computation in music," in *2010 IEEE International Conference on Systems, Man and Cybernetics*. IEEE, 2010, pp. 3501–3507.

[159] Nao Tokui and Hitoshi Iba, "Music composition with interactive evolutionary computation," in *Proceedings of the 3rd International Conference on Generative Art*, vol. 17, 2000, pp. 215–226.

[160] Igor Vatolkin, Mike Preuß, Günter Rudolph, Markus Eichhoff, and Claus Weihs, "Multi-objective evolutionary feature selection for instrument recognition in polyphonic audio mixtures," *Soft Computing*, vol. 16, no. 12, pp. 2027–2047, 2012.

[161] Paul. Faragó, C. Faragó, Sorin Hintea, and M. Cîrlugea, "An Evolutionary Multi-objective Optimization Approach to Design the Sound Processor of a Hearing Aid," in *International Conference on Advancements of Medicine and Health Care through Technology*. Springer International Publishing, 2014, pp. 181–186.

[162] Giovanni Pepe, Leonardo Gabrielli, Stefano Squartini, and Luca Cattani, "Evolutionary tuning of filters coefficients for binaural audio equalization," *Applied Acoustics*, vol. 163, p. 107204, 2020.

[163] Usman Rashid, Imran Khan Niazi, Nada Signal, Dario Farina, and Denise Taylor, "Optimal automatic detection of muscle activation intervals," *Journal of Electromyography and Kinesiology*, vol. 48, pp. 103–111, 2019.

[164] Mateusz Magda, Antonio Martinez-Alvarez, and Sergio Cuenca-Asensi, "MOOGA Parameter Optimization for Onset Detection in EMG Signals," in *New Trends in Image Analysis and Processing – ICIAP 2017*. Springer International Publishing, 2017, pp. 171–180.

[165] Aaron Garrett, "inspyred: Bio-inspired Algorithms in Python," https://pypi.python.org/pypi/inspyred (accessed March 23, 2021).

[166] Chris Cannam, Christian Landone, and Mark Sandler, "Sonic Visualiser: An Open Source Application for Viewing, Analysing, and Annotating Music Audio Files," in *Proceedings of the ACM Multimedia 2010 International Conference*, Firenze, Italy, October 2010, pp. 1467–1468.

[167] Konstantinos P Ferentinos, Konstantinos G Arvanitis, and Nick Sigrimis, "Heuristic optimization methods for motion planning of autonomous agricultural vehicles," *Journal of Global Optimization*, vol. 23, no. 2, pp. 155–170, Jun 2002. [Online]. Available: https://doi.org/10.1023/A:1015527207828

[168] Enrique Alba, Christian Blum, Pedro Asasi, Coromoto Leon, and Juan Antonio Gomez, *Optimization techniques for solving complex problems*. John Wiley & Sons, 2009, vol. 76.

[169] Anne Brindle, "Genetic algorithms for function optimization," Ph.D. dissertation, University of Alberta, 1980.

[170] Zbigniew Michalewicz and Jarosław Arabas, "Genetic algorithms for the 0/1 knapsack problem," in *International Symposium on Methodologies for Intelligent Systems*. Springer, 1994, pp. 134–143.

[171] Kusum Deep and Manoj Thakur, "A new crossover operator for real coded genetic algorithms," *Applied Mathematics and Computation*, vol. 188, no. 1, pp. 895–911, 2007.

[172] John W Tukey, *Exploratory data analysis*. Reading, Mass., 1977, vol. 2.

[173] Keunwoo Choi, George Fazekas, and Mark Sandler, "Automatic Tagging Using Deep Convolutional Neural Networks," in *Proceedings of the 17th International Society for Music Information Retrieval Conference (ISMIR)*, August 2016, pp. 805–811.

[174] Juan S Gómez, Jakob Abeßer, and Estefanía Cano, "Jazz Solo Instrument Classification with Convolutional Neural Networks, Source Separation, and Transfer Learning," in *Proceedings of the 19th International Society for Music Information Retrieval Conference, (ISMIR)*, 2018, pp. 577–584.

[175] Eduardo Meneses, Johnty Wang, Sergio Freire, and Marcelo Wanderley, "A Comparison of Open-Source Linux Frameworks for an Augmented Musical Instrument Implementation," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, Marcelo Queiroz and Anna Xambó Sedó, Eds. Porto Alegre, Brazil: UFRGS, Jun. 2019, pp. 222–227.

[176] Nicholas D. Lane, Sourav Bhattacharya, Akhil Mathur, Petko Georgiev, Claudio Forlivesi, and Fahim Kawsar, "Squeezing Deep Learning into Mobile and Embedded Devices," *IEEE Pervasive Computing*, vol. 16, no. 3, pp. 82–88, 2017.

[177] Leonardo Gabrielli and Luca Turchet, "Towards a sustainable internet of sounds," in *Proceedings of the 17th International Audio Mostly Conference*, ser. AM '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 231–238. [Online]. Available: https://doi.org/10.1145/3561212.3561246

[178] Brian Whitman, Gary Flake, and Steve Lawrence, "Artist detection in music with Minnowmatch," in *Neural Networks for Signal Processing XI: Proceedings of the 2001 IEEE Signal Processing Society Workshop (IEEE Cat. No.01TH8584)*, 2001, pp. 559–568.

[179] Tim Jago, "The role of the jazz guitarist in adapting to the jazz trio, the jazz quartet, and the jazz quintet," Ph.D. dissertation, University of Miami, 2015.

[180] Domenico Stefani and Luca Turchet, "Demo: a Guide to Real-Time Embedded Deep Learning Deployment for Elk Audio OS," in *International Symposium on the Internet of Sounds (Accepted Demo)*, Oct. 2023.

[181] Jesse Engel, Lamtharn (Hanoi) Hantrakul, Chenjie Gu, and Adam Roberts, "DDSP: Differentiable digital signal processing," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=B1x1ma4tDr

[182] Antoine Caillon and Philippe Esling, "RAVE: A variational autoencoder for fast and high-quality neural audio synthesis," *CoRR*, vol. abs/2111.05011, 2021. [Online]. Available: https://arxiv.org/abs/2111.05011

[183] Alec Wright, Eero-Pekka Damskägg, and Vesa Välimäki, "Real-time black-box modelling with recurrent neural networks," in *22nd International Conference on digital audio effects (DAFx-19)*, 2019, pp. 1–8.

[184] Luca Turchet, Carlo Fischione, Georg Essl, Damián Keller, and Mathieu Barthet, "Internet of Musical Things: Vision and Challenges," *IEEE Access*, vol. 6, pp. 61 994–62 017, 2018.

[185] Luca Turchet and Francesco Antoniazzi, "Semantic Web of Musical Things: Achieving interoperability in the Internet of Musical Things," *Journal of Web Semantics*, vol. 75, p. 100758, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1570826822000427

[186] Teresa Pelinski, Rodrigo Diaz, Adán L Benito Temprano, and Andrew McPherson, "Pipeline for recording datasets and running neural networks on the Bela embedded hardware platform," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, 2023.

[187] Keith Bloemer, "GuitarML-NeuralPi," https://github.com/GuitarML/NeuralPi, 2021.

[188] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[189] Marco Comunità, Christian J. Steinmetz, Huy Phan, and Joshua D. Reiss, "Modelling Black-Box Audio Effects with Time-Varying Feature Modulation," in *ICASSP 2023 - 2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2023, pp. 1–5.

[190] Alec Wright and Vesa Välimäki, "Neural modeling of phaser and flanging effects," *Journal of the Audio Engineering Society*, vol. 69, no. 7/8, pp. 517–529, 2021.

[191] William Brent, "A perceptually based onset detector for real-time and offline audio parsing," in *Proceedings of the 2011 International Computer Music Conference, ICMC 2011, Huddersfield, UK, July 31*

- *August 5, 2011*. Michigan Publishing, 2011. [Online]. Available: https://hdl.handle.net/2027/spo.bbp2372.2011.056

[192] Luca Turchet and Johan Pauwels, "Music emotion recognition: Intention of composers-performers versus perception of musicians, non-musicians, and listening machines," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 30, pp. 305–316, 2022.

Appendix A

Additional details

A.1 Aubio onset methods

The list of onset functions in version 0.4.9 of Aubio and the relative descriptions (from Aubio's manual) are shown in Table A.1.

Table A.1: *List of onset functions available in version 0.4.9 of Aubio, from Aubio's manual:* https://aubio.org/manual/latest/cli.html#aubioonset.

aubioonset argument	**Description (Aubio's manual)**
`default`	Default distance, currently hfc
`energy`	Energy based distance: this function calculates the local energy of the input spectral frame.
`hfc`	High-Frequency content: this method computes the High Frequency Content (HFC) of the input spectral frame. The resulting function is efficient at detecting percussive onsets [142].
`complex`	Complex domain onset detection function: this function uses information both in frequency and in phase to determine changes in the spectral content that might correspond to musical onsets. It is best suited for complex signals such as polyphonic recordings [143]
`phase`	Phase based onset detection function: this function uses information both in frequency and in phase to determine changes in the spectral content that might correspond to musical onsets. It is best suited for complex signals such as polyphonic recordings [144].
`specdiff`	Spectral difference onset detection function [145]
`kl`	Kulback-Liebler onset detection function [146]
`mkl`	Modified Kulback-Liebler onset detection function [128]
`specflux`	Spectral flux [147]

A.1.1 HFC Onset Detection Function

From [128]:

$$D_H[n] = \sum_{k=1}^{N} k \left|X_k[n]\right|^2 \tag{A.1}$$

A.1.2 Complex Onset Detection Function

From [128]:

$$D_C[n] = \sum_{k=0}^{N} \left\| \hat{X}_k[n] - X_k[n] \right\|^2 \tag{A.2}$$

A.1.3 Phase Onset Detection Function

From [128]:

$$D_\phi[n] = \sum_{k=0}^{N} \left| \hat{\phi}_k[n] \right| \quad with \quad \hat{\phi}_k[n] = \text{princarg}\left(\frac{\partial^2 \phi_k[n]}{\partial n^2} \right) \tag{A.3}$$

A.1.4 Spectral difference Onset Detection Function

From [128]:

$$D_s[n] = \sum_{k=0}^{N} \left| |X_k[n]|^2 - |X_k[n-1]|^2 \right| \tag{A.4}$$

A.1.5 Kullback-Liebler distance Onset Detection Function

From [128]:

$$D_{\text{kl}}[n] = \sum_{k=0}^{N} |X_k[n]| \log \frac{|X_k[n]|}{|X_k[n-1]|} \tag{A.5}$$

A.1.6 MKL Onset Detection Function

From [128]:

$$D_{\text{mkl}}[n] = \sum_{k=0}^{N} \log\left(1 + \frac{|X_k[n]|}{|X_k[n-1]| + \epsilon} \right), \quad \epsilon = 10^{-6} \tag{A.6}$$

A.1.7 Spectral Flux Onset Detection Function

From [147]:

$$SF(n) = \sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1} H(|X(n,k)| - |X(n-1,k)|) \tag{A.7}$$

Where $H(x) = \frac{x+|x|}{2}$ is the half-wave rectifier function.

A.2 Inspyred operations

A.2.1 Tournament Selection

"This function selects `num_selected` individuals from the population. It selects each one by using random sampling without replacement to pull `tournament_size` individuals and adds the best of the tournament as its selection. If `tournament_size` is greater than the population size, the population size is used instead as the size of the tournament.
Optional keyword arguments in args:

- `num_selected` - the number of individuals to be selected (default 1)
- `tournament_size` - the tournament size (default 2)

" from https://pythonhosted.org/inspyred/reference.html#inspyred.ec.selectors.tournament_selection.

A.2.2 Arithmetic Crossover

"This function performs arithmetic crossover, which is similar to a generalized weighted averaging of the candidate elements. The allele of each parent is weighted by the `ax_alpha` keyword argument, and the allele of the complement parent is weighted by `1 - ax_alpha`. This averaging is only done on the alleles listed in the `ax_points` keyword argument. If this argument is None, then all alleles are used. This means that if this function is used with all default values, then offspring are simple averages of their parents. This function also makes use of the bounder function as specified in the EC's evolve method.
Optional keyword arguments in args:

- `crossover_rate` - the rate at which crossover is performed (default 1.0)
- `ax_alpha` - the weight for the averaging (default 0.5)
- `ax_points` - a list of points specifying the alleles to recombine (default None)

" from https://pythonhosted.org/inspyred/reference.html#inspyred.ec.variators.arithmetic_crossover.

A.2.3 Laplace Crossover

"This function performs Laplace crossover, following the implementation specified in [171]. This function also makes use of the bounder function as specified in the EC's evolve method.
Optional keyword arguments in args:

- `crossover_rate` - the rate at which crossover is performed (default 1.0)
- `lx_location` - the location parameter (default 0)
- `lx_scale` - the scale parameter (default 0.5)

In some sense, the `lx_location` and `lx_scale` parameters can be thought of as analogs in a Laplace distribution to the mean and standard deviation of a Gaussian distribution. If `lx_scale` is near zero, offspring will be produced near the parents. If `lx_scale` is farther from zero, offspring will be produced far from the parents."

from https://pythonhosted.org/inspyred/reference.html#inspyred.ec.variators.laplace_crossover

A.2.4 Generational Replacement

"This function performs random replacement, which means that the offspring replace random members of the population, keeping the population size constant. Weak elitism may also be specified through the `num_elites` keyword argument in args. If this is used, the best `num_elites` individuals in the current population are allowed to survive if they are better than the worst `num_elites` offspring.
Optional keyword arguments in args:

- `num_elites` - number of elites to consider (default 0)

" from https://pythonhosted.org/inspyred/reference.html#inspyred.ec.replacers.generational_replacement

A.3 Elk Audio OS - Deep Learning Guide

Figure A.1 presents the shell commands required to properly cross-compile TensorFlow Lite 2.11.0 for Elk Audio OS.

```
git clone https://github.com/tensorflow/tensorflow.git
cd tensorflow
git checkout v2.11.0
mkdir -p build-aarch64 \&& cd build-aarch64

sed -i 's/GIT_TAG v2.0.6/GIT_TAG v2.0.8 #https:\/\/github.com\/
    tensorflow\/tensorflow\/issues\/57617/g' \
     ../tensorflow/tensorflow/lite/tools/cmake/modules/flatbuffers.cmake

unset LD_LIBRARY_PATH
source /opt/elk/0.11.0/environment-setup-cortexa72-elk-linux

cmake ../tensorflow/tensorflow/lite -DTFLITE_ENABLE_XNNPACK=OFF -
    DCMAKE_TOOLCHAIN_FILE=../toolchain.cmake

export CXXFLAGS="-O3 -pipe -ffast-math -feliminate-unused-debug-types -
    funroll-loops"

AR=aarch64-elk-linux-ar make -j$(nproc) CONFIG=Release CFLAGS="-Wno-
    psabi" \
     TARGET_ARCH="-mcpu=cortex-a72 -mtune=cortex-a72"
```

Figure A.1: *Set of Linux shell commands required to properly download the TensorFlow source code for version 2.11.0 and cross-compile TensorFlow Lite for Elk Audio OS and the Raspberry Pi4. The* `sed` *command represents an example of deviation from the theoretical cross-compilation procedure mentioned in Section 7.4.2, as it is required to fix a compilation bug. It does so by changing the version of the sub-dependency Flatbuffers to include from 2.0.6 to 2.0.8. Additionally, the* `toolchain.cmake` *provided in the project's repository sets the value of the* `CMAKE_SYSTEM_PROCESSOR` *variable to* `aarch64` *to address a bug with the sub-dependencies Abseil and Cpuinfo.*