BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

MODULE FOR PRONUNCIATION TRAINING AND FOREIGN LANGUAGE LEARNING

MODUL PRO VÝUKU VÝSLOVNOSTI CIZÍCH JAZYKŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. VLADAN KUDLÁČ**
AUTOR PRÁCE

SUPERVISOR **Ing. IGOR SZŐKE, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2021

Brno University of Technology
Faculty of Information Technology

Department of Computer Graphics and Multimedia (DCGM) Academic year 2020/2021

Master's Thesis Specification

Student:	**Kudláč Vladan, Bc.**
Programme:	Information Technology
Field of study:	Computer Networks
Title:	**Module for Pronunciation Training and Foreign Language Learning**
Category:	Speech and Natural Language Processing

Assignment:

1. Get familiar with basics of pronunciation training and comparison of two audio examples (using DTW). Study basics of implementation for OS Android.
2. Study provided language learning mobile application.
3. Refactor the provided mobile implementation, profile it and propose optimizations with respect to increasing accuracy, and processing speed and decreasing memory footprint (if possible).
4. Discuss achieved results and future work.
5. Create an A2 poster and a short video presenting your work.

Recommended literature:

- M. Muller, Information Retrieval for Music and Motion, Springer-Verlag, 2007.
- According to supervisor's recommendation

Requirements for the semestral defence:

- Items 1, 2 and part of item 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Szőke Igor, Ing., Ph.D.**
Head of Department:	Černocký Jan, doc. Dr. Ing.
Beginning of work:	November 1, 2020
Submission deadline:	May 19, 2021
Approval date:	October 30, 2020

Abstract

The goal of this thesis is to refactor the implementation of speech processing module for mobile application used for teaching pronunciation, profile it and propose optimizations with respect to increasing accuracy, processing speed, and decreasing memory footprint.

Abstrakt

Cílem této práce je vylepšit implementaci modulu pro mobilní aplikace pro výuku výslovnosti, najít místa vhodná pro optimalizaci a provést optimalizaci s cílem zvýšit přesnost, snížit čas zpracování a snížit paměťovou náročnost zpracování.

Keywords

speech processing, profiling, optimisation, DTW, TensorFlow, Android, parallelization

Klíčová slova

zpracování záznamu řeči, profilování, optimalizace, DTW, TensorFlow, Android, paralelizace

Reference

KUDLÁČ, Vladan. *Module for Pronunciation Training and Foreign Language Learning.* Brno, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Igor Szőke, Ph.D.

Rozšířený abstrakt

Při výuce jazyka je výslovnost důležitou součástí. Zatímco slovíčka lze trénovat psaním slov a vět, a následným porovnáním se správnou variantou, výslovnost lze trénovat pouze opakováním a vyslovováním frází a vět. U výslovnosti neexistuje správná a špatná odpověď, obvykle lze porovnat přesnost výslovnosti vůči referenční nahrávce nebo pomocí výslovnostního modelu.

Cílem této práce je optimalizovat modul pro mobilní aplikaci, který umožní získat přesnost výslovnosti. Výstupem této práce je modul, který je schopen rychle ohodnotit výslovnost i na méně výkonných chytrých telefonech a tabletech, který bude fungovat i na pomalém nebo žádném internetovém připojení.

Pro demonstrační účely má modul jednoduché grafické rozhraní, pomocí kterého je možné zkoušet různé případy použití. Rozhraní je implementované v *React Native* a není určeno pro koncové uživatele. Modul by měl být implementován do aplikace, která umožní uživateli přehrát referenční nahrávku, nahrát svůj hlas, vyhodnotit výslovnost a zobrazit uživateli výsledky.

Vedoucí práce poskytl původní výpočetní modul implementovaný v programovacím jazyce Java. Modul byl poskytnut včetně natrénovaného modelu pro extrakci příznaků z řeči, vzorové anotované nahrávky a vypočítaných příznaků pro vzorovou nahrávku. Dále poskytl jednoduché grafické rozhraní implementované pomocí *React Native* frameworku. Původní modul byl nestabilní a obsahoval chyby ve výpočtu. V rámci práce byl modul přepsán, byla přidána možnost automatizovaného spouštění a testování, a díky tomu mohly být nepřesnosti ve výpočtu lokalizovány a opraveny. V uživatelském rozhraní byly opraveny pády a bylo rozšířeno zobrazování aktuálního stavu modulu, zejména při více souběžných operacích.

Zpracování probíhá přímo v zařízení, takže funguje i off-line. Tomu bylo potřeba přizpůsobit použité algoritmy. Zvolený algoritmus je založený na porovnávání příznaků dvou audio nahrávek. Oproti převodu řeči na text a následném porovnání vyžaduje tento přístup menší neuronovou síť, tedy menší výpočetní nároky. Tento přístup je také více zaměřený na intonaci a správnou výslovnost.

Nejprve je nahrán hlas uživatele jako jednokanálové audio (mono), kde jednotlivé vzorky jsou reprezentovány pomocí nekomprimované pulzně kódové modulace (PCM) s bitovou hloubkou 16 bitů. Tento formát je používaný například u formátu WAV (RIFF). U zařízeních s OS Android je jediná garantovaná vzorkovací frekvence 44 100 Hz. Po nahrání hlasu je sníženo vzorkování nahrávky ze 44 100 Hz na 8 000 Hz. Poté jsou z jednotlivých vzorků audio nahrávky vytvořeny překrývající se rámce. Pro tyto rámce jsou vypočteny frekvenční charakteristiky pomocí *diskrétní Fourierovy transformace*. Tyto frekvence jsou poté převedeny a sloučeny do *Mel bank*. *Mel banky* charakterizují úseky nahrávek z hlediska frekvencí, které jsou pro lidský hlas význačné, pomocí 24 charakteristik (bank). Poté probíhá výpočet fonémů. Protože ale nepotřebujeme převádět hlas na textový přepis, využívají se pouze 3 vrstvy z původní neuronové sítě pro výpočet fonémů a výstupem je nejmenší z vrstev (*bottleneck* vrstva), která dokáže rámec popsat pomocí 30 číselných hodnot. Protože potřebujeme porovnat dvě odlišné nahrávky, s odlišnou délkou a s různě rychlými řečníky, potřebujeme části nahrávek zarovnat pomocí *DTW* algoritmu. Výstupem algoritmu je řetězec ve formátu *JSON*, který obsahuje celkový výsledek i úspěšnost pro jednotlivá slova.

Hlavní náplní a největším přínosem této práce je identifikace částí, které lze vylepšit a optimalizovat. Použité techniky lze použít v podobných aplikacích, protože uvedený způsob zpracování řeči je poměrně rozšířený. Nejprve byla aplikace částečně přepsána

z programovacího jazyka *Java* do jazyka *Kotlin*. To umožnilo použití knihovny *Kotlin Coroutines*. Knihovna je určena pro snadné psaní asynchronního kódu a obsahuje další nástroje pro paralelní programování (např. *Kotlin Channels* pro komunikaci mezi vlákny).

Poté jsme se zaměřili na zrychlení načítání modulu a zrychlení zpracování nahrávek při zachování přesnosti. Pro využití nízko úrovňových funkcí, které hardware dnešních telefonu poskytuje, jsme využili vysoko úrovňovou knihovnu *TensorFlow Lite*. Vyzkoušeli jsme různé akcelerátory, ale nakonec se pro použitý model osvědčila CPU varianta s jedním vláknem (použitá neuronová síť není dostatečně složitá, s dostatečnou aritmetickou intenzitou). Výpočet se podařilo dále zrychlit výpočtem více rámců zaráz. Jako optimální se jeví výpočet 16 rámců zaráz, další zvětšování počtu rámců nevede k dalšímu zrychlení. Pro složitější sítě se ukázalo výhodné provádět výpočet na GPU. Naopak *NNAPI* se neukázalo pro tento typ modelu příliš výhodné. Díky *TensorFlow Lite* se podařilo zkrátit dobu načítání neuronové sítě ze 7,6 sekund na 0,8 sekund. Zpracování 10sekundové nahrávky trvá 1 sekundu místo původních 2,7 sekund. U 5s nahrávky se doba zpracování zkrátila z 1,1 sekund na 0,3 sekund.

Jako druhou techniku jsme zvolili paralelizaci. Identifikovali jsme problematické bloky pro paralelizaci a navrhli možné způsoby paralelizace. Implementované paralelní řešení zachovává původní přesnost a přináší zrychlení 45 % oproti časům po nasazení *TensorFlow Lite*.

Poslední věcí je uživatelské rozhraní. Zpracování se může zrychlit, ale pokud je uživatelské rozhraní a vykreslování pomalé, uživatel žádné zrychlení nepocítí. Navíc i po použití výše uvedených optimalizací je pro zpracování stále potřeba 174 ms. Tuto dobu může aplikace skrýt do přechodů mezi stavy nebo do krátkých animací, takže z pohledu uživatele bude zobrazení výsledků okamžité.

Proces optimalizace lze vyhodnotit jako úspěšný. Podařilo se snížit čas zpracovávání 5s záznamu z 928 ms na 174 ms se stejnou přesností. Delší záznam (10 sekund) byl původně zpracován za 2,5 sekundy, na konci práce pouze za 0,5 sekundy. Doba načítání neuronové sítě byla snížena ze 6 sekund na 8 milisekund.

V modulu jsou stále prostory pro zrychlení. V paralelizaci lze použít normalizaci s plovoucím oknem a experimentovat s přesností. DTW lze upravit tak, aby pracovalo pouze s částí stavového prostoru. V grafickém rozhraní stále dochází ke zpomalení, pokud probíhá více operací současně.

V budoucnu je plánováno nasazení ve skutečné aplikaci pro výuku cizích jazyků. Dále je zamýšleno využít tyto techniky na další podobné aplikace, jako například na rozpoznávač řeči.

Module for Pronunciation Training and Foreign Language Learning

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Doc. Igor Szőke. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Vladan Kudláč
May 17, 2021

Acknowledgements

I would like to thanks to the supervisor who was providing valuable advice and hints and who introduced me deeper into the issue of speech processing. Many thanks to my family for their support.

Contents

Chapter 1

Introduction

The goal of this thesis is to refactor the provided implementation of the speech processing module used in the Android mobile application for learning foreign languages. Refactoring should mainly speed up processing, improve code readability and divide the code into separate modules, which should make easier development in the future.

When learning a language, we need to learn new vocabulary, the written form, and learn how to pronounce it. Then compose words into sentences and learn proper intonation and accent. The grammar can be checked by writing the sentence, composing from blocks, or selecting the correct form. But only repeating or reading phrases or words can check the pronunciation. Also, in terms of pronunciation, there is no correct or wrong answer. There is usually a reference recording or pronunciation model to be compared with. Users can get success a rate or success/fail in case of applying a threshold. The output of this thesis will be a library, which will to evaluate the user pronunciation from a recording. The application is intended, among other countries, for the Indian market. So, there is a request for fast speech evaluation even on low-end smartphones or tablets and for low mobile data usage and offline mode. That is the purpose of refactorization and finding ways, how to speed up the evaluation process.

My supervisor provided me the evaluation module, including a trained model in the binary form used for features extraction, one example of annotated recording, reference features of recording, and a simple *React Native* wrapper (GUI). The original module was created by porting JavaScript implementation used in the company *ReplayWell*[1] into Java. The module was able to evaluate the speech but it was crashing when calling the API in the wrong order or multiple times. Beyond the scope of this work, I fixed the crashes and changed error handling and logging. The evaluation results seemed inaccurate, but it was not possible to test the implementation. Before experimenting with different models and optimizations, I added loading and saving user speech into the WAV file. Thus, the application can be run multiple times with the same input data, which generates the same output results. So, the output results can be compared. The application now includes a debug mode which dumps output, as a matrix of each processing phase, into the file. With the debug mode, I was able to locate and fix the source of evaluation inaccuracy. Figures 4.1 and 4.2 display the difference between the original solution and the refactored solution. Refactorizations with impact on speed or accuracy are included in chapter five. The user interface was simplified into a more compact layout. Messages from the processing module are parsed to show multiple progress bars. All known bugs, that were causing the crashes,

[1] ReplayWell - systems for speech and video processing, https://www.replaywell.com

were fixed. The added profiling mode can automatically go through an initializing phase to the processing phases based on states received from the computing module.

This thesis is divided into six chapters. In the second chapter, I will describe the functionality and usage of this module. In the third chapter, I will explain the used algorithm for speech processing, which explains the background of algorithm complexity. In chapters four and five, the process of refactorization is described. The refactorization process is divided into two parts. Chapter four is about changes improving module architecture and chapter five is an experimental part, which describes proposed optimizations and the process of their implementation.

Chapter 2

Mobile application

In this chapter, I will describe application architecture, communication between *React Native* UI and native *Speech Engine* module, the functionality of the module, typical usage of the API, options available in configuration files, and file structure of tasks and the module itself. The module lacked documentation, so I will reference to API or schemas attached in Appendices.

2.1 React Native UI

The *SpeechEngine* module will be used in an application already implemented in *React Native*[1]. The module does not have its user interface. The *SpeechEngine* module is wrapped in a simple *React Native* user interface to demonstrate module functionalities and test them properly. Buttons are calling corresponding API functions (see appendix A) and state and progress indicators displaying feedback from the module, see figure 2.1. This interface is not intended for users as wrong combinations of call sequences will cause errors The UI of a real application will be like the UI of the module implemented in JavaScript with the user interface written in HTML (see figure 2.2).

Building *React Native* app requires *Node.js*[2] (at least version 12 for *React Native* version 0.64) and *npm*[3]. JSX (JavaScript extended by React Native) describes UI and its behaviour. JavaScript engine renders native UI and handles OS events using *React Native Bridge*. Modules implemented in *Kotlin* or *Java* are wrapped into *React Package* using *Native Modules*. Communication between the JavaScript engine and the native module is by emitting and listening to asynchronous calls.

The project can be deployed in development mode to mobile using *Android Studio*. *React Native* part is distributed using the *Metro* server[4]. The app is released for users as a regular *APK* (Android application package) file with its requirements for minimal and targeting Android version directed by application developers. The minimal required API level is currently 23 (Android Marshmallow).

[1] React Native - a framework for building Android and iOS applications using *React*, https://reactnative.dev

[2] Node.js - JavaScript runtime, build on Google's V8 engine, https://nodejs.org

[3] npm - package manager for Node.js, https://www.npmjs.com.

[4] Metro - JavaScript bundler for *React Native*, https://facebook.github.io/metro

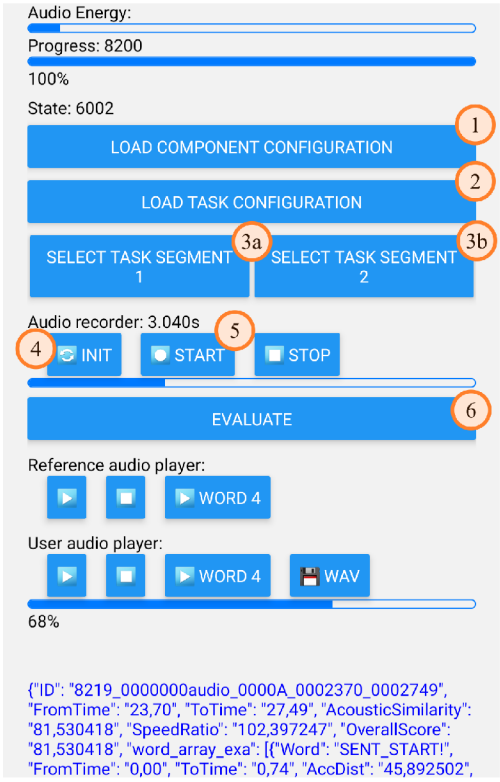

Figure 2.1: Graphical user interface (GUI) of React Native wrapper. The buttons correspond to the functions from the API. Numbers in the circles indicate the order in which the buttons are used (numbers are not part of GUI). There are two configured tasks (buttons **3a** and **3b**) to choose from.

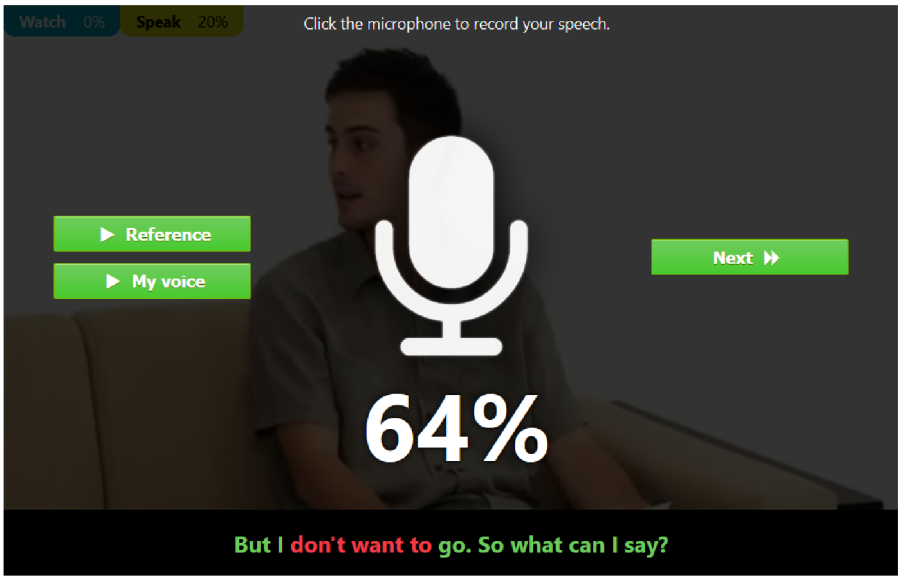

Figure 2.2: Screenshot of ReplayWell user interface demonstration written in HTML and JavaScript. Available at: https://www.replaywell.com/glplayer/demo/.

2.2 Speech Engine module

The speech engine module provides functionalities for playing reference recording, recording user's voice and evaluating the accuracy of pronunciation. This part is implemented in Java and Kotlin, and unlike to React Native part, it is targeted only to Android. Creating a native engine module for iOS is not covered by this thesis.

2.3 Program functions

The following use case diagram (figure 2.3) represents functions that can be called from the application. There are functions to initialize the speech engine and to control the audio recorder or player. There is also debugging function allowing to load user voice from WAV. Along with other options, it is available through configuration files `SpeechEngineConfig.json` and `TaskConfig.json` (see appendix B).

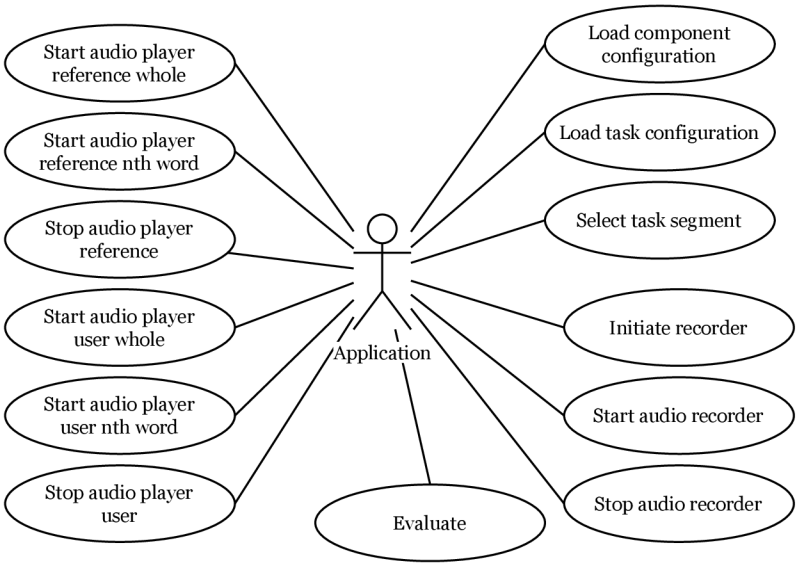

Figure 2.3: Use case diagram representing functions that can be called from the application.

2.4 Use case

In figure 2.4 can be seen an example of the typical use case of this module. A user launches the application and the component configuration is loaded in the background. If the user selects a *lesson* that includes a *task* with pronunciation, the application loads *task* configuration in the background.

As a lesson, we can imagine for example "exercise 3.12" from some book. This lesson (exercise) would be focused on job interviews. The lesson would include multiple tasks (lines "a)" to "f)" in the book). The first task would be to correctly write "interview" word, the next task would be to repeat a phrase "I perform well under pressure".

The user can solve tasks listed before the task with pronunciation, then he has to wait, till previous configurations are loaded. If the configurations are loaded, then the task segment is selected on the foreground, and the user has to wait (but it is loaded

almost immediately). Then the application initiates a recorder in the background. User can meanwhile play the whole reference recording or only part using the play/stop button. If the task allows showing transcription, then the user can play a specific word. Then the user hit the record button and the application starts the audio recorder. The recording is stopped by the user or when the time runs out. The application immediately fires processing in the background. During that time, the user can listen to his recording. The application shows the result when it is available and the user can try it again, proceed to the next task, or finish the lesson and select the next lesson.

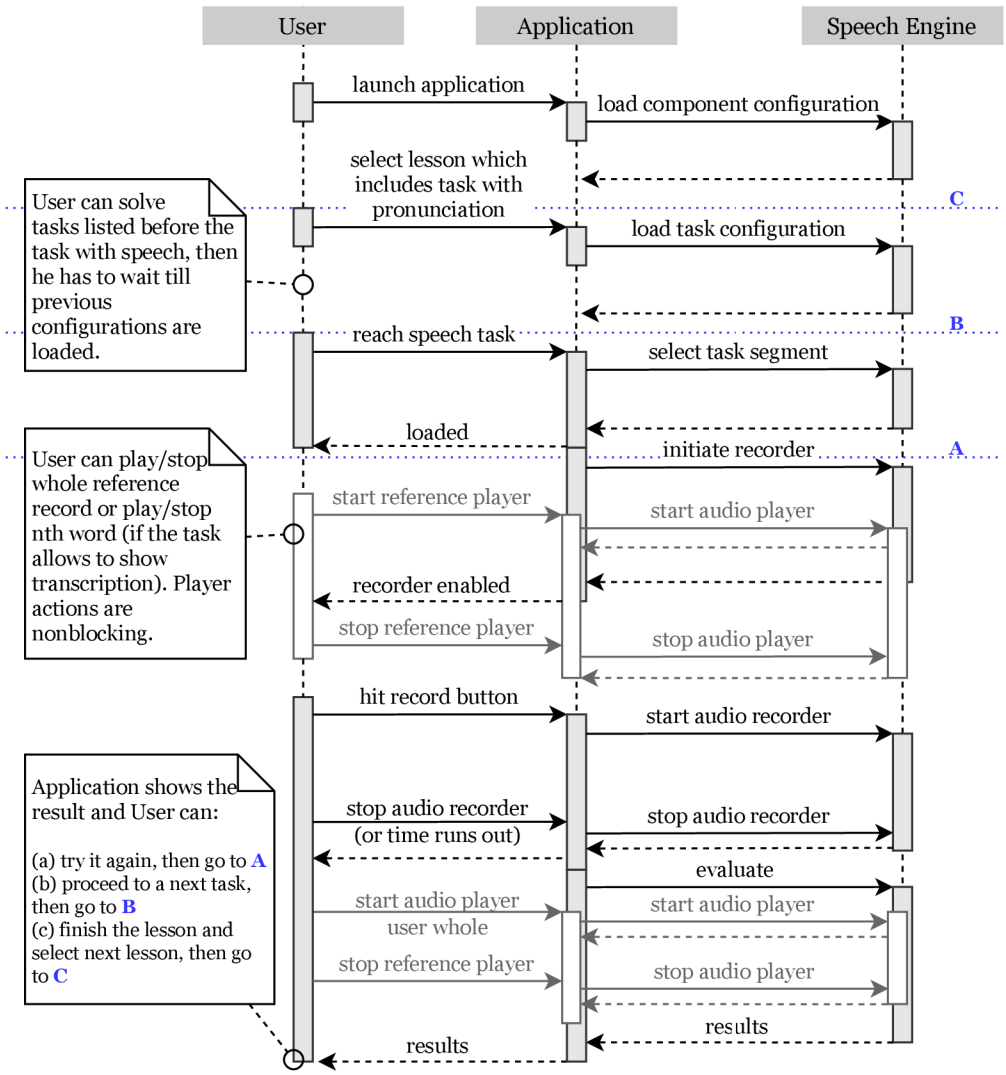

Figure 2.4: Example of the typical use case of the module. Audio players actions are non-blocking (white activation blocks and shadow arrows in the picture). Points **A**, **B**, **C** are time points referenced in the bottom text description on the left.

Chapter 3

Speech processing

The application needs to extract speech features from a recording. The widely used term *speech recognition* is inaccurate in this case as the goal of the application is not to obtain a transcript of spoken words. The processing algorithm only needs to compare features of user recordings with reference recording features.

There are more possible approaches to the issue of evaluating the user's speech. The first possibility is to recognize spoken words and compare a text transcript with a reference text transcript. The success rate would be the probabilities of these words from the recognition module. The second possibility is to compare the characteristics of audio recordings. The first option is resistant to common user errors, like word repetition, skipping a word, wrong order of words, or saying a different sentence. But this method is not focused on checking intonation, speed or pronouncing accuracy well. The second one considers pronunciation, but it cannot get over word repetition, wrong word order, totally different sentence, or over recordings (user or reference) with background noise. This app accomplishes the goal using the second way (comparing recordings) because it focuses on intonation and pronunciation. The first option would require a bigger model, thus more powerful devices, but we need fast, off-line, on-device processing (even on low-end smartphones).

The input of the processing algorithm is audio recorded from a microphone or loaded from a WAV file (see figure 3.2). As an output, there is a JSON object containing overall and partial global scores and scores for each word. Figure 3.1 shows the whole process.

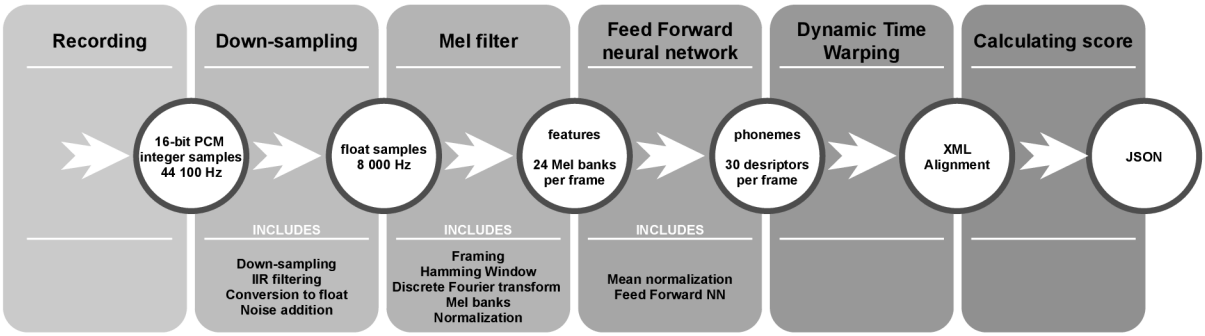

Figure 3.1: Overview of the speech processing pipeline.

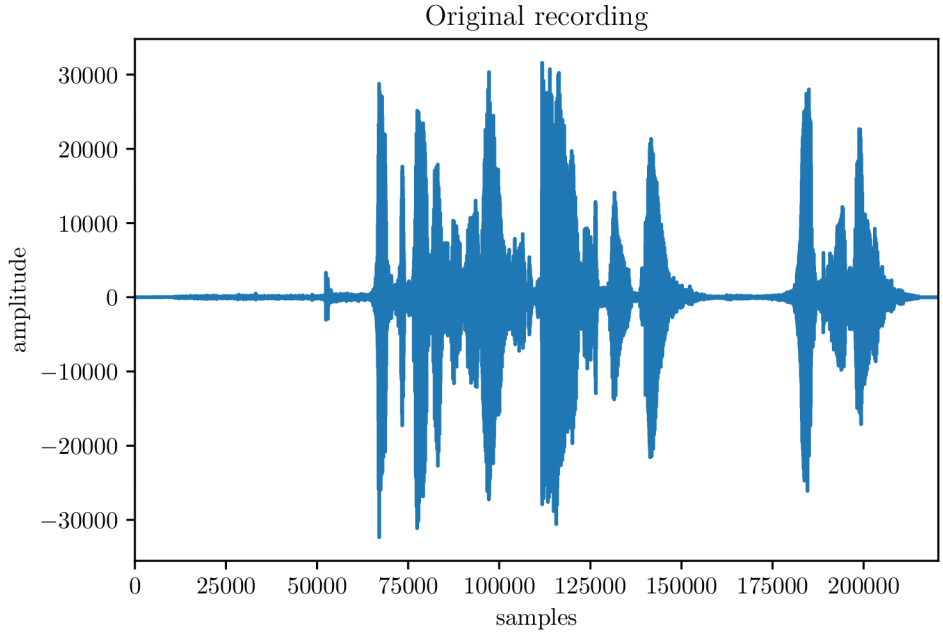

Figure 3.2: Recorded speech (5 s) represented by 220 500 samples in PCM format before the processing.

3.1 Recording

Audio is recorded from a microphone with a sampling frequency of 44 100 Hz. It is the only rate that is guaranteed to work on all devices[1]. The recording is in one channel (mono) with uncompressed *pulse code modulation* (PCM). Each sample is represented by a 16-bit signed integer value. Thus, each sample can hold discrete value in the range -32 768 and 32 767 inclusive. This format is guaranteed to be supported by all devices. The whole recording is stored in *AudioBuffer* (array of shorts). When the recording finished, the array proceeds for further processing. There is also an option to load recording from uncompressed WAV (RIFF) file with 16-bit PCM encoding for testing purposes[2]. Reference audio is stored in this format as well.

The recorder is implemented using *AudioRecord* class. The audio recorder reads chunks from *AudioRecord* using `while` cycle and copies data from *AudioRecord* buffer into large *AudioBuffer*. The recorder stops when a user hits the stop button or when enough samples were recorded (if the recording duration was previously set). The *AudioBuffer* is managed by the app. The app manages *AudioBuffer*. A new larger array of shorts is created when the large *AudioBuffer* gets full. Data from the current *AudioBuffer* are copied into this new and this new one is set as *AudioBuffer*. The output of this step is *AudioBuffer* (array of shorts).

[1] AudioRecord - *Android Developers*, https://developer.android.com/reference/android/media/AudioRecord

[2] Audio File Format Specifications, http://www-mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/WAVE.html

3.2 Down-sampling

The input of this step is *AudioBuffer* (array of shorts). It is the first step of audio processing. 16-bit samples captured 44 100 times per second means a lot of data for further processing, but not every device supports lower frequencies. Thus, the recorded signal is down-sampled to 8 000 Hz. This frequency comes from the provided neural network, but it is possible to retrain the model and use different frequency (e.g., 15 kHz). The usual frequency for voice sampling is from 300 Hz to 3 400 Hz in telephony. The sampling theorem by Shannon says that sampling frequency must be at least two times higher than the highest frequency component. In this case, the highest frequency is 3 400 Hz, thus sampling frequency has to be 6 800 Hz or higher, so 8 kHz for speech is enough to avoid the aliasing effect [3].

The aliasing effect in Shannon theorem is about sampling and reconstructing continuous signals, but aliasing is also a side effect of down-sampling itself. Each input sample goes through an *IIR* (infinite impulse response) filter, which behaves like *LPF* (low-pass filter). Output samples are converted from short integers into float values (this will be important for further steps). Every nth sample is kept, other samples are dropped. If the recorded sample has zero value, it is replaced by a random value from interval $< -1; 1 >$, which adds little noise to the signal. It is essential in cases when the recorder would record only zero values (e.g., due to faulty hardware) as the algorithm of computing Mel banks includes computing logarithm of values. The output of this step is an array of float samples. Down-sampling can be done during recording as **AudioRecorder** produces the samples in smaller parts.

3.3 Mel filter

In this step, the application gets amplitude values in time (array of float samples). But to compare two signals, the algorithm needs frequency characteristics in time and even better features of the signal in time. This step consists of framing input signal into separated frames, applying *Hamming window*, calculating spectral analysis on each frame, and calculating *Mel filter banks* from the analysis.

3.3.1 Framing

Calculating spectral analysis on input signal would return one frequency characteristics for the whole recording. Thus, the algorithm needs to split recording into frames and calculate spectral analysis on each frame. With this approach, the algorithm gets a sequence of spectral analysis in time.

Input samples are divided into 200 samples width frames. That means frames of width 25 ms when using 8 kHz frequency. We need to start a new frame quite often, and at the same time, we need the frames long enough (e.g., 25 ms). Therefore, the frames are overlapping. The frame starts every 80 samples. So, each frame contains 120 samples (15 ms) from the previous frame and 80 new samples (10 ms) (see figure 3.3).

3.3.2 Hamming window

But even with overlapping frames, the algorithm has to deal with values at both edges of the frame. Clipping signal at amplitude peak would cause distortion of spectral analysis. So each value of the frame is multiplied by weight at the corresponding position. This

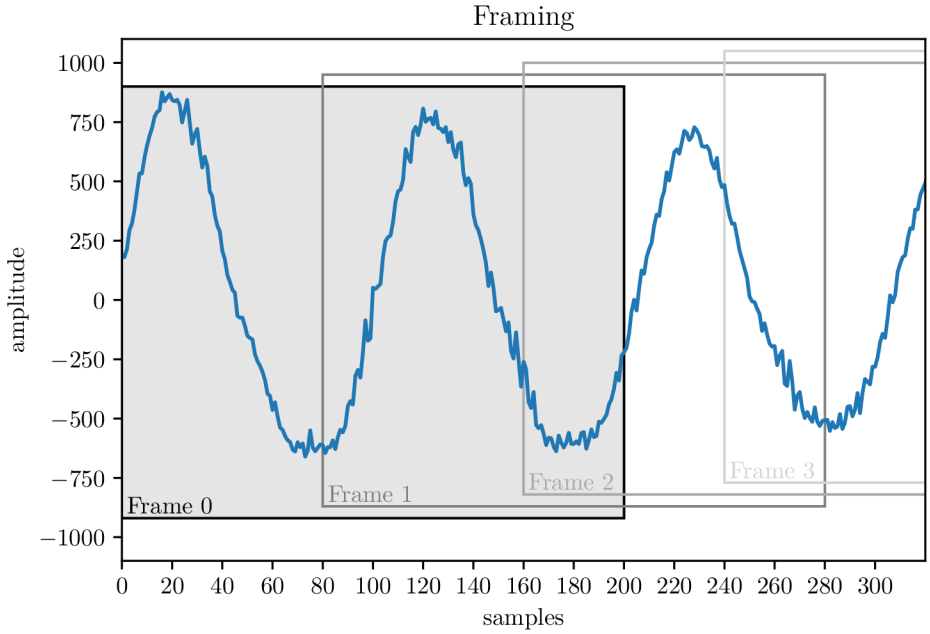

Figure 3.3: Splitting down-sampled signal into overlapping frames.

weight function is called *Hamming window* and it is defined by the following formula:

$$f_h(n) = (0.54 - 0.46 \cos\left(\frac{2\pi n}{width - 1}\right)) \tag{3.1}$$

where:

- $width$ is the width of the frame, in this case $width = 200$
- n is the position of weight in the frame, in this case, an integer value from interval $< 0; 200 >$

The frame has an actual width of 256 float values, last 56 values are padded by zeros (due to FFT). The weight of the first and 200th sample is 0.08 so the values are less important and appended constant zero values do not affect the algorithm at all (see figure 3.5). The hamming window weights are computed only once before processing, they are the same for all frames (see figure 3.4).

3.3.3 Discrete Fourier transform (DFT)

In this part of the step, every 256 discrete samples are converted into a same-length sequence of the discrete spectrum of the signal, see figure 3.6. An original DFT algorithm requires $O(N^2)$ multiplications and additions. The application uses an efficient version of DFT called *1D Fast Fourier transform* (FFT) with $O(N \log N)$ multiplications and additions [3]. This application uses the *JTransforms* library with its *DoubleFFT_1D* class[3], a parallel implementation of split-radix and mixed-radix algorithms optimized for SMP (symmetric multiprocessing) systems.

[3] `DoubleFFT_1D` class documentation, http://incanter.org/docs/parallelcolt/api/edu/emory/mathcs/jtransforms/fft/DoubleFFT_1D.html

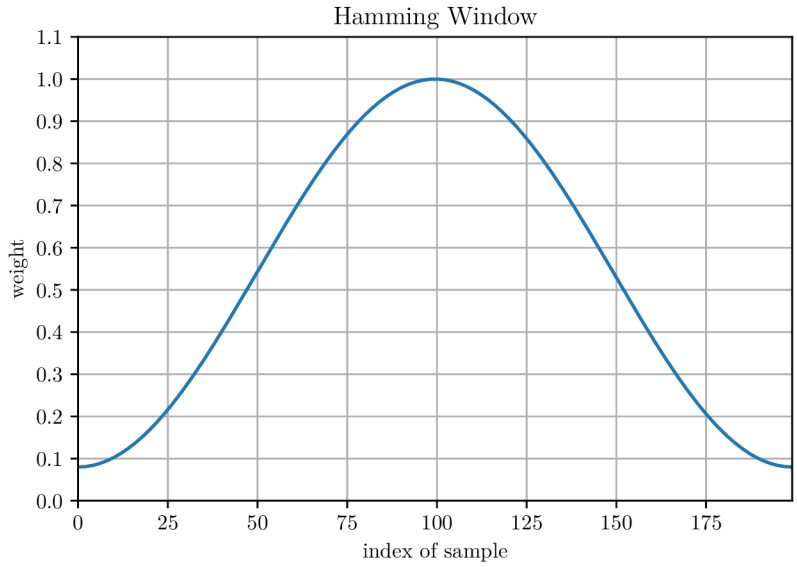

Figure 3.4: Hamming window function, defined on interval $< 0; 199 >$.

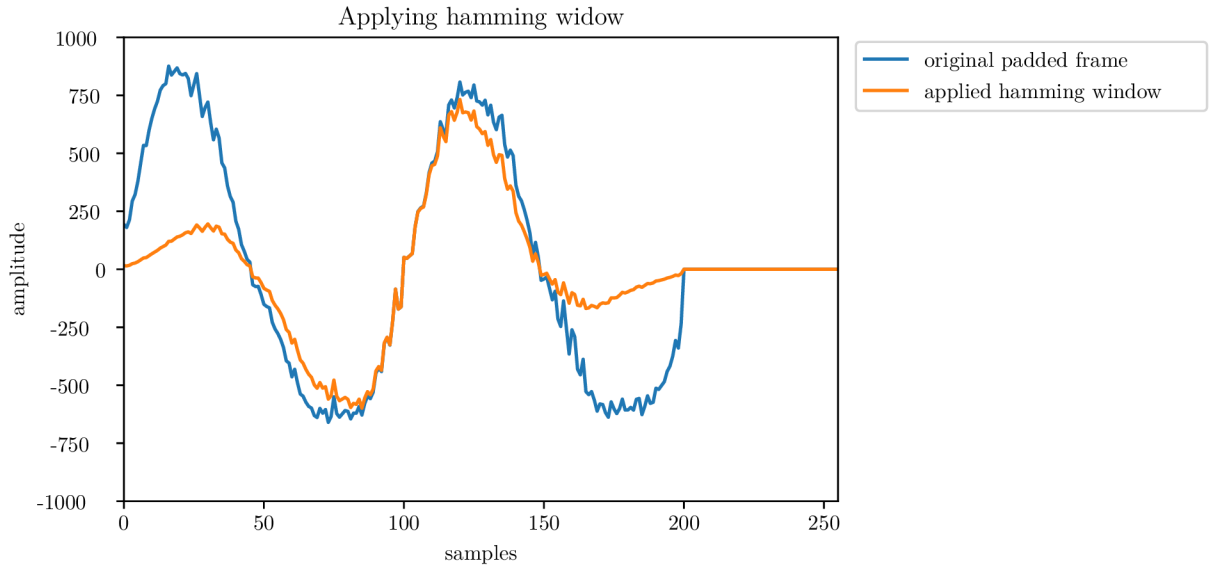

Figure 3.5: First 256 samples width frame before and after applying hamming window.

3.3.4 Mel banks

At this point, the algorithm has each frame represented by 256 complex numbers. Now the algorithm separates the input signal into 24 Mel filter banks. It is not necessary to describe the characteristic of the speech signal by 256 values as human hearing is not sensitive to all frequency intervals equally. And this is how filter banks work. It divides frequencies into 24 overlapping banks taking speech characteristics into account. The application works on each frame by following. All banks are initiated to zero values. Each frequency of spectrum weights on each of output bank, the algorithm goes through all frequencies and multiplies

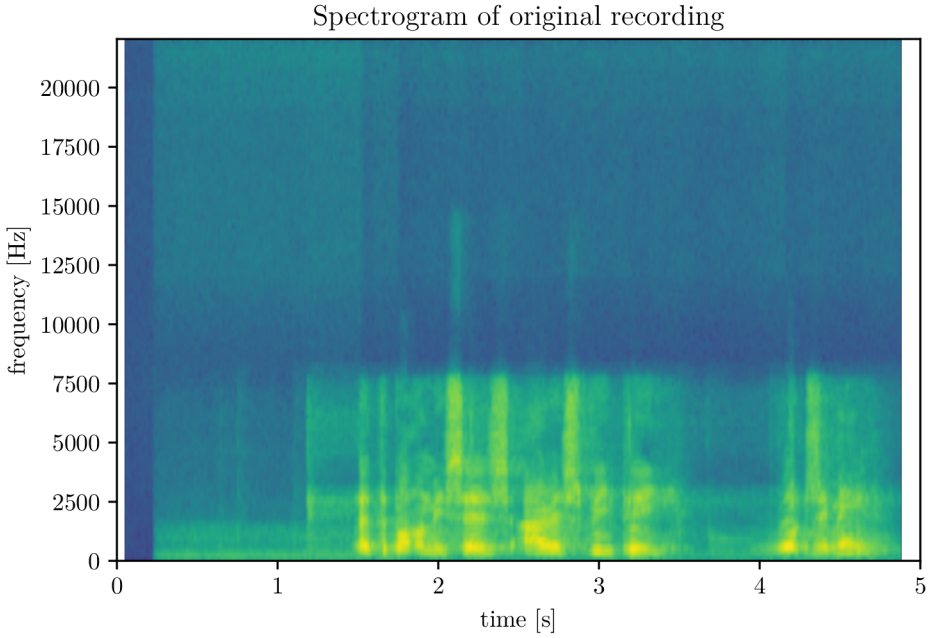

Figure 3.6: Spectrogram of the original recording from figure 3.2. Frequencies from the upper half of the spectrum occurs less frequently.

them by the corresponding vector of weights for that frequency, outcomes are added into the banks. Weights represent the bandpass filter, and its values displays figure 3.7. The output of this phase is a feature matrix. The matrix always has 24 columns (24 filter banks). The number of rows is equal to the number of frames - the number of samples after down-sampling minus length of the frame (200) divided by step (80), rounded up. So, if we have 39 040 samples after down-sampling, the *FeaturesMatrixF* (containing floating-point numbers) would be 24x486 (see figure 3.8).

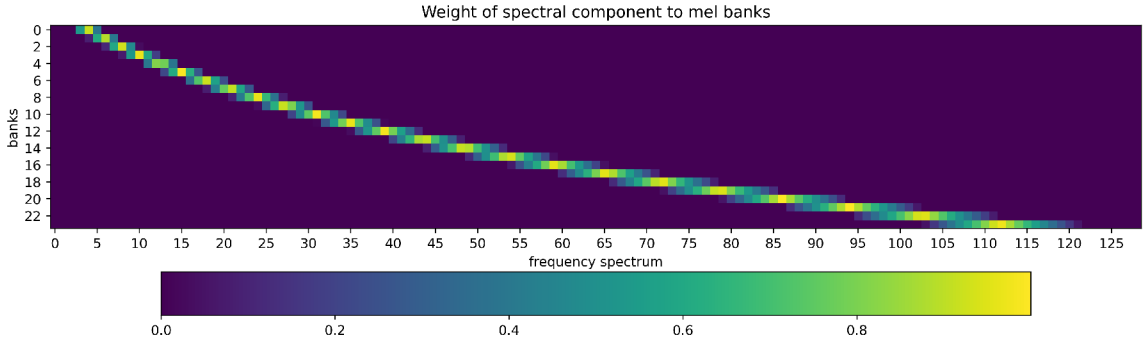

Figure 3.7: Parts of the frequency spectrum with weights on each Mel bank. The upper half of the spectrum has no effect on banks.

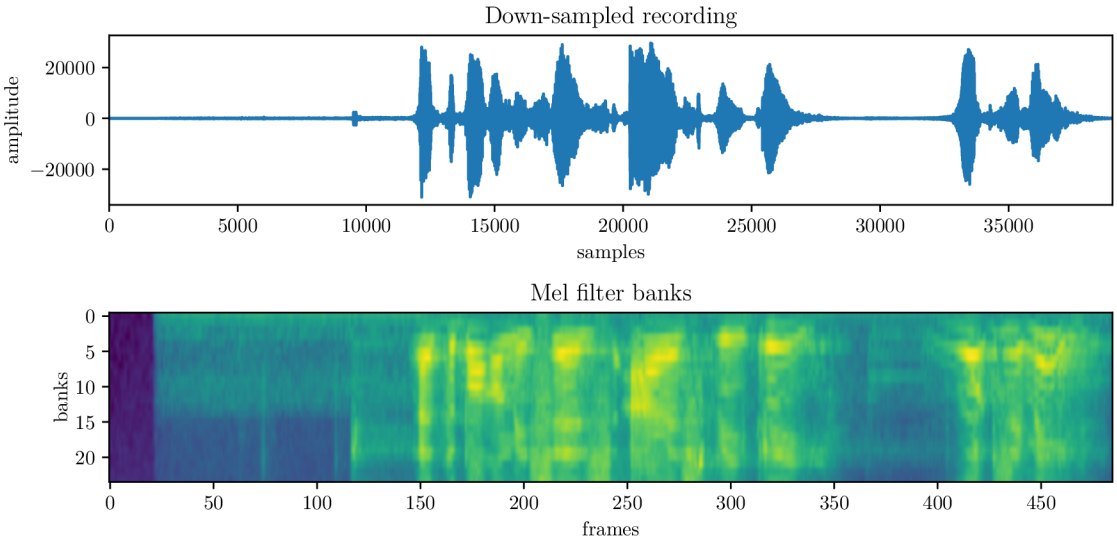

Figure 3.8: Mel banks of all frames from *FeaturesMatrixF* with a silence at the beginning. Note that while in the spectrum only the first half of frequencies were used, Mel banks are used equally.

3.4 Feed forward neural network

From the beginning to this step, the process is the same as the process of speech recognition. At this point, a forward-feed neural network (FFNN) converts spectral analysis into *phonemes* (the smallest units of sounds in a language). Forward-feed neural network is an artificial neural network where connections between nodes (neurons) do not form a cycle. The network groups into the layers, and each neuron has input from the previous layer and output connected to the next layer. The first layer is called the *input layer*, and the last layer is called the *output layer*. Layers between them are called *hidden layers*, see figure 3.9. Each neuron, sometimes called perceptron, has single output defined by the following formula [2]:

$$y = \varphi(\sum_{i=1}^{n} w_i x_i + b) = \varphi(\mathbf{w}^T \mathbf{x} + b) \tag{3.2}$$

where:

- $\mathbf{w}$ is the vector of weights
- $\mathbf{x}$ is the vector of inputs
- b is the bias
- φ is the activation function, it is a *sigmoid* function in this network
- n is number of input connections, in this network $n = 1$

When we look at the whole layer, the input is the same vector for all perceptrons in the layer. But the weight can differ for each perceptron. The whole layer can be described by a similar equation:

$$y = \varphi(\mathbf{w}^T \mathbf{x} + b) \tag{3.3}$$

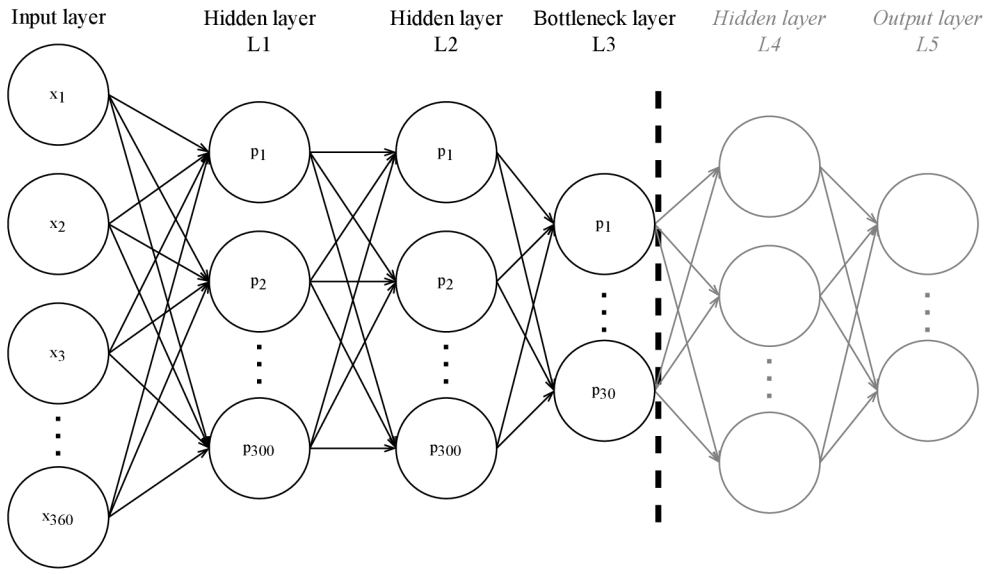

Figure 3.9: The whole feed-forward neural network with 5 layers. Layers to the right of the vertical dashed line are omitted in this project and the *Bottleneck layer* is used as the *Output layer*.

where **x** is still a vector of inputs, but **w** is a matrix of the weights. The number of weight matrix rows corresponds to matrix length and the number of columns to the output vector length (see figure 3.10). So, this feed-forward neural network is nothing different than matrix multiplication, matrix addition and applying a *sigmoid* function to each value of the output vector. Thinking of neural networks this way will be useful in connection to the optimizations and usage of TensorFlow in the later chapters.

This application does not require phonemes, so it uses only the first three layers of FFNN and uses the bottleneck layer as the output layer (the left part of the vertical dashed line in figure 3.9). The bottleneck layer output is only 30 floating-point values width vector [5].

The input of this FFNN is a vector of the width of 360 floating-point values. FFNN processes chunks of 15 frames Mel banks (each frame is defined by 24 Mel banks). This method works with the context of 7 frames before the actual frame and seven frames after it. FFNN always computes 15 frames, so each frame computes 15 times. Using a buffer and computing each frame only once can improve that. The output of this phase is different *FeaturesMatrixF*. It has 30 columns and the width depends on the number of frames.

This part is the most expensive part of processing in terms of computing time.

3.4.1 Mean normalization

Before FFNN, the Mel bank values have to be normalized (to remove channel effects). Normalization is usually done by subtracting mean values from input and then by dividing by the variance. In this case, mean and variance values are precomputed and stored as a vector of 360 values. Variance values are stored as the multiplicative inverse of values; mean values are stored as the additive inverse of values. We described the neural network as a sequence of matrix addition and multiplication. Then the process of normalization can handle the first hidden layer of the neural network as well. Before applying weights of the

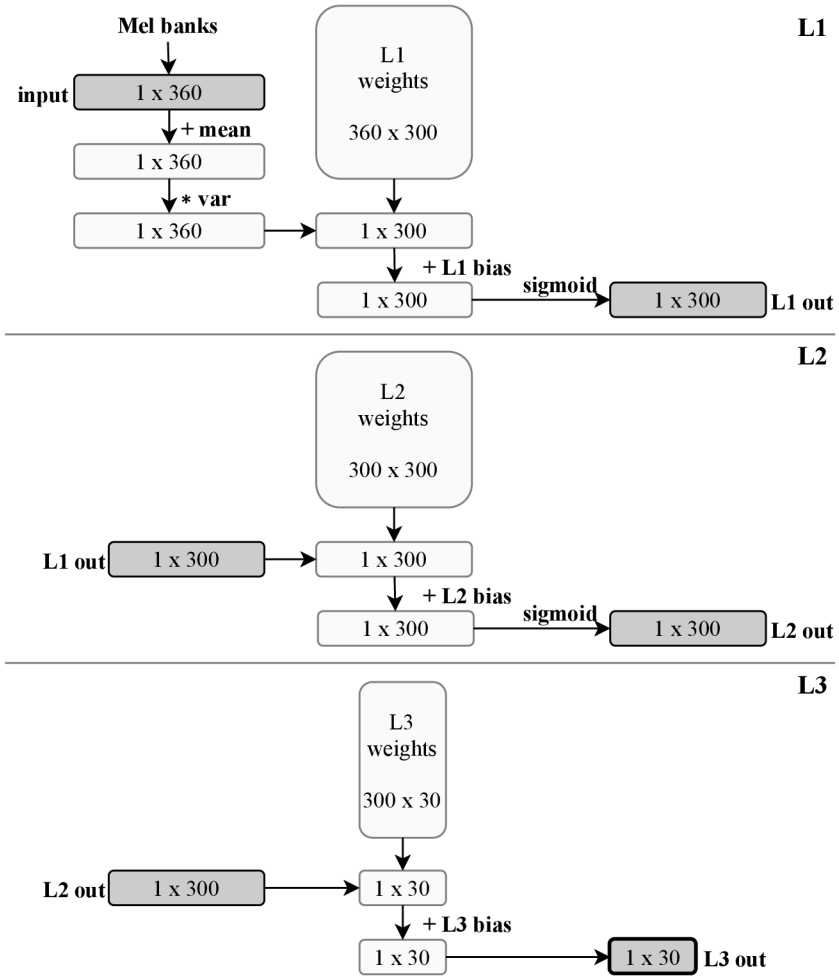

Figure 3.10: Three layers neural network displayed as a sequence of matrix operations.

first layer, the vector of mean values is added to the input vector, and then each value of the vector is multiplied by a vector of variance values.

3.5 Dynamic time warping

Dynamic time warping (DTW) is a common technique to find a nonlinear alignment of two time-dependent sequences of a digital signal. It is often used to compare different speech patterns (see figure 3.11).

The objective of DTW is to find surjective function $X \to Y$ of two sequences $X = (x_1, x_2, \ldots, x_N)$ of length $N \in \mathbb{N}$ and $Y = (y_1, y_2, \ldots, y_M)$ of length $M \in \mathbb{N}$. We can define *feature space* $\mathcal{F}$ as $x_n, y_m \in \mathcal{F}$ for $n \in X$ and $y \in Y$. Then, it is a problem of state-space search. The algorithm starts at (x_0, y_0), and the goal is to a find path to (x_N, y_M) with minimal overall cost. This algorithm needs a *local distance measure* (or *local cost measure*) which is a function $c : \mathcal{F} \times \mathcal{F} \to \mathbb{R}_{\geq 0}$. This cost depends on speech features similarity at that point. The distance matrix displays figure 3.12. The cost is computed using *cosine*

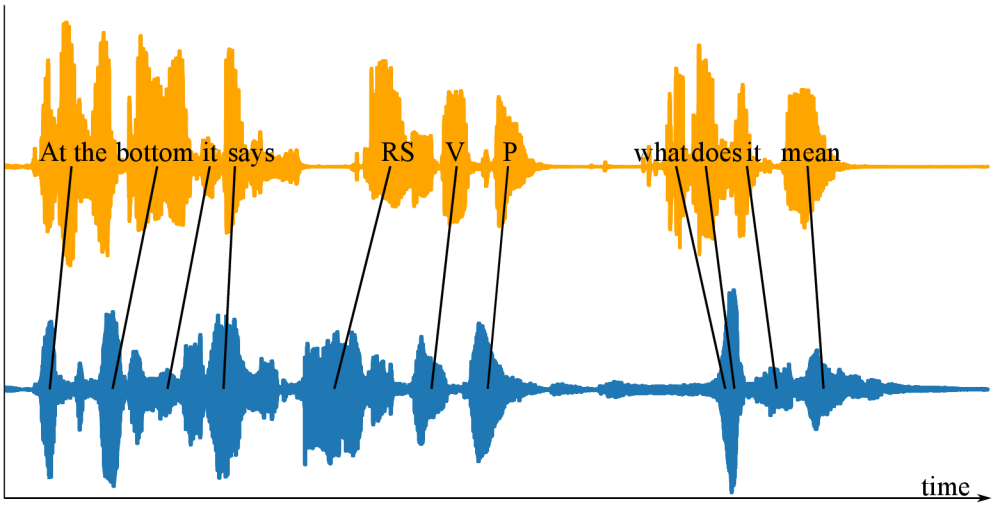

Figure 3.11: DTW algorithm principle of time alignment of two speech recordings.

distance [4] from features by the following formula:

$$c(x_n, y_m) = -\frac{\sum_{i=1}^{N} x_n[i] * y_m[i]}{\sqrt{\sum_{i=1}^{N} x_n[i]}\sqrt{\sum_{i=1}^{N} y_m[i]}} + 1; \tag{3.4}$$

where:

- **N** is the number of descriptors of each frame ($N = 30$ in this case)

When the cost matrix is computed, it is time for finding a warping path. The warping path is defined by the following definition taken from the book [3]:

Definition 3.1 *A (N, M)-warping path is a sequence* $p = (p_1, \ldots, p_L)$ *with* $p_\ell = (n_\ell, m_\ell) \in [1:N] \times [1:M]$ *for* $\ell \in [1:L]$ *satisfying the following three conditions:*

(i) Boundary condition: $p_1 = (1,1)$ *and* $p_L = (M,N)$

(ii) Monotonicity condition: $n_1 \leq n_2 \leq \ldots \leq n_L$ *and* $m_1 \leq m_2 \leq \ldots \leq m_L$

(iii) Step size condition: $p_{\ell+1} - p_\ell \in \{(1,0),(0,1),(1,1)\} for \ell \in [1:L-1]$

Before starting the finding the path, the matrix of the overall score is created and every point has a value of *Infinity* (means unreachable). Then the distance (score) from $p_1 = (1,1)$ to each point is calculated. The third phase is finding the best path, the path with the lowest overall price. During this phase, the algorithm notes the start and stop timestamps of each word. The algorithm compares even first the frame with the last frame. We can search only in the limited width from the diagonal as the speech with a path far from the diagonal of the DTW matrix would not pass any exam.

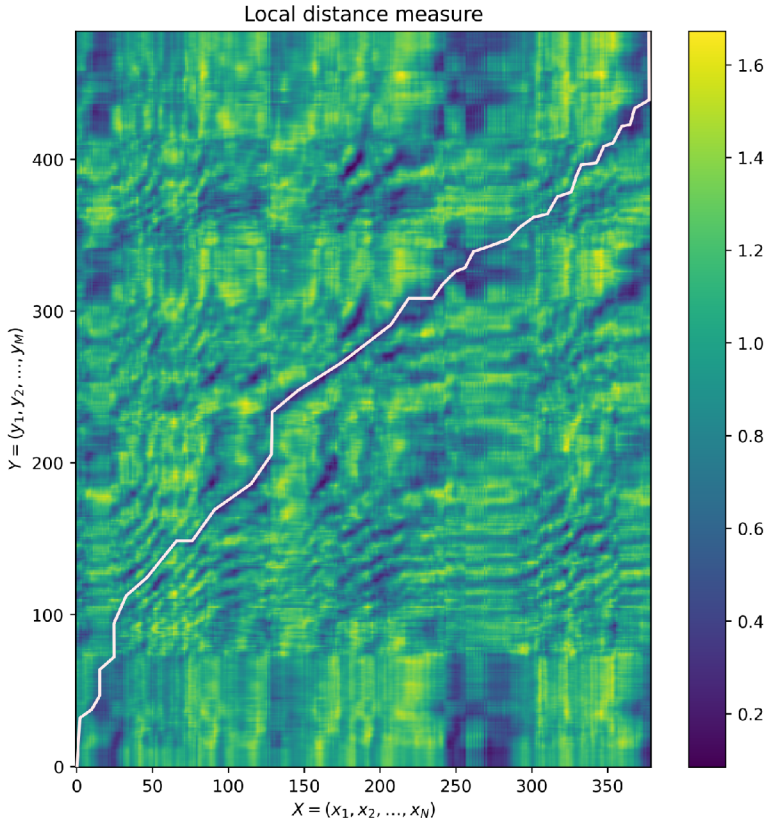

Figure 3.12: Distance (cost) matrix of two similar speech recordings. Dark blue means lower cost, which is better. The white line is drawn according to the best path matrix. This matrix does not have a rectangle shape, as the users recording **Y** is longer (5 s) than reference recording **X** (3 s).

3.6 Calculating the score

Overall Score consists of *Acoustic Similarity* and *Speed Ratio*. *Acoustic Similarity* is computed as a ratio of the total distance between all connected reference and recorded words. The best ratio is 1. The *Speed Ratio* is the ratio of the sum of lengths of all words in user and reference recording (silence is not included), the ratio 1 is the best score.

3.7 Conclusion

It is essential to understand what is behind the processing algorithm and divide the algorithm into independent parts. In the following chapters, we will improve the time complexity of those parts. Complexity depends on the number of samples, which depends on recording length and sampling frequency. Down-sampling can reduce the sampling frequency. We are using a widely used speech processing algorithm used for extracting phonemes and text from speech recording. We are only using the first part of the feed-forward neural network. Values from the bottleneck layer are the output of the algorithm as we are not decoding phonemes into transcription. To compare phonemes of reference and user audio recording we use DTW. The final score consists of acoustic similarity and speed ratio (for each word and as an overall score). The score is provided to the application as JSON.

Chapter 4

Refactoring

The original solution of *SpeechEngine* had shortcomings. It was necessary to fix all bugs causing application crashes. The engine contained an error in the filter, which pre-processes the input, decreasing evaluation accuracy. Now the application can work with multiple threads; asynchronous tasks are submitted into queues and dynamically assigned to the threads (using Kotlin coroutines[1]). In the previous solution, there was one thread shared for all background tasks.

4.1 Code readability and maintainability

The previous solution was based on sending messages between synchronous and asynchronous threads. The class diagram of the old solution displays figure 4.1, the new solution is in figure 4.2. From *React Native*, the requests come as asynchronous callbacks to the `ISpeechEngine` interface implementation. The main thread handles those callbacks. Requests are transmitted to the background thread by sending messages from the main thread in `SpeechWorkerThread` to the `handlerThread` in the same class. The `handlerThread` listens for incoming messages in a loop. It can handle only one background task at the same time. For communication from `handlerThread` of `SpeechWorkerThread` to the `SpeechEngine` (e.g., to report progress), the messages are sent to the `handlerThread` of `SpeechEngine` which scope allows calling *React Native* callbacks. The example communication that follows starting audio recorder shows the blue arrows.

The disadvantage of this solution is mainly only one task at the foreground thread, only one task at the background (processing) thread and one task at the state reporting thread (calling callbacks). The readability for programmers is difficult because of thread switching using messages. Simple task includes multiple call and messages to be sent. Jumping through code does not provide even advanced IDE like *Android Studio*, it is necessary to find usages of exact message through the project. The first idea was to move to *AsyncTasks*, which is perfect for the needs of this project, but it is now deprecated[2]. The solution is to extend the *Thread* object or to use *Kotlin coroutines*. Because Android is going forward to *Kotlin first approach*[3], we decided to rewrite the module controller into Kotlin and *Kotlin coroutines*.

[1] Kotlin coroutines on Android - *Android Developers*, https://developer.android.com/kotlin/coroutines

[2] AsyncTask is Deprecated, Now What? - *TechYourChance*, https://www.techyourchance.com/asynctask-deprecated

[3] Android's Kotlin-first approach - *Android Developers*, https://developer.android.com/kotlin/first

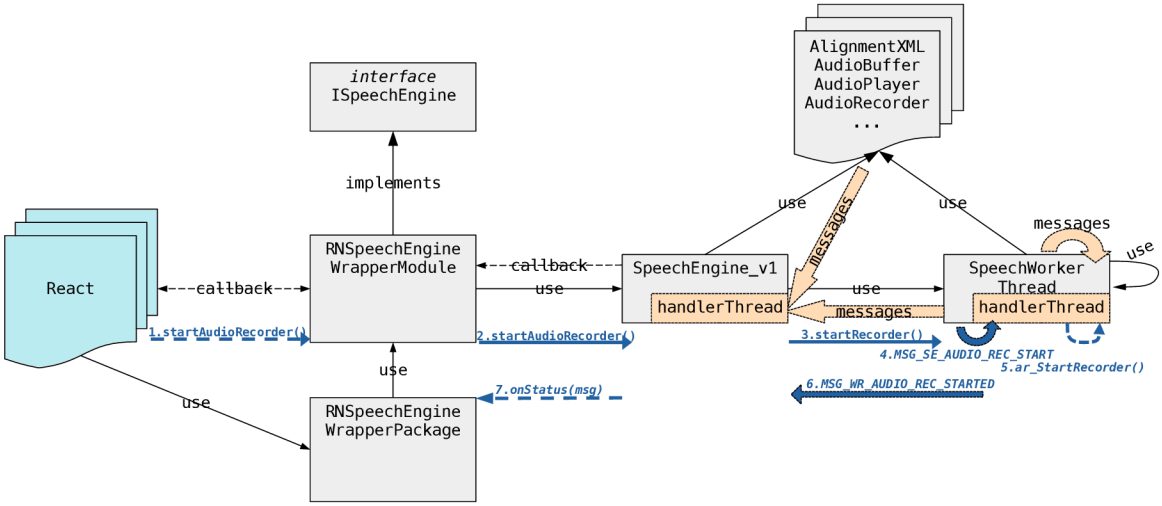

Figure 4.1: Class diagram of the old solution. The bold orange arrow represents messages, and orange blocks represent message receivers. Dashed arrows mean non-blocking communication considering the main thread. Blue arrows are an example of calls required to start the recorder.

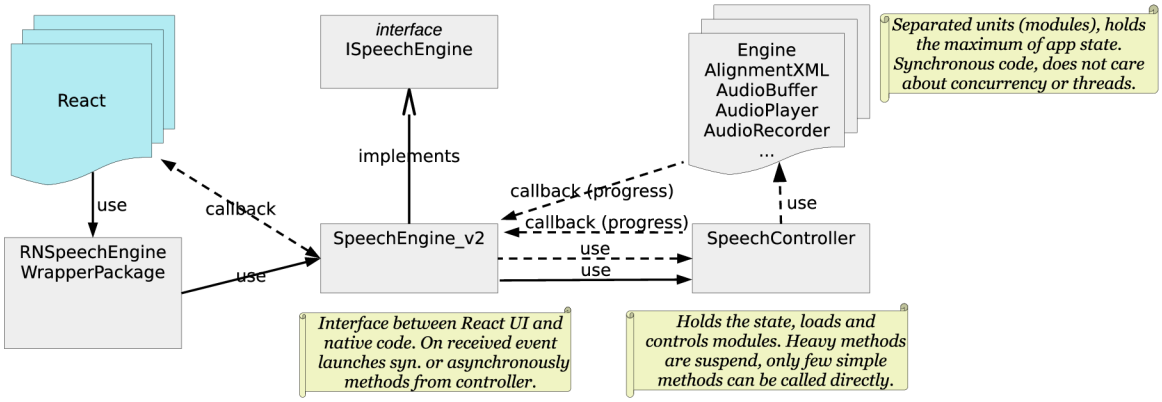

Figure 4.2: Simplified class diagram, after refactoring, without communication using messages. Dashed arrows mean non-blocking communication considering the main thread.

4.2 Kotlin with Java code

New parts of code are written in Kotlin and all classes except those from separated modules are ported into Kotlin. I refactored and ported them only when I needed to use new features like *Coroutines* or *Channels*. Some of the previous modules are still implemented in Java (e.g., `AlignmentXML` or `AudioBuffer`). Those two languages are mutually compatible, that allows the code written in one of this language to be used in the second of these languages[4]. The biggest change is the *nullable types*. When not explicitly stated, the objects may be null (type `T?`), or of the provided data type `T`. This ambiguity is represented as `T!` type in Kotlin. If we are aware that the value can be null, we should add `@Nullable` annotation to the Java code. If the null cases of value are handled in the code, we can add `@NotNull` annotation. Then it is not necessary to do null checks when using the value in Kotlin code.

4.3 Kotlin coroutines

Kotlin coroutines are design pattern and library to simplify code that executes asynchronously. Coroutines are an idea of suspendable computations. A suspendable function can suspend its execution at some point and resumes later. Launching an asynchronous block of code is like submitting it into the pool of manually created thread. Threads and pools are created and managed by the system. The number of threads depends on the number of CPU cores and the amount of concurrency work, but it is guaranteed to have at least two threads. The following code shows how to launch heavy execution and print info before and after execution. Specified *default dispatcher* (the targeted pool) determines coroutine context and on which thread(s) will be this code executed (it can be for example UI thread or thread designed for IO operations)[5].

```
1 private val scope: CoroutineScope = CoroutineScope(Dispatchers.Default)
2 fun speechEngineExecute() {
3     onStatus("RNWP − Starting speech engine")
4     scope.launch {
5         speechController.speechEngineExecute()
6         onStatus("RNWP − Speech engine finished")
7     }
8 }
```

The widely propagated benefit of *coroutines* is that when the job of scope is cancelled, it cancels all coroutines started in that scope, which brings more control over background tasks. Also, the function `speechEngineExecute()` can return value, or throw exceptions that can be caught and handled like in synchronous code.

[4]Mixing Java and Kotlin in one project, https://kotlinlang.org/docs/mixing-java-kotlin-intellij.html

[5]Coroutine context and dispatchers, https://kotlinlang.org/docs/coroutine-context-and-dispatchers.html

Chapter 5

Optimizing

In this chapter, I will propose possible improvements to decrease processing time with respect to the accuracy, implement them, compare and evaluate them. The chapter is divided into applied techniques or tools in chronological order as they were implemented.

To speed up the application, we can move computation closer to HW, use parallelization or optimize used algorithms. To use the HW features, we can rewrite the code to a low-level programming language (use *Android NDK*[1]), use compiler directives in already implemented code, or use a high-level library (e.g., *TensorFlow*).

5.1 Measurement methodology

In the following sections, this methodology of measuring computing time is used. ***First run*** means loading and running the program ten times and taking the median of the result. Side effects between invocations are not eliminated. ***Multiple evaluation*** means running the program and then running evaluation ten times and taking the median of the results.

Taking the median of values comes from my previous experience with measuring the time complexity of algorithms. If we take an algorithm with complexity described by an equation, then the average is impacted by peaks in multiple measurements, thus, it is not applicable. Median follows the expected values better, and taking the lowest value almost follows the equation. The lowest value is best for comparing different algorithms or methods isolated from OS or other applications. Median is best for speed up measurement and comparing before/after, as some algorithm may be prone to random interruptions or actual load of a device (e.g., algorithms using multiple cores). We decided to consider the runtime environment, as Android users usually have no control over running processes in the background, so we are using the median.

Time of invocation was measured using the most precise available system timer `System.nanoTime()` as a difference between time before invocation a function and time after returning from the function. Measured time using custom code can be logged or printed into the user interface with no need to install development tools, which was necessary, as measurements on other devices could not be performed in person due to the pandemic. Another option is to use a profiler bundled with *Android Studio*. A usable result was achieved with *Java Method Sample Recording* configuration by composing the time from multiple threads, and from times before and after suspending. But some functions were missing in the profiler output, and some of the times did not correspond to reality. *Trace*

[1] Android NDK - *Android Developers*, https://developer.android.com/ndk

Java Methods configuration did not provide good results. Thus, custom implementation fits better this project.

5.2 Profiling before optimization

Before proposing any speed up it is necessary to analyse the existing solution and focus on parts with a big impact on processing time. The profiled code is the original code with rewritten controllers, fixed bugs, handled exceptions and some small improvements. The original code before factorization would be hardly profiled. Some parts would be impossible to measure (e.g., **Load engine config**).

	ms	%
Load Engine Config	7 687.6	74.7
Load Task Config	629.7	6.1
Select Task Segment	812.7	7.9
Init AudioRecorder	33.0	0.3
Execute	1 123.3	10.9
$\sum$	10 253.2	

Table 5.1: Profiling by comparing elapsed time. The recording is not included as it is almost the same as the recording duration. The evaluation was executed as ***first run*** on 5 s recording.

Execute ***first run***	*ms*	%
Prepare Static Data	4.5	0.5
Down-sampling	19.2	2.1
Mel banks	223.2	24.2
FFNN	611.1	66.2
DTW	53.9	5.8
Calculate Score	0.9	0.1
Print JSON	10.6	1.2
$\sum$	923.4	

Execute ***multiple evaluation***	*ms*	%
Prepare Static Data	1.7	0.2
Down-sampling	14.4	1.6
Mel banks	221.8	24.1
FFNN	619.3	67.3
DTW	53.2	5.8
Calculate Score	0.6	0.1
Print JSON	9.1	1.0
$\sum$	920.0	

Table 5.2: Profiling detail of **Execute**. The evaluation was executed on the 5 s recording. The sum of times in the left table is less than time in 5.1, and it is caused by **Execute** function overhead.

The most expensive is **Load Engine Config**, specifically loading neural network. Parsing neural network is implemented using `RandomAccessFile`, which is slow, and it is a known issue[2]. The second most expensive part is **Execute**. If we investigate **Execute** part, we can see that the most expensive part is computing **FFNN** (feed-forward neural network). It takes about 0.6 seconds. And if we add loading of neural network from **Load Engine Config** part, it takes 7 seconds to load and compute the neural network. To improve that, we used *TensorFlow Lite*, which describes the next section. If we compare ***first run*** with the following ***multiple evaluation***, the percentages of each phase are almost

[2]JDK-4056207 Add a buffered version, https://bugs.openjdk.java.net/browse/JDK-4056207

the same. The time is saved for example by already allocated memory, but the time of the heaviest parts (**FFNN** and **Mel banks**) are not significant.

5.3 TensorFlow Lite

TensorFlow Lite is an open-source high-level deep learning framework for mobile and IoT devices developed by Google, presented during Google I/O in 2017[3]. It is a simplified version of TensorFlow, an open-source deep-learning software library for defining, training and deploying machine learning models, that was open-sourced in November 2015 by Google [1]. It provides hardware acceleration or parallelization. The same code can be used for example on GPU or multi-core CPU.

TensorFlow Lite consists of two main components - converter and interpreter. TF Lite works with compressed models as displayed in figure 5.1. First, it is necessary to pick an existing model or create own. Trained models are available through the TensorFlow Hub repository[4], but it is possible to create a custom model using supported operations.

Figure 5.1: TensorFlow Lite interpreter works as a filter with one input and one output tensor.

5.3.1 Creating custom model

The custom model can be any model that can benefit from HW acceleration or parallelization, not just a model based on machine learning. TF Lite operations support 32-bit floating-point and quantized (`uint8`, `int8`) values. Strings or 16-bit floats are not supported yet. We are using 32-bit floating-point values obtained from `AudioRecorder`. As 2021 the TF Lite model supports 123 operations, but even if the operation does not have a direct equivalent, it can be fused into a more complex operator, replaced by tensors, or removed from the computation graph[5].

TensorFlow Lite model can be converted from TensorFlow *SavedModel* using the *TensorFlow Lite converter*. TF Lite model is an optimized `FlatBuffer` format identified by the `.tflite` file extension. The recommended way, with more features, is converting TF *SavedModel* or *Keras* model using Python API (see figure 5.2). A simple way, how to set up the TensorFlow environment with Python API, is to use a Docker image containing configured TensorFlow and Jupyter server[6].

The feed-forward neural network, displayed in figure 3.10, can be implemented in TensorFlow as `tf.Module`. The `Module` class describes a sequence of operations, as shows the listing 5.1. Static values (e.g., variables like `NN_layer1_mean`) are *NumPy arrays*, and

[3]Google's new machine learning framework is going to put more AI on your phone - *The Verge*, https://www.theverge.com/2017/5/17/15645908

[4]TensorFlow Hub - a repository of trained TF models, https://tfhub.dev

[5]TF Lite and TF operator compatibility, https://www.tensorflow.org/lite/guide/ops_compatibility

[6]Install TensorFlow using Docker, https://www.tensorflow.org/install/docker

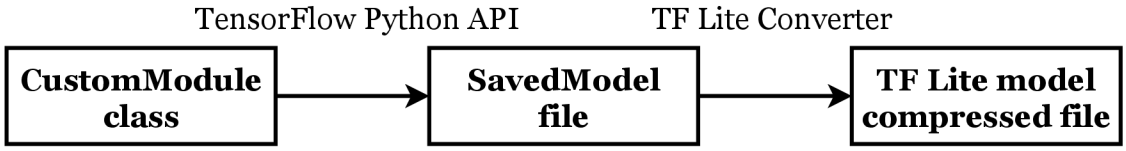

Figure 5.2: Process of converting *CustomModule* class into compressed TF Lite model.

they are stored in *SavedModel* as well. Those arrays are loaded from a file, reshaped or transposed, but the loading and pre-processing is not part of the model. The model stores constant data, which makes interpreting the model faster (but sometimes with the cost of a bigger model).

```
class CustomModule(tf.Module):
  # Init defines and loads static values, their shape and used data types, stored inside the model
  def __init__(self):
    super(CustomModule, self).__init__()
    self.nn_layer1_mean = tf.constant(NN_layer1_mean, shape=(1,360), dtype=tf.float32)
    self.nn_layer1_var = tf.constant(NN_layer1_var, shape=(1,360), dtype=tf.float32)
    self.nn_layer1_weights = tf.constant(NN_layer1_weights, shape=(360,300), dtype=tf.float32)
    self.nn_layer1_bias = tf.constant(NN_layer1_bias, shape=(1,300), dtype=tf.float32)
    self.nn_layer2_weights = tf.constant(NN_layer2_weights, shape=(300,300), dtype=tf.float32)
    self.nn_layer2_bias = tf.constant(NN_layer2_bias, shape=(1,300), dtype=tf.float32)
    self.nn_layer3_weights = tf.constant(NN_layer3_weights, shape=(300,30), dtype=tf.float32)
    self.nn_layer3_bias = tf.constant(NN_layer3_bias, shape=(1,30), dtype=tf.float32)

  # Function 'call' describes a sequence of operation applied to the input tensor
  @tf.function(input_signature=[tf.TensorSpec((1,360), tf.float32)])
  def __call__(self, x):
    # Layer1 (first hidden layer with normalization)
    tensor = tf.add(x, self.nn_layer1_mean)
    tensor = tf.multiply(tensor, self.nn_layer1_var)
    tensor = tf.matmul(tensor, self.nn_layer1_weights)
    tensor = tf.add(tensor, self.nn_layer1_bias)
    tensor = tf.sigmoid(tensor)
    # Layer2 (hidden layer)
    tensor = tf.matmul(tensor, self.nn_layer2_weights)
    tensor = tf.add(tensor, self.nn_layer2_bias)
    tensor = tf.sigmoid(tensor)
    # Layer3 (bottleneck layer and the output layer at the same time)
    tensor = tf.matmul(tensor, self.nn_layer3_weights)
    tensor = tf.add(tensor, self.nn_layer3_bias)

    return tensor
```

Listing 5.1: FFNN implemented using TensorFlow Python API as TensorFlow Module.

TensorFlow Module is then saved into the `custom_module` file in the *SavedModel* file format. The *SavedModel* file can be loaded back into TensorFlow and can be interpreted. It is useful to check the output of this model before and after exporting using sample input (see listing 5.2).

```
# Save custom model
module = CustomModule()
tf.saved_model.save(module, 'custom_module')
# Load custom model
loaded = tf.saved_model.load('custom_module')
# Run models
```

```
module(test_tensor).numpy().round(6)
loaded(test_tensor).numpy().round(6)
```

Listing 5.2: Save and load the custom TensorFlow Module as *SavedModel*.

If the model outputs meet expected results, the model can be converted into TensorFlow Lite binary file using listing 5.3. The TF Lite model cannot be loaded and run under Python API, so it is necessary to verify the correctness of the TensorFlow model.

```
converter = tf.lite.TFLiteConverter.from_saved_model('custom_module')
tflite_model = converter.convert()
with open('model.tflite', 'wb') as f:
  f.write(tflite_model)
```

Listing 5.3: Convert the TensorFlow model into the TensorFlow Lite model.

The same code can be implemented using *Keras*[7]. *Keras* is a deep learning API written in Python running on top of the machine learning platform TensorFlow. It can combine matrix multiplying, bias, and activation function (sigmoid) into a *Dense* layer, see figure 5.4. Layers of the *Keras* model are defined inside the `init` function, normalization is applied in the `call` function (before the invocation of the *Keras* model).

```
self.model = tf.keras.Sequential([
    tf.keras.layers.Dense(300, activation="sigmoid", name="layer1",
                          weights=[self.nn_layer1_weights, self.nn_layer1_bias] ),
    tf.keras.layers.Dense(300, activation="sigmoid", name="layer2",
                          weights=[self.nn_layer2_weights, self.nn_layer2_bias] ),
    tf.keras.layers.Dense(30, activation=None, name="layer3",
                          weights=[self.nn_layer3_weights, self.nn_layer3_bias] )
])
```

Listing 5.4: The neural network defined as a *Keras* model inside the `init` function.

5.3.2 Deploying model

The TensorFlow Lite interpreter is a library that loads a model file, then takes input data, executes the operations defined by the model on input data, and produces the output data, see figure 5.1. The interpreter works across multiple platforms and provides a simple API for running TensorFlow Lite models from Java or Kotlin, Swift, Objective-C, C++, and Python. It can be used on Android, iOS or Linux platform[8].

Running TF Lite interpreter works in the following steps - loading a `.tflite` model into memory, transforming input data, running inference (executing model), interpreting the output. Input and output tensors are primitive type arrays (`float`, `int`, `long`, `byte`, or `String`). Complex data types like `Integer` or `Float` are not supported. Using primitive types as input makes the invoking slow. The interpreter always checks the shape of the input array and tries to reshape it, which causes a slowdown. The API is more efficient if a direct `ByteBuffer` (or `FloatBuffer`, `IntBuffer`, `LongBuffer`) is used as the input data type of interpreter. The input primitive type array can be wrapped into a buffer using `ByteBuffer.wrap()` function. Wrapping data is fast enough to do not affect the model execution time. It is not clear whether buffer should be preferred for output as well. According to my experiments, using primitive type arrays as output is as fast as using `ByteBuffer`.

[7] Keras - a deep learning API, https://keras.io

[8] TensorFlow Lite inference, https://www.tensorflow.org/lite/guide/inference

Invocation is done by the `interpreter.run(input, output)` command. The application cannot change the behaviour of the binary model. But the same model can be invoked with different environment configurations, which can be changed dynamically during the run. TensorFlow Lite allows running accelerated computation. Those accelerators are called *delegates*. Using the right delegate for a specific model and device can have a big impact on execution time.

By default, TensorFlow Lite uses CPU kernels optimized for the ARM Neon instruction set. However, the CPU is a multi-purpose processor that may not be suitable for the heavy arithmetic typical in machine learning models[9]. Or even when there is no suitable delegate, the model can use parallel computation on multiple CPU cores. Available delegates depend on the platform, and in the case of Android, even on the Android version. Some delegates support only certain types of model, as shows the table 5.3.

Model Type	**GPU**	**NNAPI**	Hexagon	CoreML
Supported platforms	**Android**, iOS	**Android 8.1+**	Android	iOS
Floating-point (32 bit)	**Yes**	**Yes**	No	Yes
Post-training float16 quantization	Yes	No	No	Yes
Post-training dynamic range quantization	Yes	Yes	No	No
Post-training integer quantization	Yes	Yes	Yes	No
Quantization-aware training	Yes	Yes	Yes	No

Table 5.3: TF Lite delegates platform and model type support. Delegates suitable for this project are bold.

5.3.3 Profiling

Finding the best delegate and its configuration, like the number of threads or number of inputs values, maximises the benefit of the *TensorFlow*. The achieved results may also depend on used devices.

Delegates and models

In figure 5.3 you can see the time of different delegates during loading and execution. Each delegate is with the model implemented using TF functions and using *Keras* model. Initialization of GPU variants costs significantly more time (GPU 277 ms, NNAPI 5 ms, and CPU variant only 3 ms). CPU is also slower in the execution of these models. NNAPI and CPU variants have similar time. The CPU variant is the winner as this variant can be run on any device with the same speed as NNAPI.

We can take into account the times of ***multiple evaluation***. These times are displayed in table 5.4. The table compares only times of *FFNN*. There is no significant speed improvement with the next execution.

[9]TensorFlow Lite Delegates, https://www.tensorflow.org/lite/performance/delegates

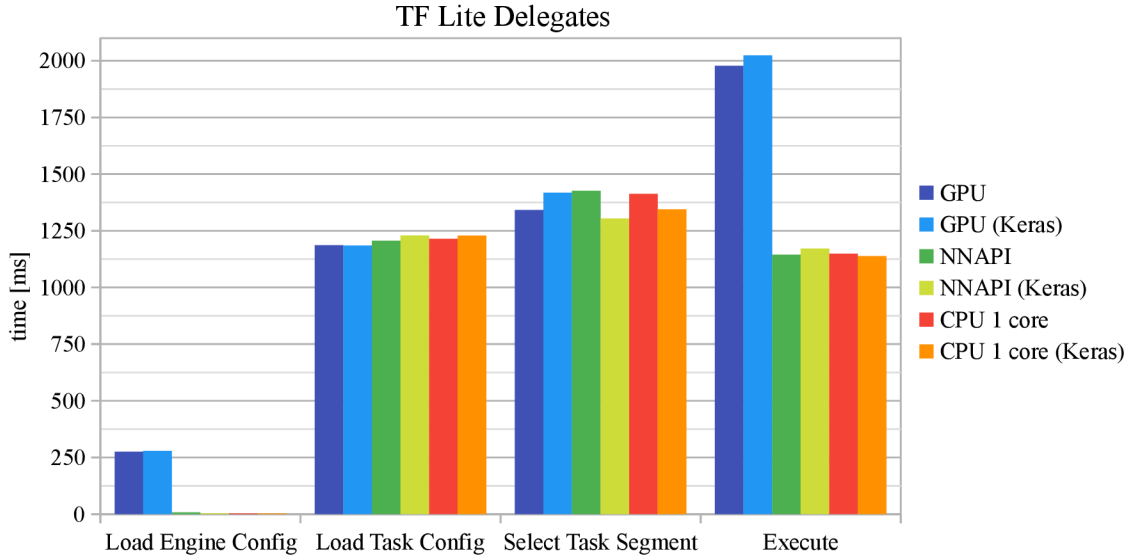

Figure 5.3: Execution times of TF Lite delegates on 10 seconds recording.

TF Lite Delegate	***first run*** $[ms]$	***multiple evaluation*** $[ms]$
GPU	1 157	1 157
GPU (Keras)	1 185	1 104
NNAPI	362	353
NNAPI (Keras)	385	361
CPU 1 core	362	363
CPU 1 core (Keras)	359	351

Table 5.4: *Execute* times of *FFNN* on 10 s recording using different delegates.

The reason why TF Lite delegates are slower than CPU is the small size of the model. We have a small model, which is not worth delegating to either the NNAPI or the GPU. Accelerators are better for large models with high arithmetic intensity[10].

In the case of GPU, the *TensorFlow Lite Interpreter* needs to copy data into CPU before execution and copy output from GPU into CPU memory. The other reason is that tensor data is sliced into 4-channels for GPU delegate. But we have an input vector (1D data), eventually a 2D matrix (in case of processing multiple frames at once). Input has to be transformed into 4-channel values, which will affect the final speed[11].

Delegates can be also slowed down by operator incompatibility. Delegates do not support all operators as TF Lite. If the model uses not supported operator, the operation has to be computed on the CPU. This will require synchronization of HW or copy-in/copy-out in the case of GPU, which will reduce the speed. As this behaviour is undesirable, it is disabled by default, and the delegate would throw an exception instead of execution. Our models are compatible with all these delegates.

The results also depend on the used chipset (CPU, GPU, and various coprocessors). The following experiment explains when it is better to use GPU or NNAPI than CPU. We

[10]TF Lite performance best practices, https://www.tensorflow.org/lite/performance/best_practice
[11]TensorFlow Lite on GPU, https://www.tensorflow.org/lite/performance/gpu_advanced

tried different models of different size and on different chipsets. In all experiments, we use *Samsung Galaxy A40*. In this comparison, we use different devices listed in table 5.5.

Device	**Samsung Galaxy A40**	**Samsung Galaxy Tab S6 Lite**
Released	2019	2020
Chipset	Exynos 7904 (14 nm)	Exynos 9611 (10nm)
CPU	8 cores: 2x1.77 GHz Cortex-A73 6x1.59 GHz Cortex-A53	8 cores: 4x2.3 GHz Cortex-A73 4x1.7 GHz Cortex-A53
GPU	Mali-G71 MP2	Mali-G72 MP3
Device	**Xiaomi Mi 9**	**LG G8S ThinQ**
Released	2019	2019
Chipset	Qualcomm SM8150 Snapdragon 855 (7 nm)	Qualcomm SM8150 Snapdragon 855 (7 nm)
CPU	8 cores: 1x2.84 GHz Kryo 485 3x2.42 GHz Kryo 485 4x1.78 GHz Kryo 485	8 cores: 1x2.84 GHz Kryo 485 3x2.42 GHz Kryo 485 4x1.78 GHz Kryo 485
GPU	Adreno 640	Adreno 640

Table 5.5: Devices used in the experiments with bigger models.

The first model in figure 5.4 is a neural network with 6 624 630 parameters (input matrix enlarged from 1x360 to 32x360, hidden and bottleneck layers contain 32 times more perceptrons). This model produces valid outputs; it computes 32 frames at once. The file of the compressed TF Lite model increased from 0.8 MB to 1 MB. On all devices, the CPU variant was still slightly faster than the NNAPI variant. There is notable that only previously used *Samsung Galaxy A40* has similar performance with NNAPI and CPU. GPU variant is missing as the TF Lite was crashing during the execution of this model. GPU delegate is still marked as experimental, similar bug reports to this problem can be found on the TF Lite GitHub repository. This model does not have sufficient arithmetic intensity to run using NNAPI.

The second model is a neural network with three duplicated inner layers. The network has 388 230 parameters, input and output layers are the same. The output is not valid. This model is used only to compare delegates. Dimensions of static data (like weights) are the same, so the size of the model remained 0.8 MB. This model works on the GPU but is six times to 10 times slower than the CPU variant. Three inner layers are not still enough to use the GPU or the NNAPI, which is still slightly slower.

The third model has 1 928 628 000 parameters (input layer is 1200x360, the hidden layer has 1 440 000 perceptrons, and the output layer has 36 000 perceptrons). Even this model is not valid, but model size increased from 0.8 MB to 7.6 MB. In this model, we can see that GPU speed up is 1.3 to 1.7 against CPU variant. The GPU is the best option in this case, and with even bigger models, GPU will be more efficient. NNAPI is probably not suitable for these models (without reinforcement learning and other advanced NN features).

5.3.4 Batch processing

The model can compute multiple frames at once. That was used in the second model in figure 5.4. It was not good to delegate this model to the GPU or NNAPI, but it can still

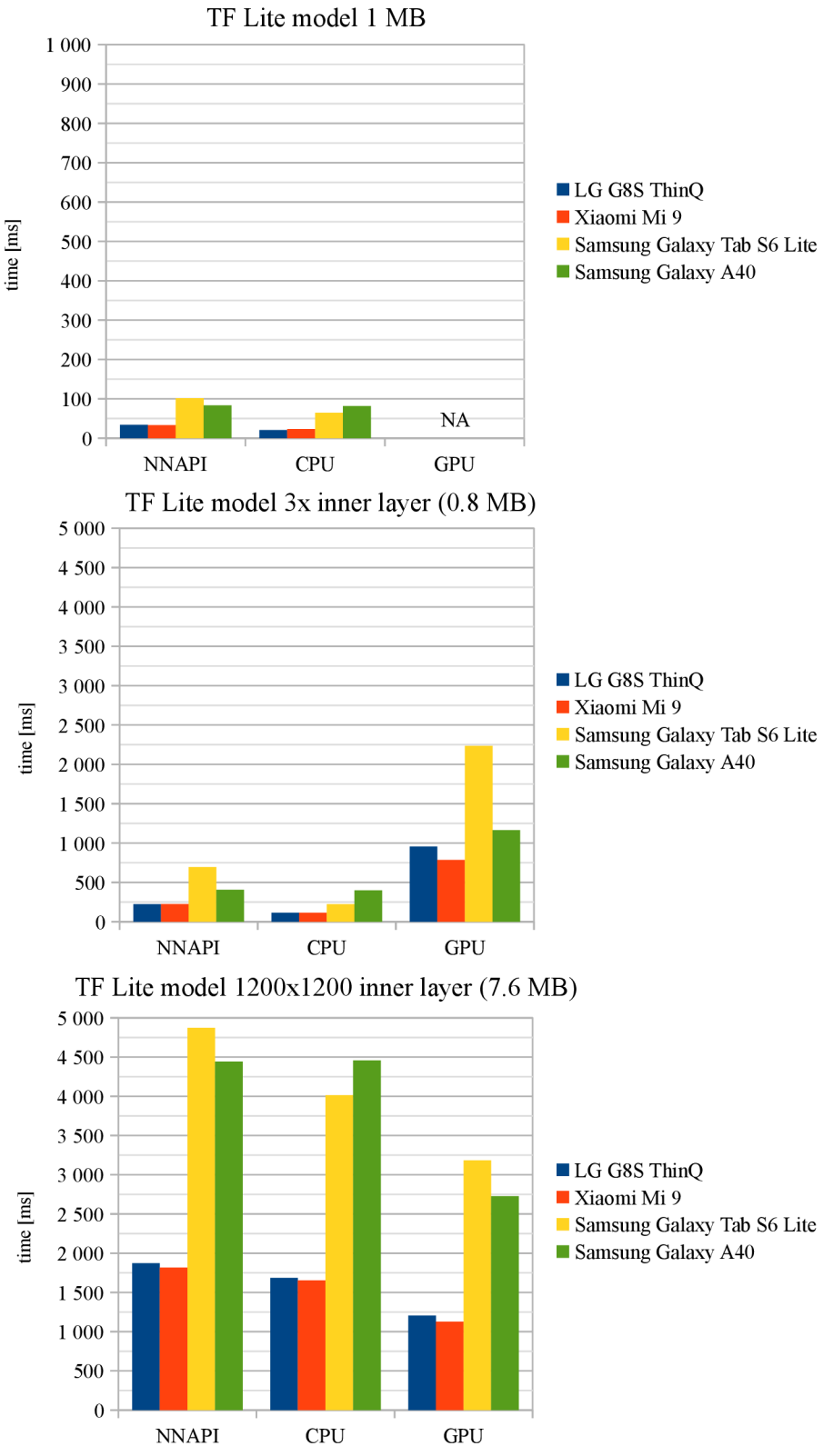

Figure 5.4: Execution times of TF Lite delegates on bigger models.

bring some improvement to the CPU. Before execution, the data has to be prepared and copied into the input buffer. After the execution, TF Lite copies the output into the output array. Computing multiple frames at once (batch processing) can reduce this overhead, but it will require a bigger model and more operation memory for computing. The batch size cannot be changed in runtime as changing the input and output shape of the model requires creating and converting a new model. The workaround would be storing multiple models, but it increases the size of the app.

Figure 5.5 displays dependency of *Execute* time on the number of frames computed in one invocation of the TF Lite model. We also tried four models on the NNAPI (but there is still no benefit of the NNAPI). There is a notable increase when computing two frames in a batch. It requires more array copies, and the handling of model input/output is complicated. Starting from 4 frames per batch, there are significant time savings. From 12 frames per batch, the savings are smaller, and from 24 frames the time is not decreasing at all but slightly increasing. We chose the model with 16 frames per batch. The size of this model is 895 kB which is acceptable.

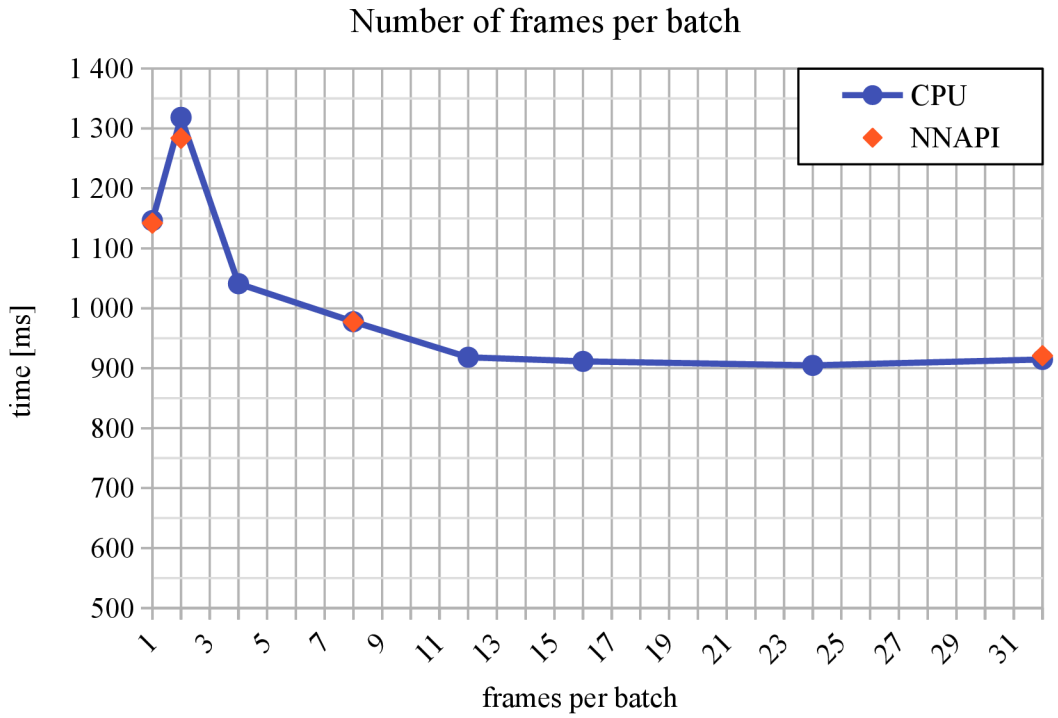

Figure 5.5: Number of frames in one batch and the time of *Execute* processing.

5.3.5 Number of threads

By default, the TF Lite CPU delegate uses only a single thread, as only some operators can be parallelized. Multi-threading may speed up execution, but it will consume more resources and power. The speedup is device-dependent and depends on the actual usage of the CPU by other apps.

As it is clear in figure 5.6, using multi-threading does not come with speed up for a simple model without computing in batches, and it does not matter whether *Keras* version is used or not. All provided devices have only eight cores. Thus it is pointless to try

more than eight threads. We also tried to use multi-threading in combination with batch processing. Figure 5.7 compares the time of different batch size computed using a single thread and using eight threads. The single thread variant is slightly faster, as running a single thread is easier for the scheduler.

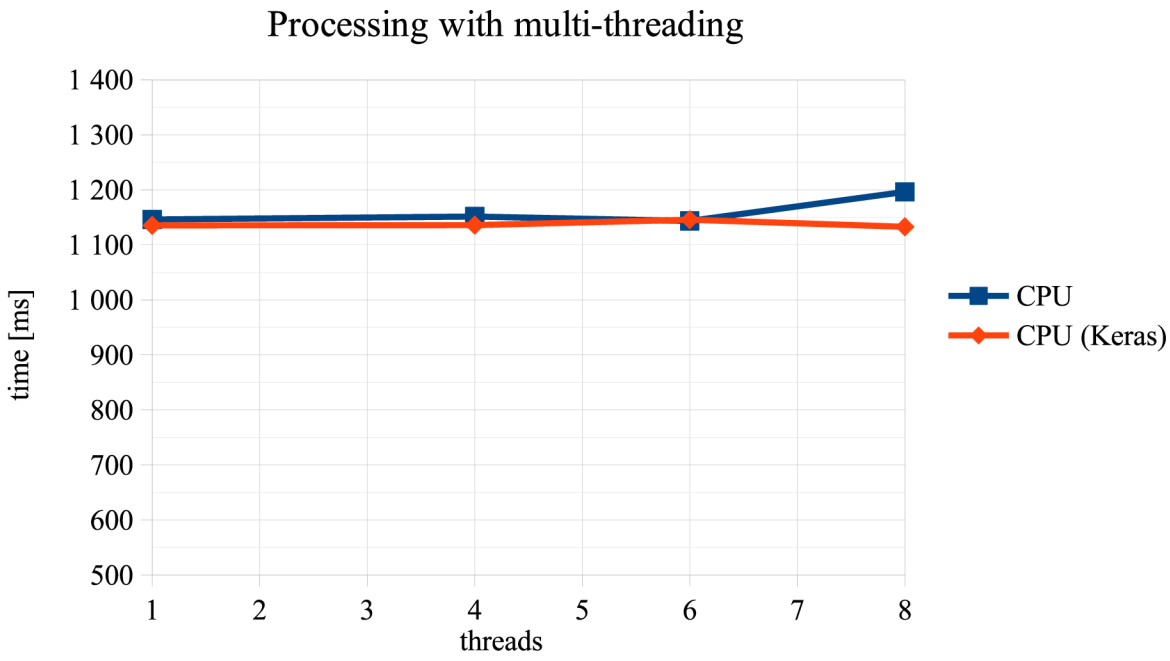

Figure 5.6: Using multiple threads does not speed up *Execute* processing.

5.3.6 Conclusion

CPU delegate best suits our model, but using multi-threading is useless in our case. Using batch computing is a good improvement and computing 16 frames at once seems like a good choice. Previous charts, and graphs, were measured on 10 seconds recording. Longer recording means bigger differences. Table 5.6 compares the median of times before and after implementing TF Lite both on 10 seconds and 5 seconds recording. With 5 seconds recording, the **Execute** time was reduced from 1.1 seconds to 0.3 seconds, and **Load Engine Config** (loading of neural network model) was reduced from 7.6 seconds to 0.8 seconds. **Execute** done under a second means sufficient time from the user's point.

5.4 Parallelization

Speech processing time can be more improved. For example, **Prepare static data**, and **Down-sampling** can be done during recording. That would save 12 ms, which is only 3.1 % of processing time. We need to identify parallelizable code and find those parts that prevent code from parallelization. Those problematic parts are: using the number of samples, using the sum of samples, or using samples "from future".

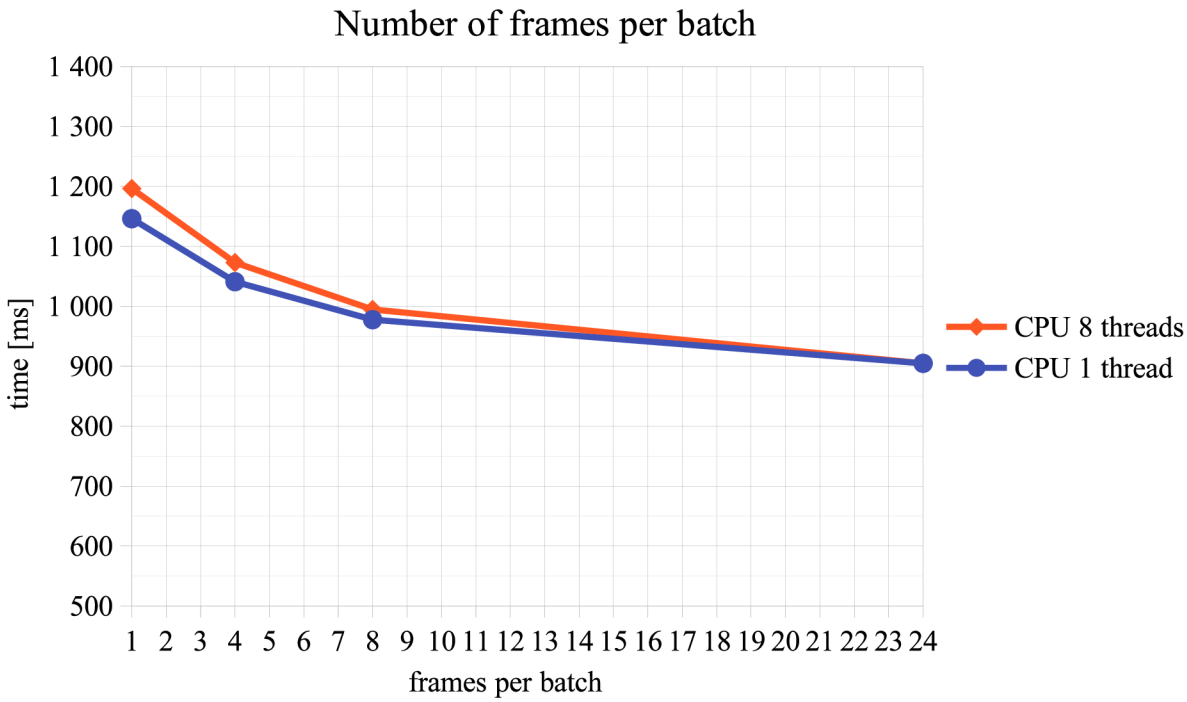

Figure 5.7: Time of *Execute* processing is slightly faster when using a single thread.

5 s recording	**before TF Lite** $[ms]$	%	**with TF Lite** $[ms]$	%
Load Engine Config	7 687.6	74.7	7.9	0.4
Load Task Config	629.7	6.1	658.2	36.1
Select Task Segment	812.7	7.9	808.9	44.3
Init AudioRecorder	33.0	0.3	33.0	1.8
Execute	1 123.3	10.9	316.3	17.3
$\sum$	10 253.2		1 791.3	
10 s recording	**before TF Lite** $[ms]$	%	**with TF Lite** $[ms]$	%
Load Engine Config	7 677.3	54.7	7.6	0.2
Load Task Config	1 415.5	10.1	1 424.0	34.7
Select Task Segment	2 191.8	15.6	1 635.1	39.9
Init AudioRecorder	33.0	0.2	33.0	0.8
Execute	2 714.6	19.3	1 000.8	24.4
$\sum$	13 999.2		4 067.5	

Table 5.6: Profiling by comparing elapsed time without and with implemented TF Lite. The evaluation was executed as the ***first run***. The recording is not included as it is almost the same as the recording duration.

5.4.1 Identify parallelizable code

Static data has to be prepared when the recording starts and before the first data are produced by `AudioRecorder`. The **Prepare Static Data** block can be split into two parts and make the maximum of preparing (e.g., initiating Hamming window) in previous loading

Execute ***multiple evaluation***	*ms*	%
Prepare Static Data	2.7	0.7
Down-sampling	8.8	2.4
Mel banks	219.5	59.3
FFNN	56.0	15.1
DTW	46.9	12.7
Calculate Score	1.1	0.3
Print JSON	34.9	9.4
$\sum$	369.9	

Table 5.7: Profiling with TF Lite by comparing elapsed time. The evaluation was executed on a 5 s recording. The sum of times in this table does not match the time in **Execute** in the above table 5.6 as those tables come from different measurement.

phases and minimum before each processing. All global variables have to be checked for R/W conflicts.

Down-sampling can be parallelized as it simply takes every nth sample and drops others. The first challenge comes with *framing*. `AudioRecorder` produces chunks of samples, that has to be framed. The last frames will overlap into "future" samples, see figure 3.3. Original code would handle this like end of recording and pad samples by zeros. That would make the speech regularly interrupted by silence, and it would worsen the final score. The number of frames has to be rounded down, and samples starting from the first omitted frame has to be copied into the beginning of the next chunk from `AudioRecorder`, see figure 5.8. The recording is not distorted during recording, but up to 199 last samples of recording are dropped (as they are not padded by zeros). That means up to 25 ms. Considering the recorder stops recording ±200 ms against the required length, a deviation of 25 ms does not introduce much inaccuracy into the evaluation.

Hamming window and *Discrete Fourier transform* can be parallelized as well. *Mel banks* can be computed except the final normalization. The normalization computes the difference of the banks mean and actual value. That requires knowing the sum of the bank and number of frames, as this module uses normalization over all frames, so $N = num\_frames$. We can use *Uniform last-N normalization*, a floating window of width $N < num\_frames$. But the accuracy grows with N so the best result will be with $N = num\_frames$. Another option is exponential normalization [5]. In the following experiments, I will keep $N = num\_frames$ and exclude normalization from *Mel banks* as normalization itself takes only 3.7 ms.

The *feed-forward neural network* will not be parallelized at this time, as I decided to use normalization through all frames, which stop the parallel part. But it can be parallelized. The only difference is that the neural network uses previous frames as input and works like multiple shift registers. Thus, this step (the controller of FFNN) has to preserve the inner state between each invocation. We can store input data and pointers into the input vector as a global variable that is reset only at the beginning at *Prepare static data* phase.

Some parts of *Dynamic time warping* could be parallelized as well. The cost matrix can be computed piecewise for just computed frames, and the matrix of the overall score can be initiated to *Infinity*.

Another thing to consider is the approach to parallelization. We can use pipelining known from CPUs or use linear processing on a background thread (see figure 5.9). The first approach is effective when each phase takes the approximately same time. Otherwise,

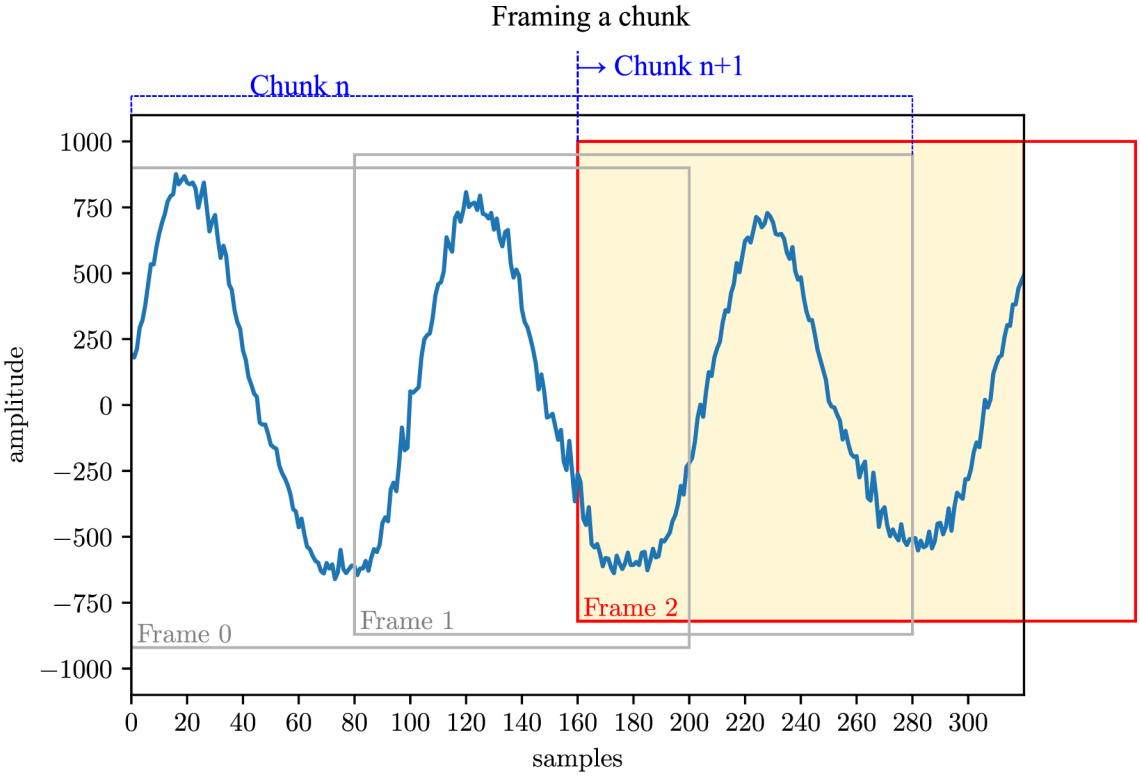

Figure 5.8: A chunk of 320 samples. Frame 2 (samples 160 to 299) will be copied into the next chunk and the current chunk will process samples from 0 to 159 (frame 0 and 1).

the time will depend on the slowest part. In our case, the whole processing is faster than recording (to process 5 seconds of recording, it takes 0.3 s), so threads 2, 3 and 4 would not be fully utilized (resource wasting). The second approach can be used, as the whole linear process of thread 2 is faster than recording. The advantage of this approach is using only two threads, instead of 4. That means one queue and mutex between threads instead of 3. We will use the second approach and check whether the second thread finishes processing before new data comes from the recorder.

5.4.2 Implementation

Background thread (task) is implemented as *Kotlin Coroutine* (see chapter 4.3). The recording thread creates a new asynchronous task and then continues to start recording. It is a simple "producer-consumer problem". The recording thread is a producer, and the new asynchronous processing task is a consumer. That can be easily implemented using *Kotlin Channels*[12]. Channels are like queues. It provides two interfaces, `SendChannel` and `ReceiveChannel`. The channel capacity can be 0, which is called the *rendezvous* queue, and the sender has to meet with the receiver to transfer value. Or the capacity can be limited, and more values are dropped or the sender is suspended until the queue is free[13]. The capacity can be unlimited, which is used in this case. The sender is then never suspended. If there are no values inside the queue, the receiver is suspended. If the sender has no

[12]Kotlin Channels, https://kotlinlang.org/docs/channels.html

[13]Kotlin Channels API documentation, https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.channels/-channel

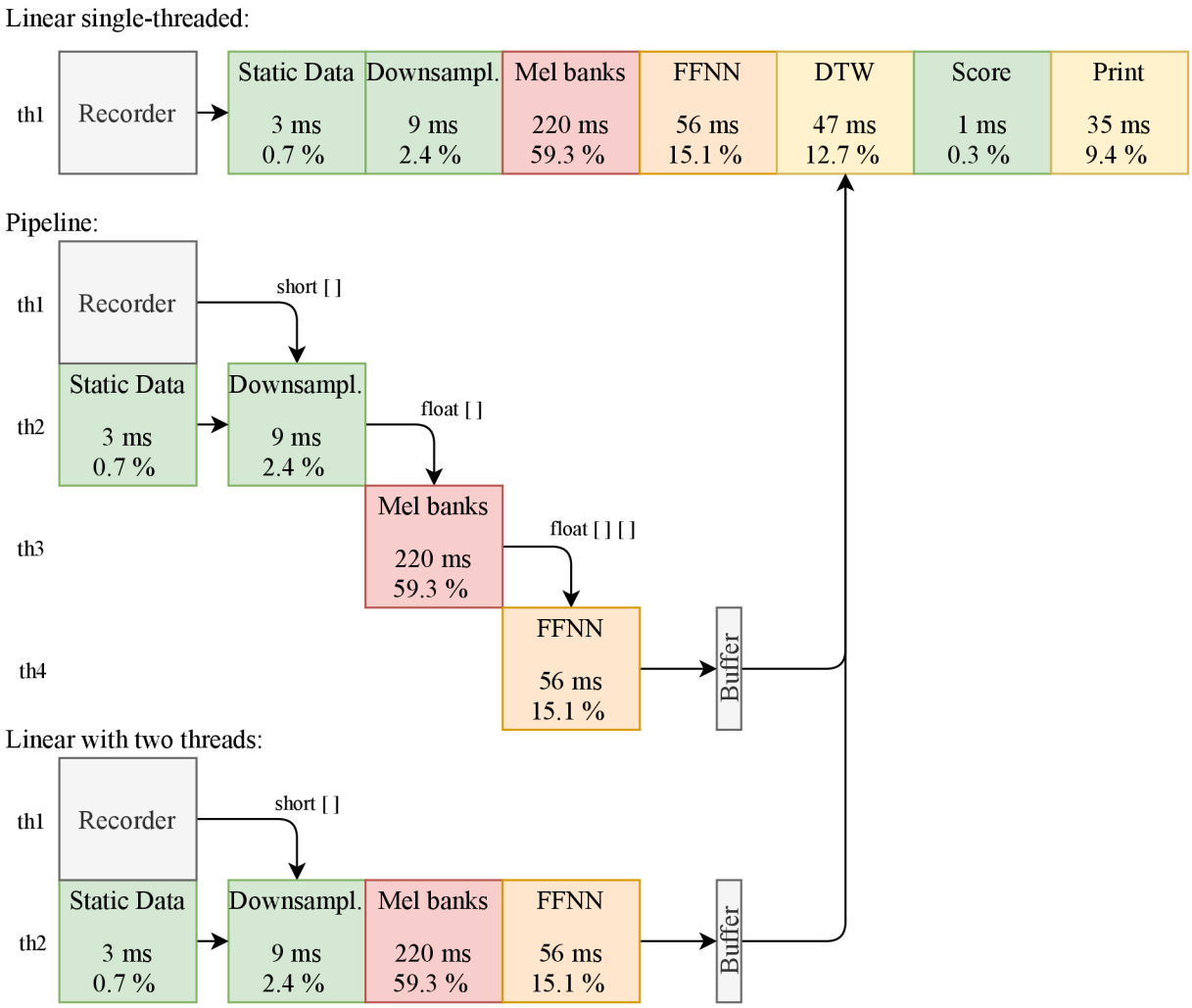

Figure 5.9: Baseline and two approaches to parallelization - pipelining or linear processing on background thread.

samples to send, the sender can close the channel, the receiver is resumed and can check whether the `isClosedForReceive` is set.

`AudioRecorder` is filling the internal buffer, and when the buffer is full, it provides an array of `short` samples. The buffer size depends on the sampling frequency, the number of audio channels and bit depth. This value can be obtained from `AudioRecord.getMinBufferSize()` function, and it can be for example 3 528 samples. The recorder puts the values one by one from the full buffer into the channel. The receiver takes values and fills its buffer. This buffer should have the same width as `AudioRecorder` or its multiples. In that case, the processing covers the time during which the next samples are recorded.

5.4.3 Profiling

What multiple of the buffer size to choose, is part of experiments. Whether the background thread finishes processing in time and the sum of processing times are the values included in table 5.8.

buffer size	execution time sum	blocks not finished in time
3 528	1 805 986	0-1
7 056	1 772 244	0-1
10 584	1 795 072	0

Table 5.8: Sum of parallel execution times (**Execute** part) using ***multiple evaluation*** on 10 s recording. Buffer size means the number of received samples from the recorder before invocation **Execute** in the background. It can be considered as a delay.

Based on the table, the size does not impact the sum of execution time and the number of blocks not finished in time. Not finished blocks were in the middle of processing, both in the case of 3 528 and 7 056 block size, and the following blocks were processed in time. Bigger blocks mean a longer time of processing the last block when the recording is finished. Thus, a smaller block size means faster processing from the user's point. The block size of 10 584 samples represent 0.24 ms of recording, and in the case of sampling frequency 44 100 Hz, so 5 s recording would be divided only into 21 blocks. We chose the block size to the buffer size. Bigger size requires more operation memory for computing, which is a limiting factor for smartphones.

Conclusion and comparison are in the following tables 5.9 and 5.10. The recording is not included as it is almost the same as the recording duration. The right table is from the user's point of view, **Prepare Static Data** is done during recording. Time of **Downsampling** is computed as the time of processing the data left in the buffer after finishing the recording. **Mel banks** time consists of the time of normalization and time of computing Mel banks for the data left in the buffer after finishing the recording.

5 s recording	**after TF Lite** $[ms]$	%	**with parallelization** $[ms]$	%
Load Engine Config	7.9	0.4	8.1	0.9
Load Task Config	658.2	36.1	57.9	6.1
Select Task Segment	808.9	44.3	659.6	69.7
Init AudioRecorder	33.0	1.8	34.4	3.6
Execute	316.3	17.3	186.1	19.7
$\sum$	1 824.3		946.0	
10 s recording	**after TF Lite** $[ms]$	%	**with parallelization** $[ms]$	%
Load Engine Config	7.6	0.2	7.5	0.3
Load Task Config	1 424.0	34.7	61.2	2.8
Select Task Segment	1 635.1	39.9	1 553.1	70.4
Init AudioRecorder	33.0	0.8	37.1	1.7
Execute	1 000.8	24.4	546.1	24.8
$\sum$	4 100.5		2 205.0	

Table 5.9: Profiling by comparing elapsed time without and with implemented parallelization. The evaluation was executed as the ***first run***.

For lack of time, the parallelization was not finished whole. Mel banks normalization and FFNN can be still parallelized, which would save another 90 ms. But even this solution reduced processing time by 45 % in the case of 5 s recording and by 55 % in 10 s recording.

Execute 5 s ***first run***	*ms*	%	**Execute 10 s** ***first run***	*ms*	%
Prepare Static Data	-	0.0	Prepare Static Data	-	0.0
Down-sampling	0.02	0.0	Down-sampling	0.01	0.0
Mel banks	2.81	1.4	Mel banks	3.71	0.7
FFNN	104.49	52.8	FFNN	178.11	31.5
DTW	62.68	31.7	DTW	329.67	58.4
Calculate Score	1.01	0.5	Calculate Score	1.66	0.3
Print JSON	26.91	13.6	Print JSON	51.55	9.1
$\sum$	197.92		$\sum$	564.72	

Table 5.10: Profiling with parallelization by comparing elapsed time. The time of all phases was computed after the recording was stopped and the user was waiting for results. **Prepare static data** is computed during recording. Thus, it does not affect processing time.

5.5 Improving the UI response time

The speech module was speedup in previous sections, but those changes did affect the client site. The user interface displayed the results still after 3 seconds, although the results were available in *Logcat*[14] much earlier.

It was caused by flooding the bottleneck of *React Native* called *React Native bridge*[15], visualized in figure 5.10. The bridge manages communication between JavaScript app controller and native modules (e.g., OS callbacks and API, *SpeechEngine* module).

React community is aware of this limitation, and they are currently working on the new architecture of a native module system called *TurboModules*[16].

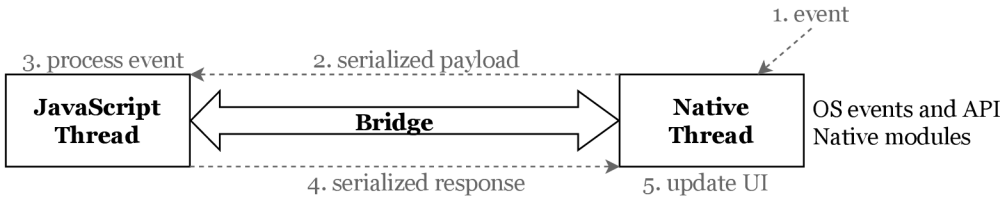

Figure 5.10: React native bridge with visualized event handling.

The actual problem was with the frequency of `onProgressChange` events sent from the *SpeechEngine* module into the *React Native*. The bridge was flooded by those events. Redrawing and responding to the user's touches became more and more delayed, and the response to the `onResults` event was processed and passed for rendering several seconds after the event occurred.

The frequency of reporting was reduced. The `onProgressChange` events were sent after processing 500 ms of data, then it was reduced to each 2 560 ms. After parallelization, the

[14]Logcat command-line tool - *Android Developers*, https://developer.android.com/studio/command-line/logcat

[15]Android Native Modules, https://reactnative.dev/docs/native-modules-android

[16]TurboModules proposal, https://github.com/react-native-community/discussions-and-proposals/issues/40

reporting was removed from background parallel parts and remained only in *FFNN* and later phases.

But still, when the recording and player are active at the same time, the UI is slow and delayed as both of the activities produces events for the UI.

I would not recommend *React Native* for real-time visualisations that need to be accurate and up-to-date.

5.6 Conclusion

In previous sections, we applied two main optimizations of the processing pipeline (computation closer to HW and parallelization) and one optimization of GUI rendering response. The processing pipeline of **Execute** phase summarizes along with processing times figure 5.11.

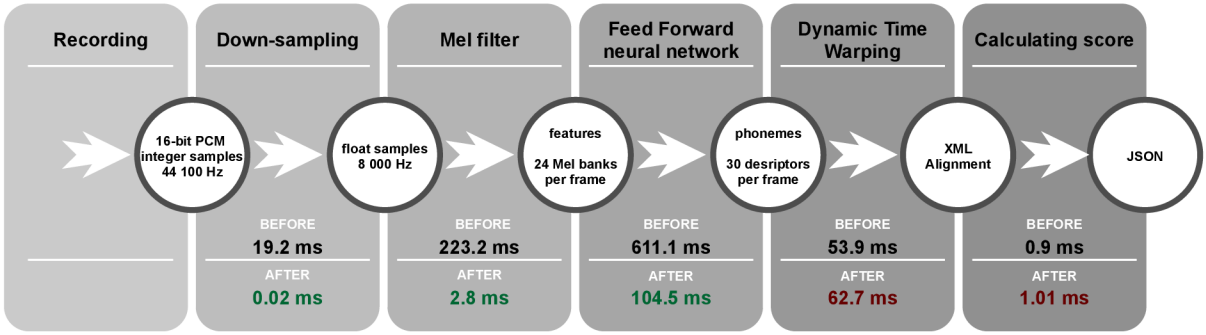

Figure 5.11: Comparison of profiling **Execute** pipeline before and after optimizations (from tables 5.1 and 5.10) on 5 s recording using the ***first run*** methodology.

The first optimization reduced loading and computing feed-forward neural network using high-level library *TensorFlow Lite*. This reduced **Load Engine Config** time and **FFNN** time in **Execute** phase (see tables 5.11 and 5.12). *TF Lite* uses CPU kernels optimized for the ARM Neon instruction set. Even better performance is achieved by computing 16 frames at once. It means that the input and output of this NN are 16 times bigger, but the NN model is executed 16 times less often.

The second optimization reduced **Prepare Static Data**, **Down-sampling**, and **Mel banks** times in **Execute** phase. Parallelization is implemented using *Kotlin Coroutines* and *Channels* from the *Kotlin Coroutines* library. **Execute** phase is invoked each time when `AudioRecorder` produces new samples on a background thread. The application needs only two threads for recording and processing, as processing is faster than recording. Figure 5.12 displays the implementation of the second approach (*Linear with two threads*).

The last optimization is not related to the evaluation module. It reduces the delay of the UI renderer. The number of state and progress updates sent by the evaluation module to the *React Native* UI was reduced more than five times. If progress updates come too often, *React Native* queues them, and the results are displayed after all previous progress updates are processed and displayed. With further optimizations, it will be necessary to further reduce, or even remove, reporting of processing progress.

The final architecture of the evaluation module is displayed in figure 4.2.

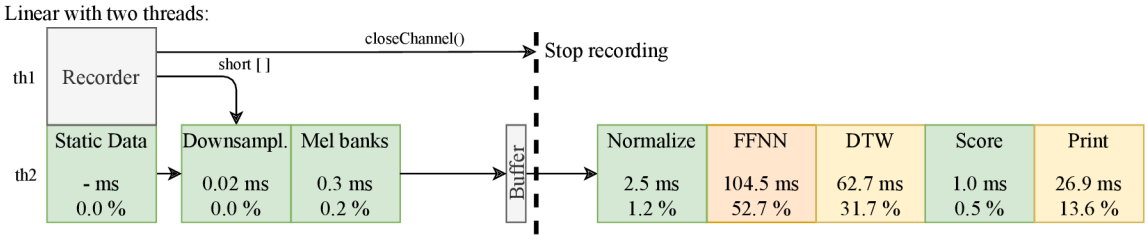

Figure 5.12: The final scheme of implemented parallelization. Unlike to 5.9, the **Mel banks** block was split into computing Mel banks and normalizing them. Due to this change, the FFNN block was not parallelized.

5 s recording	**before optimizations** $[ms]$	**after optimizations** $[ms]$
Load Engine Config	7 687.6	8.1
Load Task Config	629.7	57.9
Select Task Segment	812.7	659.6
Init AudioRecorder	33.0	34.4
Execute	1 123.3	186.1
$\sum$	**10 253.2**	**946.0**
10 s recording	**before optimizations** $[ms]$	**after optimizations** $[ms]$
Load Engine Config	7 677.3	7.5
Load Task Config	1 415.5	61.2
Select Task Segment	2 191.8	1 553.1
Init AudioRecorder	33.0	37.1
Execute	2 714.6	546.1
$\sum$	**13 999.2**	**2 205.0**

Table 5.11: Profiling by comparing elapsed time before and after implementing optimizations (from tables 5.7 and 5.10). The evaluation was executed using the ***first run*** methodology. The recording is not included as it is almost the same as the recording duration.

Execute 5 s ***first run***	**before optimizations** $[ms]$	**after optimizations** $[ms]$
Prepare Static Data	4.5	-
Down-sampling	19.2	0.02
Mel banks	223.2	2.81
FFNN	611.1	104.49
DTW	53.9	62.68
Calculate Score	0.9	1.01
Print JSON	10.6	26.91
$\sum$	**923.4**	**197.92**
Execute 10 s ***first run***	**before optimizations** $[ms]$	**after optimizations** $[ms]$
Prepare Static Data	8.3	-
Down-sampling	56.1	0.01
Mel banks	525.7	3.71
FFNN	1 451.9	178.11
DTW	329.2	329.67
Calculate Score	3.1	1.66
Print JSON	90.7	51.55
$\sum$	**2 465.0**	**564.72**

Table 5.12: Profiling by comparing elapsed time before and after implementing optimizations (from tables 5.2 and 5.10). **Prepare static data** is after the optimizations computed during recording. Thus, it does not affect the processing time.

Chapter 6

Conclusion

As part of this work, we explained the process of speech processing and comparing two speech recordings. We introduced, how the processing is used in the application and what is the expected functionality of the application. The largest part and contribution of this work is identifying parts that can be refactored and optimized. Those techniques can be used in similar applications as the presented processing pipeline is quite common in speech processing.

We used a high-level library *TensorFlow Lite* to access low-level functionalities provided by the hardware of today's smartphones. We tried different accelerators but finally, we stick to the single thread CPU variant as we do not have so complex neural network. We showed an example of a complex network and speed up achieved by delegating to the device GPU. Then we gained more speed up by computing more data at once and found the optimal data size.

As a second technique, we chose parallelization. We identified problematic blocks for parallelization and proposed two ways of parallelization. Implemented parallelization preserves the original accuracy and still brings 45 % speed up.

The last thing considered is the user interface. Processing may speed up, but if the user interface and the rendering is slow, the user cannot feel the acceleration techniques. After applying the techniques above, there is still needed 174 ms for processing. We can trickly hide this time into transitions or short animation, so the user cannot note any delay before the results.

We were able to reduce speech processing of 5 s recording from 928 ms to 174 ms at the same accuracy. Longer (10 seconds) recording originally takes 2.5 seconds, and at the end of the work, it takes only 0.5 seconds. Loading of neural network was reduced from 6 seconds to 8 milliseconds.

However, there are still opportunities to speed up the program. We can continue with parallelization and use uniform-n normalization (floating window with initial value or value from previous processing) and experiment with the size of windows and initial value in the context of accuracy. Dynamic time warping uses an algorithm that computes the whole graph. We can choose windows of limited width where we are looking for the best path. React Native can be still slowed down when there is multiple progress reported at the same time. We could try to compute the progress on the React side and send only start/stop/change events. The other option is discovering alternative solutions like *Flutter*, *Apache Cordova* or *Fuse*.

We are going to integrate this module into a real application and then try to apply the same methods on the different app (speech recognizer).

Bibliography

[1] Goldsborough, P. A Tour of TensorFlow. *CoRR*. 2016, abs/1610.01178. Available at: http://arxiv.org/abs/1610.01178.

[2] Honkela, A. Multilayer perceptrons. *Nonlinear Switching State-Space Models*. 2001. [cit. 2021-01-15]. Available at: https://users.ics.aalto.fi/ahonkela/dippa/node41.html.

[3] Müller, M. *Information Retrieval for Music and Motion*. Springer, 2007. ISBN 9783540740476.

[4] Szőke, I., Skácel, M., Černocký, J. and Burget, L. Coping with Channel Mismatch in Query-By-Example - BUT QUESST 2014. In: *Proceedings of 2015 IEEE International Conference on Acoustics, Speech and Signal Processing* [electronic, physical medium]. IEEE Signal Processing Society, April 2015, chap. 119899, p. 5838–5842. DOI: 10.1109/ICASSP.2015.7179091.

[5] Čuba, E. *Implementation of Simple Speech Recognizer in Android*. Brno, 2018. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Igor Szöke, Ph.D.

Appendix A

API Documentation

The following list contains functions and properties which can be used from the application and messages that are sent to the *React Native* application.

- Config options:
  - `LogVerbosity logVerbosity`
- Interface methods:
  - `loadTaskJSONConfig(jsonData: String)`
  - `initAudioRecorder(length: Int)`
  - `startAudioRecorder()`
  - `stopAudioRecorder()`
  - `selectTaskSegment(taskSegment: String)`
  - `startAudioPlayerRef(wrdId: Int)`
  - `stopAudioPlayerRef()`
  - `startAudioPlayerUsr(wrdId: Int)`
  - `stopAudioPlayerUsr()`
  - `speechEngineExecute()`
  - `saveAudioUsrAsWAV(filename: String)`
- Interface events (differs from Interface methods):
  - `onError(WritableMap payload)`
    - `String payload.error`
    - `Int payload.state`
  - `onProgressChanged(WritableMap payload)`
    - `Double payload.progress`
    - `Int payload.state`
  - `onStateChanged(WritableMap payload)`
    - `Int payload.state`
  - `onStatus(WritableMap payload)`

    * String payload.status
    * LogLevel payload.logLevel
- onResults(WritableMap payload)
    * String payload.results
- onAudioEnergy(WritableMap payload)
    * Double payload.audioEnergy
    * Double payload.progress

Appendix B

Configuration files

The module can be configured through configuration files `SpeechEngineConfig.json` and task specific configuration is stored in `TaskConfig.json` of each task. Those files are described by JSON Schema `Schema_SpeechEngineConfig.json` and `Schema_TaskConfig.json`. JSON schemas are not included here, as they are too long for a single page.

B.1 `SpeechEngineConfig.json`

```
{
  "SpeechEngine_StatusLevel": "INFO",
  "AudioRecorder_AudioEnergy_Refresh": 0.1,
  "AudioRecorder_AudioSamplingFrequency": 44100,
  "AudioRecorder_AudioChannels": "MONO",
  "AudioRecorder_AudioSource": "MIC",
  "AudioRecorder_AudioEncoding": "PCM_16BIT",
  "FeatureExtraction_Use": "default",
  "FeatureExtraction_List": [
    {
      "Version": "default",
      "NeuralNet_BinFile": "/storage/emulated/0/sewrapperdemo/SpeechEngine/model16x.tflite"
    }
  ]
}
```

B.2 `Task1Config.json`

```
{
  "Task_Transcript_XML": "/storage/emulated/0/sewrapperdemo/Task1/0000000audio.xml",
  "Task_Audio_Reference_WAV": "/storage/emulated/0/sewrapperdemo/Task1/0000000audio.wav",
  "Task_Audio_User_WAV": "/storage/emulated/0/sewrapperdemo/Task1/user_audio.wav",
  "Task_Features_List": [
    {
      "Version": "default",
      "FileName": "/storage/emulated/0/sewrapperdemo/Task1/0000000audio.bnfea"
    }
  ]
}
```