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

IMPLEMENTATION OF SIMPLE SPEECH RECOGNIZER IN ANDROID

IMPLEMENTACE JEDNODUCHÉHO ROZPOZNÁVAČE ŘEČI PRO ANDROID

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR — **EDUARD ČUBA**
AUTOR PRÁCE

SUPERVISOR — **Ing. IGOR SZÖKE, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2018

Zadání bakalářské práce/19075/2017/xcubae00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií | Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Čuba Eduard**
Obor: Informační technologie
Téma: **Implementace jednoduchého rozpoznávače řeči pro Android**
Implementation of Simple Speech Recognizer in Android
Kategorie: Softwarové inženýrství

Pokyny:

1. Nastudujte základy implementace aplikací pro Android (SDK, NDK) a základy rozpoznávání řeči.
2. Implementuje jednoduchý rozpoznávač řeči (extrakce příznaků, forward pass přes neuronovou síť, dekodér). Pokud to bude možné, využijte dostupných knihoven.
3. Tam kde to půjde, využijte low level implementace pro urychlení (NDK, RenderSript).
4. Vyhodnoťte úspěšnost a hlavně časové a paměťové nároky. Navrhněte směry dalšího vývoje.
5. Vyrobte A2 plakátek a cca 30 vteřinové video prezentující výsledky vaší práce.

Literatura:

- Dle pokynů vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- Bod 1 a část bodů 2 a 3 ze zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Szőke Igor, Ing., Ph.D.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
L.S. 612 66 Brno, Božetěchova 2

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstract

The goal of this project is to implement speech recognition software for Android platform. This paper outlines fundamental components of a speech recognizer and reviews the techniques used to optimize the process of speech recognition on Android devices. Firstly, it examines the implementation of the acoustic feature extraction and phoneme estimation processes. Then, it describes the design and implementation of a decoder used to process phoneme estimations into transcription, utilizing only limited resources of a mobile device. The project is divided into several modules, forming an Android library, which should be easy to extend and can be provided with custom models tailored for the desired use. Later, this paper discloses various approaches to modeling abstract data structures for recognition network representation, as well, as the ways of further development and applications of this project.

Abstrakt

Cieľom projektu je vytvoriť jednoduchý rozpoznávač reči pre platformu Android. Práca rozoberá základné komponenty rozpoznávača reči a venuje sa technikám, ktoré boli použité pre optimalizáciu procesu rozpoznávača reči na zariadeniach so systémom Anrdoid. Ako prvá je popísaná teória extrakcie akustických príznakov, odhadu posteriórnych pravdepodobností fonémov a dynamického dekódovania. Následnej je popísaný dizajn a implementácia dekodéra, ktorý prevádza sériu rečových príznakov na text, za použitia obmedzených výpočetných prostriedkov mobilného zariadenia. Implementácia je rozdelená do modulov tvoriacich knižnicu, ktorú je možno jednoducho rozšíriť, či integrovať do požadovanej aplikácie. Do rozpoznávača je taktiež možné dodať vlastné modely, ktoré môžu byť navrhnuté a natrénované pre konkrétne použitie. V experimentoch sme skúmali rôzne prístupy ku modelovaniu abstraktných dátových štruktúr pre reprezentáciu rozpoznávacej siete tak. V závere práca rozoberá potencionálne smery budúceho vývoja a aplikácií tohoto projektu.

Keywords

speech recognition, dynamic decoder, Android, NDK

Kľúčové slová

rozpoznávanie reči, dynamický dekodér, Android, NDK

Reference

ČUBA, Eduard. *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.

Rozšírený abstrakt

Rozpoznávanie reči je v dnešnej dobe jednou z najviac diskutovaných tém v oblasti informatiky. Navzdory stále rastúcemu výkonu mobilných zariadení sa však väčšina rozpoznávania reči odohráva na vzdialených, často špecializovaných serveroch. Možnosti pre získanie prístupu k modifikovateľnému kontinuálnemu rozpoznávaniu reči na mobilných zariadeniach bez použitia internetu sú značne obmedzené. Jedným z hlavných problémov je nedostatok operačnej pamäte, ktorá je potrebná pre beh dekodéra pracujúceho so statickou rozpoznávaciou sieťou. Táto práca sa venuje návrhu a implementácii kompletného rozpoznávača reči, optimalizovaného pre mobilné zariadenia s operačným systémom Android. Kvôli vysokej výpočetnej náročnosti rozpoznávania reči je celý proces vykonávaný v natívnom prostredí. Rozpoznávač reči je vytvorený v jazyku C++ a preložený do nízko-úrovňového kódu pre cieľovú architektúru. Výstupom práce je knižnica, ktorú je možné integrovať do vlastných aplikácii a osadiť ju modelmi vytvorenými pre špecifické použitie. Veľká časť knižnice je prenositeľná a po doplnení rozhrania pre nahrávanie zvuku, použiteľná na rôznych platformách.

V prvom rade práca rozoberá problematiku extrakcie akustických príznakov, ktoré sú následne použité pre akustické modelovanie. Použité príznaky sú vo forme Mel filtrov. Nad výstupmi z filtrov sme navrhli a implementovali dynamickú normalizáciu, ktorá umožňuje extrakciu príznakov v reálnom čase.

Použité akustické modely sú založené na pravdepodobnostiach stavov foném. V priebehu práce sme používali jedno-stavové fonémy, v rámci experimentov a ďalšieho smerovania práce sme však experimentovali aj s troj-stavovými fonémami. Odhad posteriórnych pravdepodobností foném je vykonávaný pomocou doprednej umelej neurónovej siete. Vstupom do siete sú posuvné okná cez hodnoty jednotlivých filtrov. Pre extrakciu posteriórnych pravdepodobností jedno-stavových foném sme používali neurónovú sieť s úzkym hrdlom, ktorá mala 500 až 1200 perceptrónov v skrytej vrstve. Na výpočty spojené s neurónovou sieťou sme použili dostupné knižnice pre lineárnu algebru s využitím vektorizačných techník pre urýchlenie výpočtov. Zároveň sú výpočty paralelizované na niekoľko výpočetných jadier. Experimentovali sme s rôznymi veľkosťami neurónových sietí, sledovali ich výpočetnú náročnosť pre použitie v reálnom čase a vplyv ich veľkosti na presnosť výsledkov rozpoznávania.

Ako podklad pre dekódovanie sú použité pravdepodobnosti foném v rečových rámcoch – posteriórne pravdepodobnosti v rečových segmentoch upravené o apriórne pravdepodobnosti jednotlivých foném. Vysokú priestorovú zložitosť dekódovania sme vyriešili použitím takzvaného dynamického dekodéra, ktorý na rozdiel od bežných (statických) dekodérov používa len malú, uni-gramovú rozpoznávaciu sieť. Pravdepodobnosti slovných sekvencií (n-gramov) sú dohľadávané či dopočítavané dynamicky, v dobe dekódovania. Najdôležitejšími časťami implementácie boli reprezentácia uni-gramovej siete, algoritmus hľadania najlepšej cesty rozpoznávaciou sieťou a reprezentácia úložiska n-gramov. Vo všetkých týchto častiach bola použitá optimalizovaná, vlastná implementácia, ktorú sme porovnali s kontajnermi a šablónami zo základnej knižnice jazyka na základe pamäťovej a výpočetnej náročnosti. Proces dekódovania je paralelizovaný a poskytuje podporu pre obmedzené, aj väčšie – konverzačné modely jazyka.

Pre zníženie veľkosti a času načítania modelov boli navrhnuté špecializované binárne formáty vhodné pre doprednú neurónovú sieť a jazykový model používaný dynamickým dekodérom, ktorý je rozdelený na uni-gramovú sieť a úložisko n-gramov. Čas načítania a potrebnú pamäť sme porovnali s pôvodnými, textovými reprezentáciami.

Výsledky rozpoznávania reči boli zhodnotené na nahrávkach z konferenčných prednášok a porovnávané s výsledkami statického dekodéra s použitím rovnakých rečových príznakov. Knižnica, spolu s ukážkovým aplikačným rozhraním je použitá v demonštračnej aplikácii, ktorá umožňuje živé testovanie za použitia rôznych jazykových modelov.

Výstupom z tejto práce je plne funkčná knižnica pre rozpoznávanie reči, optimalizovaná pre použitie na mobilných zariadeniach. S využitím aktuálnych telefónov strednej triedy s operačným systémom Android bolo možné pracovať v reálnom čase s modelom veľkosti 12500 slov, 1.69 milióna bi-gramov a 1.91 milióna tri-gramov. Práca, spolu s výsledkami a živou ukážkou bola prezentovaná a ocenená na študentskej konferencii Excel@FIT 2018.

Implementation of Simple Speech Recognizer in Android

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Ing. Igor Szöke Phd. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . .
Eduard Čuba
May 15, 2018

Acknowledgements

I would like to thank my supervisor Ing. Igor Szőke, Ph.D. for his help.

Contents

1 Introduction **3**

2 Speech recognition **4**
2.1 Speech recognition on Android platform . . . 4
2.2 Audio processing . . . 5
2.2.1 Preprocessing . . . 5
2.2.2 Spectral analysis . . . 9
2.2.3 Voice activity detection . . . 10
2.2.4 Mel filter banks . . . 11
2.2.5 Mean normalization . . . 13
2.3 Phoneme posteriors estimation . . . 14
2.3.1 Perceptron . . . 14
2.3.2 Feed-forward neural network . . . 15
2.3.3 Feature stacking . . . 15
2.3.4 Feature post-processing . . . 16
2.4 Decoder . . . 17
2.4.1 The task of decoding . . . 17
2.4.2 Acoustic model . . . 18
2.4.3 Language model . . . 18
2.4.4 Token passing . . . 19
2.4.5 Decoding process . . . 20
2.5 Models . . . 21
2.5.1 Neural network for phoneme posterior probabilities . . . 21
2.5.2 Acoustic model and uni-gram network . . . 22
2.5.3 N-gram model representation . . . 23

3 Android specific implementation **24**
3.1 Implementation goals . . . 24
3.2 Library architecture . . . 24
3.3 Recorder module . . . 26
3.3.1 Audio recording . . . 26
3.3.2 Audio feature extraction . . . 26
3.4 Acoustic feature extraction module . . . 28
3.4.1 Phoneme posteriors estimation performance . . . 28
3.4.2 Phoneme posteriors estimation parallelization . . . 29
3.5 Decoder module . . . 29
3.5.1 Uni-gram network representation . . . 29
3.5.2 Token structure . . . 31

3.5.3 Word link record . . . 31
3.5.4 Token passing . . . 32
3.5.5 N-gram storage implementation . . . 35
3.5.6 Decoder parallelization . . . 36
3.6 Binary representation of the models . . . 36
3.6.1 Binary representation of a neural network . . . 37
3.6.2 Uni-gram model binary representation . . . 37
3.6.3 *N*-gram model binary representation . . . 38
3.7 Implementation summary . . . 38

4 Evaluation and testing **39**
4.1 Evaluating the results . . . 39
4.1.1 Test set . . . 39
4.1.2 Test results . . . 39
4.1.3 Comparison with a static decoder . . . 40
4.1.4 Three-state phonemes . . . 40
4.2 Demo application . . . 41
4.3 Unit testing . . . 42

5 Conclusion **44**
5.1 Further work . . . 44

Bibliography **45**

Chapter 1

Introduction

The domain of automatic speech recognition is unquestionably one of the most discussed topics in the world of computer science. New technologies, the expanding capabilities of smartphones, intelligent assistants and wearable devices are unveiling and delivering many use-cases, which have not been possible in the past. However, most of the speech recognition is still performed in the cloud by powerful and often specialized hardware. Thus, the possibilities of getting a customizable, real-time speech recognition for a specific use-case, without an internet connection and paid cloud APIs, are substantially limited. The goal of this work is to create a portable, easy-to-use library, which provides end-to-end access to speech recognition on Android platform and empowers the user to use custom acoustic and language models.

The main problem is the limited amount of resources available in popular Android phones. For this reason, real-time recognition must be run within several hundred megabytes of operating memory using a reasonable amount of CPU time, not significantly affecting the functionality of a device. Therefore, it is necessary to introduce various optimizations, including a so-called dynamic decoder and voice activity detection – to suspend demanding phoneme posteriors estimation and decoding processes whenever possible. Due to these limitations, the decoder will work with a limited language model, specially designed for desired use. For example, such model can contain a subset of conversational language for phone calls transcription or number recognition.

Firstly, Chapter 2 explains the theoretical background of the work along with used methods and models. Secondly, in Chapter 3 we reviewed the implementation part, outlined the used optimizations and compared the performance and memory requirements of proposed methods. Finally, Chapter 4 reveals the results achieved using the recognition system, the way of evaluating the results, sample library integration and introduced test techniques.

Chapter 2

Speech recognition

Generally speaking, the task of the speech recognizer is to provide a transcription of word sequence uttered in an audio signal. Although the task is clear, the fulfillment is not that straightforward. For the most part, the speech signal might be distorted by several negative factors, including background noise, dialect or tone. As a result, defining a single model how a signal for a word or sentence should look is not possible, which leads us to use a statistical approach. Usually, the recognition is based on phoneme occurrence probabilities in small speech segments called frames. This part of the decoder is called an acoustic model. One level higher, we have to estimate the words represented by a sequence of phoneme probabilities and struggle with language problems including homonyms and unclear word boundaries. These problems establish a need of having a language model, which defines probabilities of various word sequences.

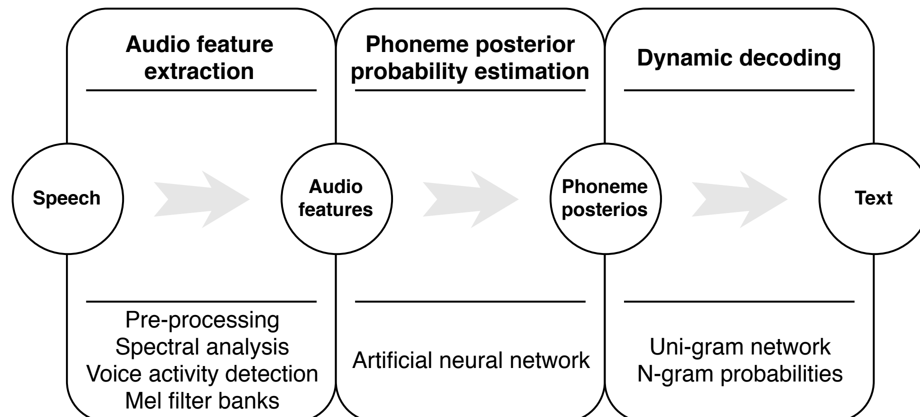

Figure 2.1: Simplified block scheme of automatic speech recognition based on statistical language model and single state phoneme posterior probabilities extracted by an artificial neural network.

2.1 Speech recognition on Android platform

One of the major issues holding back the expansion of ASR applications in everyday devices is the lack of operating memory, which is required to hold relatively big language model. Although Android devices these days provide a lot of computing power, the size of operating memory available is usually quite limited. The problem can be partially resolved, by using

a dynamic decoder, which is constructing recognition network on the fly [9], at the price of higher CPU utilization. From the existing solutions, it is worth mentioning the Android internal speech recognizer, that is able to work in the offline mode. However, it is more suitable for voice commands rather than continual recognition, and it is not possible to equip it with custom models. On the other hand, there is an open-source alternative PocketSphinx[1], a lightweight implementation of CMUSphinx toolkit. Still, being quite complex and complicated makes it hard to use for not involved developers, and it is mostly used with limited vocabulary models for application control purposes.

In this project, we will focus on the implementation of a simple speech recognizer with all the mandatory components for having reasonable recognition results. The acoustic model is based on a feed-forward neural network, which might be trained for the desired purpose. Similarly, we will take a statistical approach to language modeling, empowering the user to deliver his own language model based on recordings and transcriptions from the target area. All the core components are implemented in a portable way, making it possible to deploy the library to any platform, which provides sufficient amount of resources for the aspired use-case.

With this in mind, the project might be a valuable resource for mobile and embedded developers seeking for speech recognition engine to integrate with their applications.

2.2 Audio processing

This section discusses the steps performed on a speech recording in order to prepare data for further processing. The goal of audio processing is to get an approximation of frequencies involved in a speech over time. Consequently, this approximation is used for the acoustic feature extraction and phoneme posterior estimation.

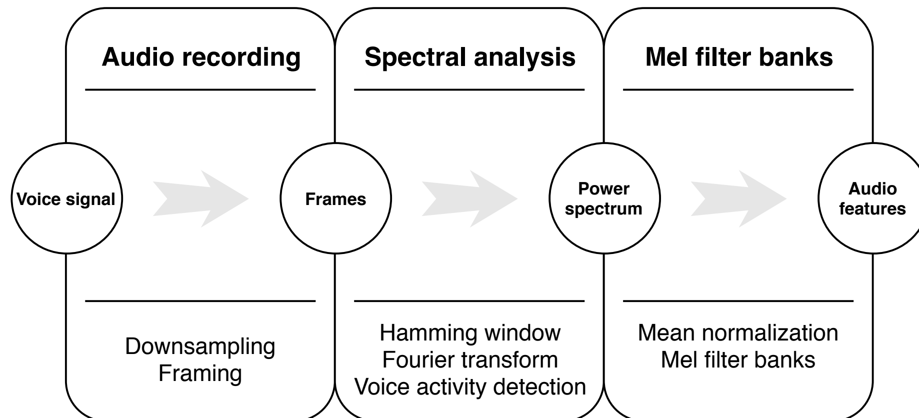

Figure 2.2: A block scheme of the audio feature extraction with Mel filter bank values as features.

2.2.1 Preprocessing

The very first step of speech recognition itself is to obtain a source recording. Then, the recording is sub-sampled to a sample rate suitable for speech recognition and divided into

[1]https://cmusphinx.github.io/

shorter frames. Subsequently, a window function is applied to enhance the results of spectral analysis by reducing the ripples on the window borders.

Pulse code modulation format

Speech is recorded in pulse code modulation (PCM) format. Firstly, speech waves are periodically sampled. Amplitudes of particular samples are represented by discrete values – usually 16-bit long integers. Accordingly, amplitude can be expressed on scale of $2^{16} = 65536$ levels representing range of approximately 96 dB [3].

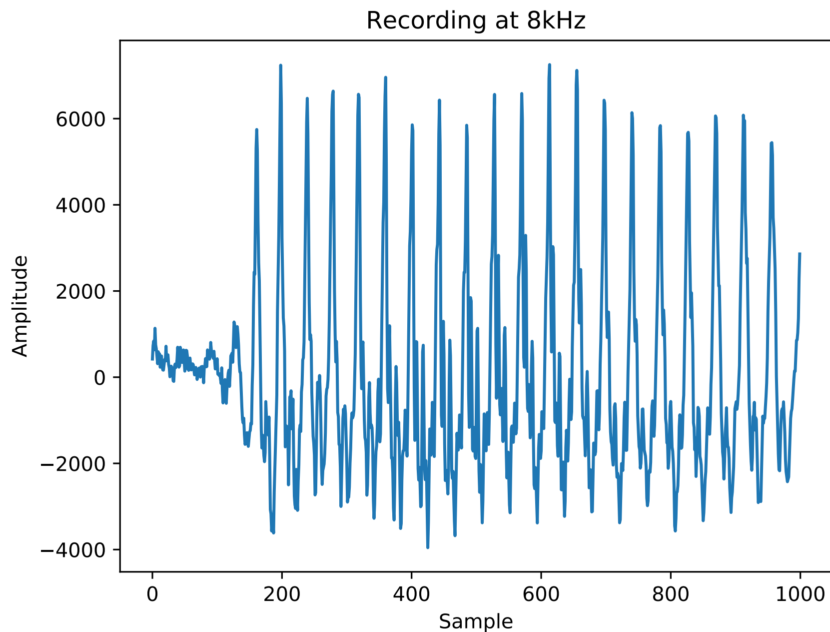

Figure 2.3: A speech segment in PCM format.

Sub-sampling

Frequencies involved in human speech are usually above 50 Hz, while most of the energy is concentrated between 300 Hz and 3000 Hz. For the speech recognition system, we use a frequency band from 64 Hz to 3800 Hz. The sampling frequency, according to the Nyquist theorem, has to be at least two times higher than the highest listed frequency. That makes the sample rate 8000 Hz a suitable choice between accuracy and performance. Although, input data may be sampled at a different sample rate since the only guaranteed sample rate on the Android devices is 44 100 Hz. Most of the Android devices, however, support recording at many different sample rates using an internal resampler. As of Android 5.0 (Lollipop), the audio resamplers are entirely based on FIR filters[2], but although the 8 kHz sample rate is supported by the most of Android devices, the actual list of supported sample rates is vendor-specific. Therefore, in order to support all the devices, it is necessary to provide a recorder with a custom resampler, including an appropriate low-pass filter to avoid aliases.

[2] https://developer.android.com/ndk/guides/audio/sampling-audio.html

Framing

To get the estimation of frequencies involved in a speech over time, the recording has to be cut to shorter frames, on which a spectral analysis will be performed. A common approach to audio framing for speech recognition is to use 25 ms long frames with 10 ms step. Consequently, a single frame will contain 10 ms of recorded samples and 15 ms of samples from the previous frame.

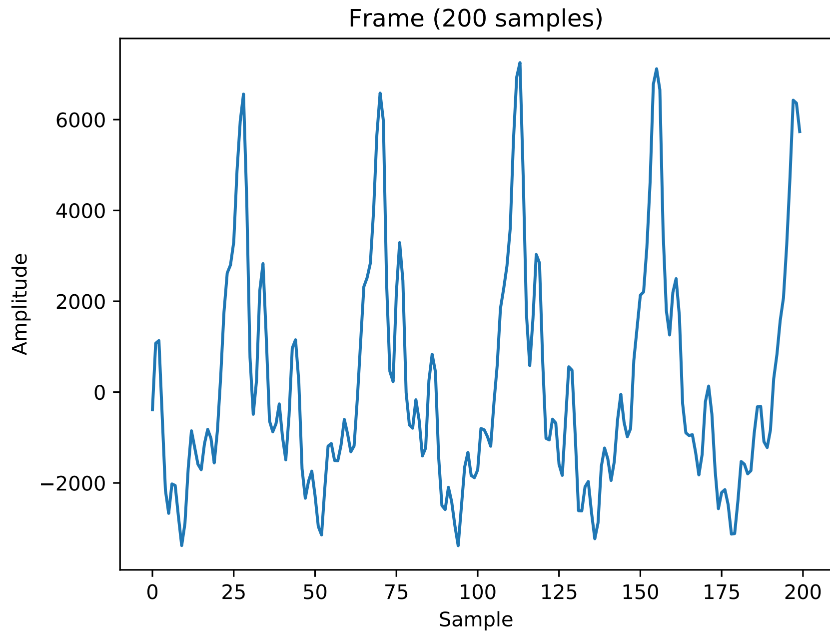

Figure 2.4: A single 200 sample long speech frame in PCM format.

While using a sample rate of 8000 Hz, the length of a single sample is $\frac{1000}{8000} = 0.125$ ms. 25ms long frame contains $\frac{25}{0.125} = 200$ samples. Thus the length of the step in samples is therefore $\frac{10}{0.125} = 80$.

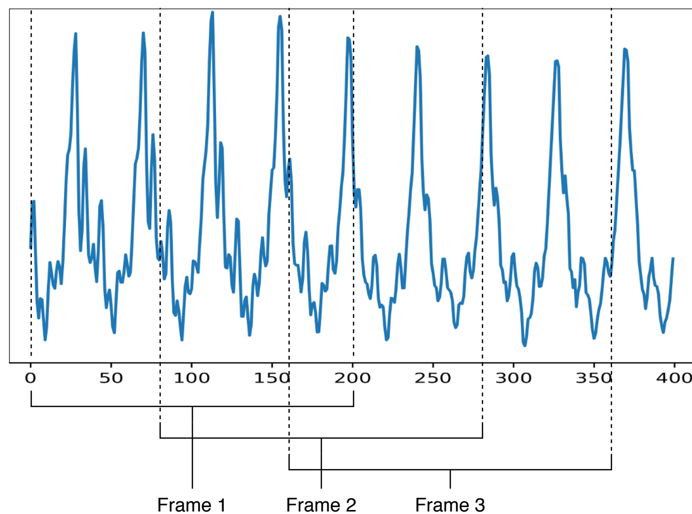

Figure 2.5: Framing recording into particular frames with step 80 samples.

Hamming window

Cutting down a signal to frames may cause, that a ripple will be cut off at its peak, possibly compromising the results of the spectral analysis. For this reason, it is necessary to apply appropriate window function. In this work, we use Hamming window function defined by Formula 2.1.

$$f_h[n] = f[n](0.54 - 0.46\cos(\frac{2\pi n}{N-1})) \tag{2.1}$$

Where:

N is the length of a frame (200)

n is the index of a single sample

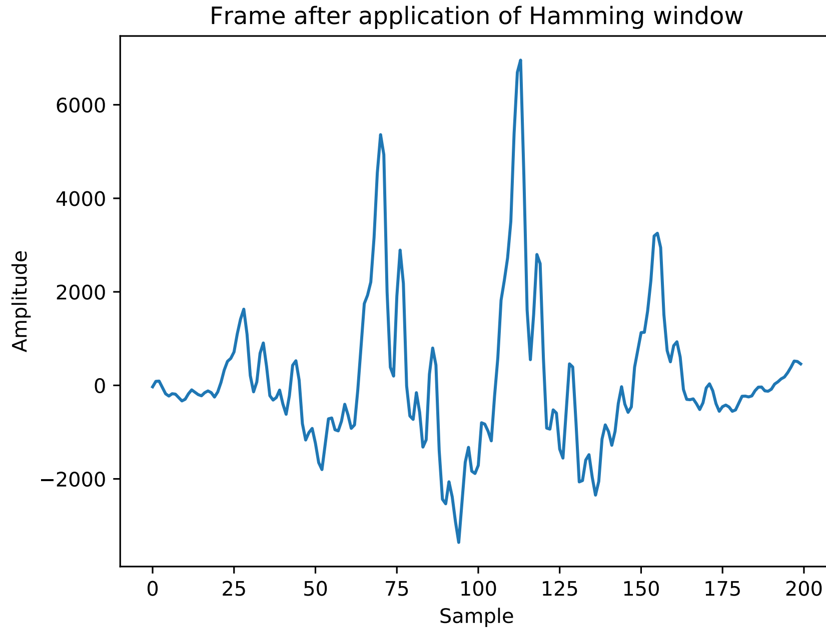

Figure 2.6: The speech frame after application of Hamming windows function.

2.2.2 Spectral analysis

Spectral analysis provides an insight into frequency domain of particular speech frames. Later on, the frequency spectrums involved in speech frames are used for audio feature extraction.

Fast Fourier transform

In general, the goal of the spectral analysis is a conversion between time and frequency domain of the signal. Whereas the input signal is not continuous but rather discrete, the transformation of the frequency spectrums is to be performed using Discrete Fourier transform (DFT). Ordinary DFT algorithm defined by Formula 2.2 is not regularly used due to its complexity of N^2 steps [1]. Therefore, optimized variants of DFT collectively called Fast Fourier Transform are used.

$$D(n) = \sum_{k=0}^{N-1} d(k)e^{-\frac{ink2\pi}{N}} \qquad n \in [0, N-1] \tag{2.2}$$

Usually, these optimized variants take advantage of periodicity and symmetry of the DFT algorithm to reduce its computational complexity. N-point DFT is decomposed into r, $\frac{N}{r}$-point DFTs, where r is the radix of FFT. Most of the implementations work with radix 2 or/and 4, what leads to a requirement for transformation length N to be a power of 2 (and 4 for radix 4).

Power spectrum

Fundamental frequencies in a frame can be accentuated by computing the power spectrum. The power spectrum is computed as the square of the magnitude of a complex magnitude, as follows:

$$P_n = \sqrt{x_n^2 + i_n^2}^2 = x_n^2 + i_n^2 \tag{2.3}$$

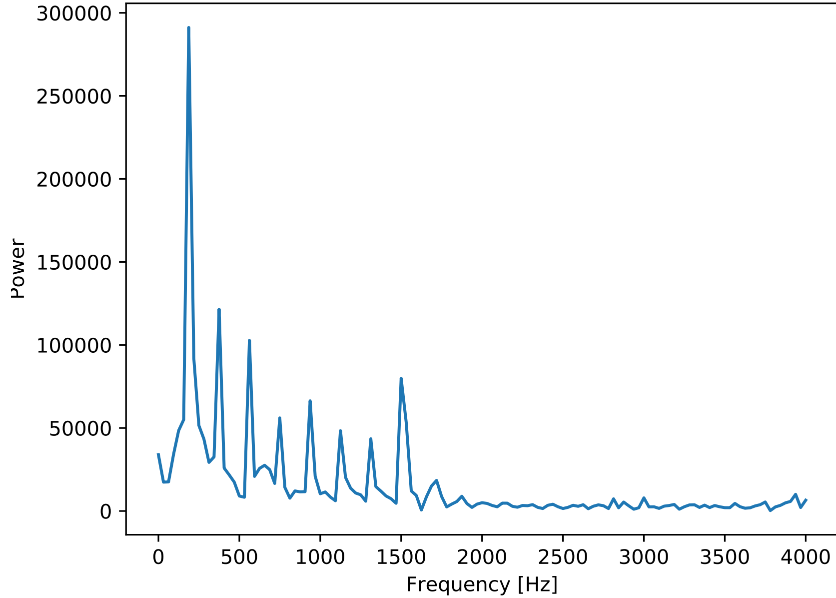

Figure 2.7: The power spectrum of the speech frame.

2.2.3 Voice activity detection

In order to spare the resources of the device and enhance the results of mean normalization, we use simple voice/silence detection. If the recorder is in the silent state, it is possible to suspend both phoneme posteriors estimation and decoding, keeping CPU utilization by the recognizer close to none.

This detection is performed by computing the energy of each frame and comparing it against a statistically chosen threshold. After a silent frame series of the specified length, the recognizer enters a suspended mode, in which incoming frames are stored in a queue of restricted length. Thus, the queue is used for smoothing the transition to the active state. Since the threshold might be crossed by short, but intensive sounds like clapping or typing on a keyboard, it is beneficial to introduce an activation smoothing mechanism. The recorder becomes active only after series of frames classified as a speech. If the series is long enough, the speech is considered confirmed and the most recent frame is passed to further processing along with preceding frames from the queue. The length of the queue is determined by the width of the rolling window on feature extraction model input, increased by the number of frames required for speech activation. Figure 2.8 shows the histogram of

energies involved in the speech. According to this histogram, it is possible to estimate the appropriate threshold for the voice activity detection.

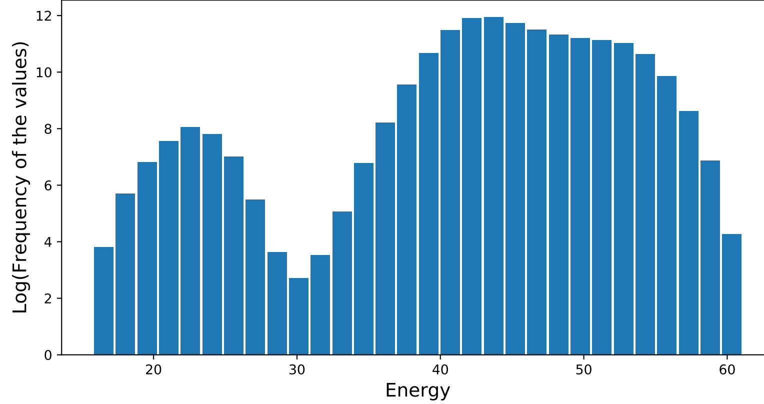

Figure 2.8: A histogram of energies involved in a sample recording. The histogram shows, that silence (on the left) might be separated by appropriate threshold (e.g. 30).

2.2.4 Mel filter banks

The idea behind using Mel frequency scaling is to simulate how the human ear works, having a better resolution at lower frequencies. Conversion between Hertz and Mel is done using Formula 2.4 and Formula 2.5.

$$m = 1127 \log(1 + \frac{f}{700}) \tag{2.4}$$

$$f = 700(e^{\frac{m}{1127}} - 1) \tag{2.5}$$

The power spectrum is split into 26 overlapping bands, on which 24 triangular filters called banks are applied. The first bank starts at the first point, reaches its peak at the second point, and ends at the third point. The next one starts at the second point, reaches its peak at the third point, ends at the fourth point and so on.

The first bank starts at the lowest desired frequency – e.g. 64 Hz = 98.6 Mel. Here, the last one ends on Nyquist frequency for given sample rate – e.g. 3800 Hz = 2097.07 Mel. Accordingly, the other banks are linearly spaced in between on the Mel scale. Mel values are transformed back to frequencies and used to compute particular power spectrum vector bins. Let the power spectrum be stored in an array of length N, then a particular bin's value can be estimated using Formula 2.6.

$$n = \lfloor \frac{f * N}{F_s} \rfloor \tag{2.6}$$

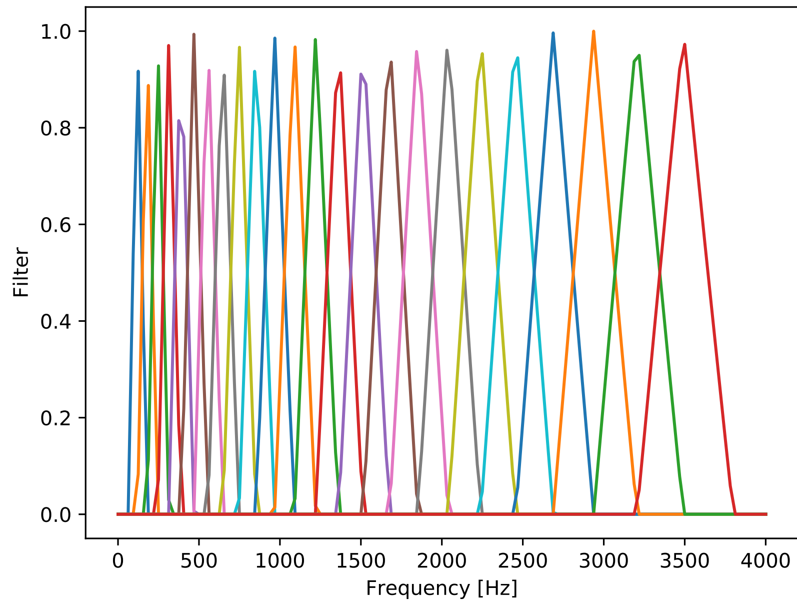

Figure 2.9: 24 triangular Mel filter banks.

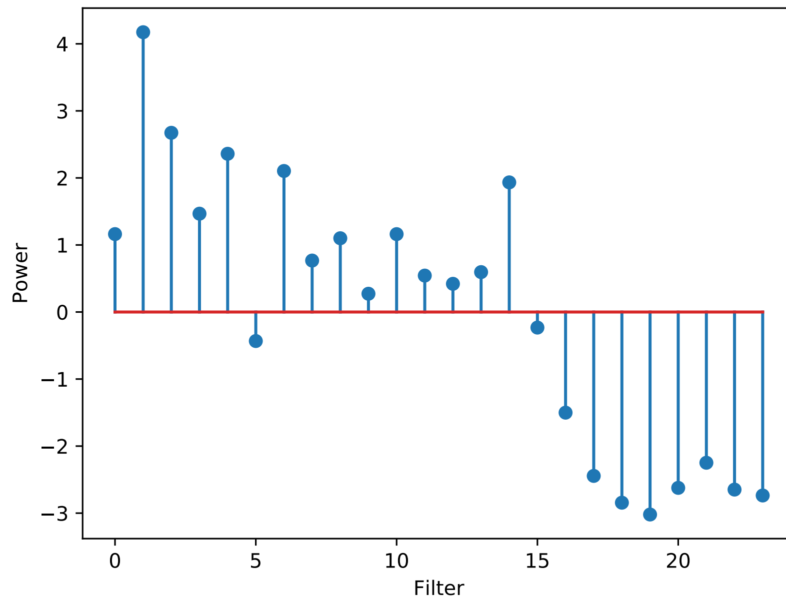

Figure 2.10: Mean 24 Mel filter bank values of the frame.

The values of triangular filter banks for particular frequencies of the power spectrum are defined Formula 2.7.

$$H_m(k) = \begin{cases} 0 & k < f(m-1) \\ \frac{k-f(m-1)}{f(m)-f(m-1)} & f(m-1) \leq k \leq f(m) \\ \frac{f(m+1)-k}{f(m+1)-f(m)} & f(m) \leq k \leq f(m+1) \\ 0 & k > f(m+1) \end{cases} \tag{2.7}$$

Where:

k is an index of power spectrum bin, $k \in [O, N]$

N is the length of the power spectrum

M is count of Mel banks(24)

f is an array of $M + 2$ bins

m is an index of a filter bank

In the end, filter banks for a single frame are summed and logarithmically transformed.

$$B[m] = \log \sum_{k=0}^{N} H_m(k) \tag{2.8}$$

2.2.5 Mean normalization

To find out whether specific Mel banks are significant in particular frames, their values have to be normalized – computing the difference between bank's mean and its current value. When performing offline feature extraction, it can be simply done by computing the mean value for each filter bank and subtracting it from samples. Obviously, that is not possible with online feature extraction. Mean value has to be computed over time, what might make first few hundred samples notably malformed. Consequently, longer recordings can also suffer from distortion caused by a change of background noise over time (e.g. change of environment during a phone call). Therefore, we will introduce more advanced normalization techniques.

Uniform last-N normalization

The simplest way of adaptation to contemporary background noise is the limitation of samples taken into consideration when computing a filter bank mean value. Admittedly, the problem with this method is choosing N. The accuracy grows with N, on the other hand, large N will cause the longer period of malformed values after a change of the background noise.

Exponential normalization

In contrast, exponential function $f(x) = b^x$ with base $b \in (0, 1)$ can be used to restrain the negative impact of older frames by decreasing their weight over time. Similarly, this method requires the choice of suitable base parameter. For example, if a 5 second old sample should be included with weight 0.5, using sample rate 8000 Hz, and frame step 80

samples, then the base parameter can be expressed as $b = \sqrt[5*\frac{8000}{80}]{0.5} = \sqrt[500]{0.5}$. Generally, mean value M for bank m for sample $n+1$ with base b might be expressed by Formula 2.9 and Formula 2.10.

$$S_{n+1}[m] = B_{n+1}[m] + bS_n[m] \tag{2.9}$$

$$M_{n+1}[m] = \frac{S_{n+1}[m]}{\int_0^\infty b^x dx} \tag{2.10}$$

Formula 2.9 computes a weighted sum of the samples in the new time step. Subsequently, Formula 2.10 computes new mean value M by dividing the sample sum by the total weight represented by the area under the exponential curve of $f(x) = b^x$ from zero (new sample) to infinity.

2.3 Phoneme posteriors estimation

The process of phoneme posteriors estimation describes the transformation of audio features retrieved from the speech signal to the posterior phoneme probabilities used for language modeling. In this work, we use posterior probabilities of single-state or three-state phonemes involved in particular speech frames. The estimation model is defined by a feed-forward artificial neural network (ANN), that might be trained for the task-specific purpose. We use split context ANN [8] with a bottle-neck inside [2]. In this section, we will briefly explain how phoneme posteriors estimation works and introduce models used for this purpose.

2.3.1 Perceptron

Artificial neural networks are computational models composed from smaller units – mathematical models of a neuron called perceptrons [7]. Perceptron is a simple model which takes a set of inputs and propagate level of its activation to the output. Firstly, the inputs are multiplied by their weights, and the sum of their products is supplied to the input of an activation function. The activation function can either be a threshold or a logistic function. Here, we used the logistic sigmoid function, therefore, the model is called sigmoid perceptron. Furthermore, perceptrons often have a special input called the bias with a constant value of 1, and its own weight. This value is included in the weighted sum of the inputs what allows to modify the behavior of activation function. Correspondingly, the relationship between input vector x and activation rate a is shown in Formula 2.11. For an illustration, a graphical representation of a perceptron is shown in Figure 2.11.

$$a = f(\sum_{i=0}^{n} w_i x_i) \tag{2.11}$$

Where:

- a is the activation rate
- x is the input vector of length n
- w is the weight vector of length n

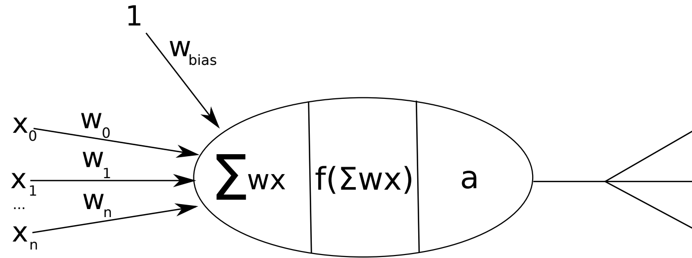

Figure 2.11: Graphical representation of a perceptron. The input to the model is represented by values $\langle x_0, x_n \rangle$ on the left, while its output value is on the right.

2.3.2 Feed-forward neural network

A neural network defines a way how individual perceptrons are interconnected. If all the connections are in a single way, the neural network is called feed-forward[7]. In this situation, the output does not depend on the internal state of a network – there are no loops. Usually, perceptrons are arranged in layers, where the output of a layer is connected to the inputs of the next layer.

In this specific application, the input of the first layer is a rolling window over acoustic data. Therefore every perceptron in the first layer expects N inputs while N is the length of the acoustic data vector. Perceptrons multiply the inputs by their individual weights – they are used for classification of different features.

2.3.3 Feature stacking

The input to the neural network used for feature extraction is a vector of rolling windows over individual normalized Mel banks. Each rolling window shows the progress of a single Mel filter bank over time with p values before and p after the frame currently being processed. The rolling windows are linearly stacked to the vector of length $(2p + 1) * M$ where p is the radius of the rolling windows, and M is the number of Mel filter banks. Consequently, the rolling windows are updated with every frame by a new sample pushed from the right, and the rest of the samples shifted to the left by one.

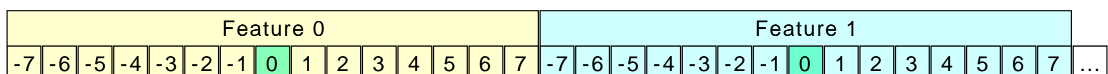

Figure 2.12: First two rolling windows of the stacked audio feature vector with $p = 7$.

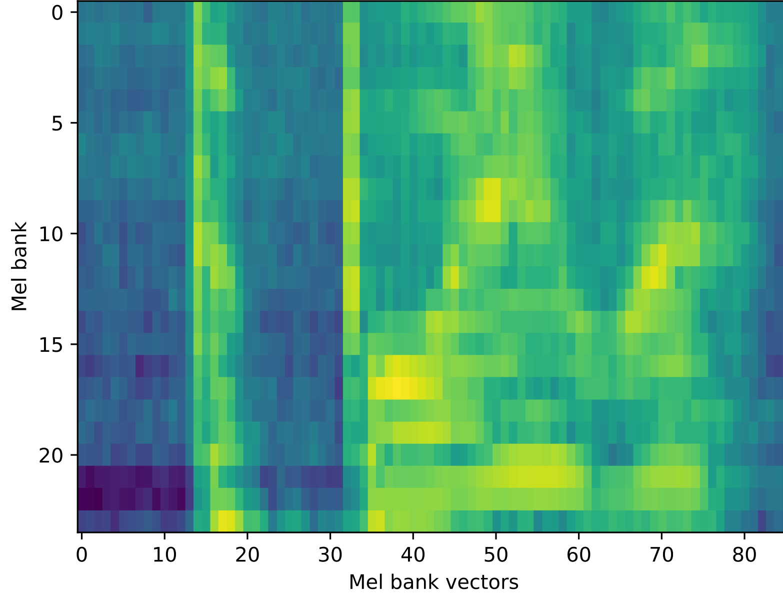

Figure 2.13: Visual representation of stacked Mel filter banks over the period of 90 samples.

2.3.4 Feature post-processing

In the end, the outputs of the neural network are represented by the series of real numbers. However, these results must be transformed into probabilities for further processing. Results are for this reason transformed into probabilities using normalized exponential function (also known as softmax function), which is used to map real values into values in range $(0, 1)$. The function is defined by Formula 2.12.

$$\sigma_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \tag{2.12}$$

For $j \in N, j \in [1, K]$ where:

K is the width of output layer

z_j is the j-th value

Prior probabilities

The features on the output of the neural network represent posterior probabilities that the input frame contains specific phoneme states. To enhance the estimation of particular phonemes in a speech, we applied the prior probabilities of the phonemes. These prior probabilities are computed from the utterance of individual phonemes in a test set. Usually, the decoder works with the logarithm of phoneme state probabilities. Given the prior probabilities, the phoneme state probabilities can be expressed as:

$$P(s, p_j) = P(s|p_j)P(p_j) \tag{2.13}$$

Here, p_j is a phoneme state, s is a sound, $P(s|p_j)$ is the posterior phoneme state probability given by the network and $P(p_j)$ is the prior probability of the particular phoneme state. Since the posterior probability is computed using softmax function 2.12, the problem might be expressed in the following way:

$$P(s, p_j) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} P(p_j) \tag{2.14}$$

Where z_j is the output of the network mapped to phoneme state p_j. Then, using logarithmic transformation, the problem can be simplified as follows:

$$\ln P(s, p_j) = \ln P(p_j) + \ln \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} = \ln P(p_j) + z_j - \ln \sum_{k=1}^{K} e^{z_k} \tag{2.15}$$

In summary, the phoneme state probabilities used as the feature for decoding are computed from the logarithm of prior probability $P(p_j)$ summed with the output from the neural network z_j and subtracted by the logarithm of a sum over exponential of all the neural network outputs.

2.4 Decoder

Generally, the decoding is a process of finding the most probable word representation given the feature sequence. In the following chapter, we will review the fundamental techniques used in phoneme posterior based decoding and outline necessary components used to build a simple speech decoder. Later, we will apply the recommendations for dynamic decoding proposed in [9].

Decoding using static recognition provides several advantages including vast language modeling possibilities and various model optimizations, which lead to better results and faster recognition process. The major problem of static decoders is the size of recognition network, usually in units of gigabytes. We tried to overcome this issue by using a dynamic decoder based on small uni-gram recognition network. N-gram probabilities are applied at run-time by performing a lookup in an appropriately designed data structure. This modification makes memory requirements significantly smaller, at an expense of making the recognition procedure more demanding on computing power. However, considering the performance of modern smartphones, this approach might be used even with quite large vocabularies and n-gram models.

2.4.1 The task of decoding

The task of decoding might be considered a problem of finding the most probable word sequence explanation, given a sequence of sounds – observations [7]. In this case, the observations are phoneme posterior probability vectors extracted from the audio signal. The most probable sequence W can be computed using Bayes' rule as shown in Formula 2.16.

$$W = \arg\max_{word_{1:t}} P(word_{1:t}|sound_{1:t}) = \arg\max_{word_{1:t}} \frac{P(sound_{1:t}|word_{1:t})P(word_{1:t})}{P(sound_{1:t})} \tag{2.16}$$

Here, $P(sound_{1:t}|word_{1:t})$ is called the acoustic model. It describes the decomposition of words to phonemes. Then, $P(word_{1:t})$ is called the language model, which describes the

prior probabilities of word utterances. Since $P(sound_{1:t})$ is not a function of $word_{1:t}$, it can be ignored in the process of maximization. Later, we will expand the language model with n-gram probabilities, which describe prior probabilities of word sequences.

2.4.2 Acoustic model

The acoustic model maps sequences of acoustic features to speech phonemes. More precisely, it defines phoneme states and probabilities of transitions between them. For this purpose, acoustic features are considered to be probabilities of individual states being the particular frames. Usually, the transitions between states are modeled using left-to-right hidden Markov models (HMM).

Typically, a phoneme is modeled by several HMM transducers and accepting a wide range of acoustic features. However, in this work, we used a simplified form of the acoustic model. We modeled the phoneme as a single HMM transducer that accepts a single feature. Each transducer defines the accepted phoneme and probabilities of self-loop or moving to the next state.

Thus, our acoustic model will be represented by represented by a lexicon, which defines the set of words known to the decoder and their phoneme decompositions. Additionally, we need a feature map, which performs a mapping of the features to phoneme posteriors, and provides information about probabilities of leaving and staying in a transducer.

This modification simplifies the process of building the uni-gram network since we can directly use phoneme decompositions of words defined by the lexicon.

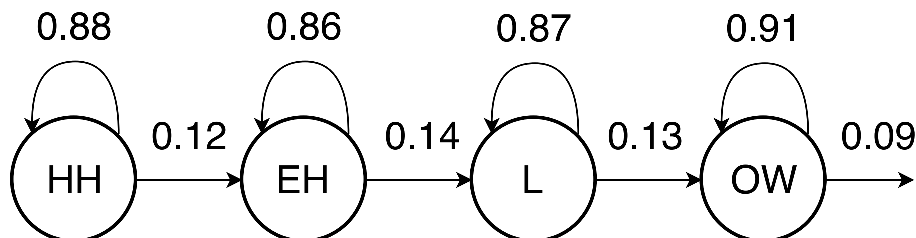

Figure 2.14: The acoustic representation of word „Hello“. The phoneme tag in a transducer determines the phoneme, which posterior probability is accepted. The value above a transition arrow defines the probability of leaving the transducer.

2.4.3 Language model

Additionally, we use a language model, which supplies probabilities of a word, and word sequence utterance in a speech. Particularly, records about these probabilities are called n-grams, while n is the length of the sequence. More precisely, the model of single words and their general probabilities of appearing in speech is called uni-gram model. Conventional speech recognizers usually use bi-gram or tri-gram models, because larger n-grams significantly increase recognition network size. In contradiction, the dynamic decoder is based on a simple uni-gram network, in which n-grams are dynamically retrieved from an n-gram storage during recognition.

The presented technique is called statistical approach to the decoding process. Although it is possible to build a speech recognizer on top of grammars, the statistical approach is much more flexible and suitable for free-form input.

N-gram probabilities

While uni-grams define the prior utterance probabilities of words in a speech given the language model, longer n-grams define whole word sequences. If a specific n-gram model is used, it means that all the $(n-k)$-grams, for $k \in [0, n-1]$ are involved. In this work, we used tri-gram models. This length should be sufficient for getting reasonable results on free-form speech while keeping the memory requirements low.

Back-off probabilities

However, defining all the possible combinations of n-grams in the language model would result in an exponential growth of the language model size. Therefore, the uni-gram model is the only fully defined one. Other layers of the model are defined only partly. Still, missing of sequences might be computed by using the back-off probabilities [4] of lower layers.

2.4.4 Token passing

All the possible phoneme sequences and words on top of them are forming a graph often called the recognition network. The problem of decoding is to find the most probable path in this graph. It is not known which features will come as next and how the situation will develop. Therefore, it is necessary to consider multiple paths which may become the best solution at the end of the recognition. Namely, one of the algorithms used for this task is called a token passing [10].

A token can be represented by an object, which remembers its path through the graph and its price. In the beginning, there is only one token placed in the start node. Afterward, in every next step, all the tokens are expanded – copied into interconnected nodes. Accordingly, the scores of tokens (logarithms of probability) are recomputed using the probabilities of transitions between nodes, which are usually representing phoneme states. Each node defines possible transitions to the next nodes (including staying in the same one) with respective probabilities.

Viterbi criterion

Token replication in every step admittedly leads to exponential growth of generated tokens count. Viterbi criterion specifies, that if multiple tokens meet in the same node, keeping only the one with the lowest price will not change [9]. Thus, the maximal number of tokens is then limited to the size of the recognition network. Yet, even with this optimization, the token count may still be too high for real-time processing. For this reason, two further optimizations are introduced. Namely, beam pruning and best N live states pruning. Firstly, we define a band between the best token and the last score to keep. Tokens which scores are outside of the given bands are dropped. This optimization technique is called the beam pruning. Similarly, the second optimization restrains the number of live tokens. Here, the defined count of best tokens is preserved, while the rest is dropped.

Modified Viterbi criterion

Basic Viterbi's criterion might not apply to the dynamic decoding working with n-grams. The problem is that a token in a certain state might be superseded by a token whose score would be better after the application of n-gram probabilities [9]. For this reason,

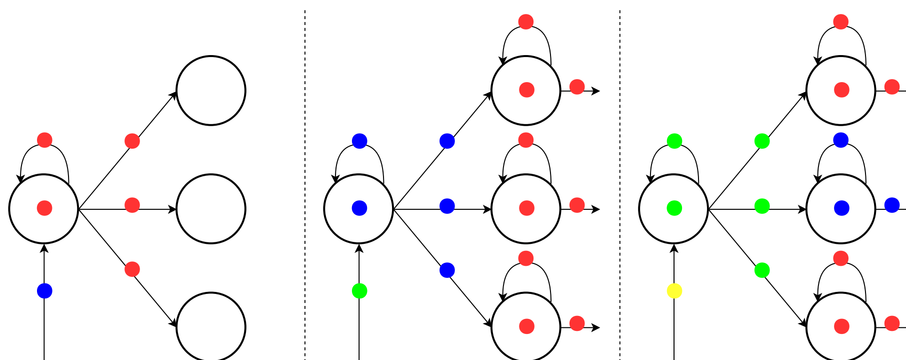

Figure 2.15: The token passing algorithm. In each step, tokens are expanded to all succeeding nodes. If two tokens meet in a single node, then the one with the better score is kept.

the criterion is modified in a way that only tokens with the same history might be considered redundant. This modification requires support for having multiple tokens in a single transducer.

2.4.5 Decoding process

In the beginning, an empty token is inserted into the initial state labeled as `<s>` (the beginning of a sentence), that accepts the posterior probability of silence. Consequently, the token is expanded to the uni-gram network and makes its path through the words. When the token leaves a word, its score is recomputed using n-gram probabilities. Then, the token is expanded to the network again, including the terminal state marked as `</s>` (end of the sentence). The sentence is complete when the decoder receives a signal from the voice activity detection system. Next, the token from the terminal state is withdrawn, and its history is used to create the final transcription of the sentence. New sentence starts on a voice signal from the voice activity detection system. Single sentences are appended to the overall transcription of the recording. To make the recognition system more responsive to the user, it is possible to display the current hypothesis of the decoder. That might be performed by printing the history (or most recent words) of the token in the terminal state in each step.

Conditional silence after a word

For the most part, a word might be followed either directly by a next word or by a short break – denoted by a comma in written language. The common approach to the issue is to have two variants for each word, with or without the silence at the end. In static networks, it might be used to form different word sequences. In contrast, having a simple uni-gram network, this would mean to use twice as many words and far more active tokens. To overcome this issue, we used a conditional silent state after each word. Each time a token is making a cross-word transition, it is both emitted from the word and copied to the silent state at the end. In the silent state, the token is self-looped and emitted from a word in each decoder step. These states are used to represent the silence between words in a sentence. However, the silence is not considered in n-gram structures.

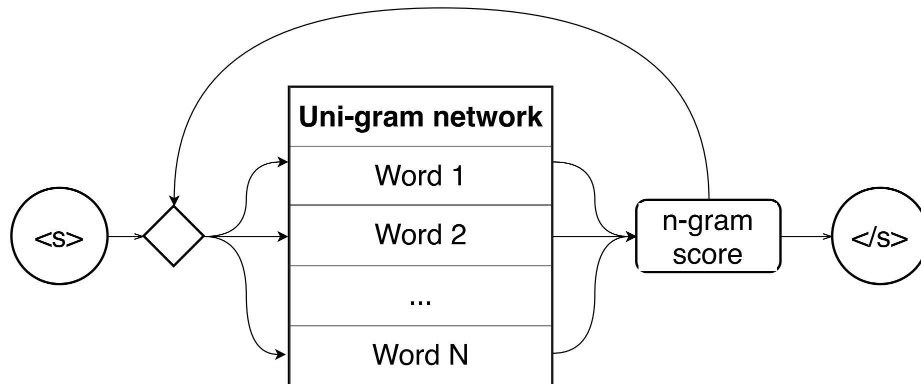

Figure 2.16: The sample representation of a dynamic decoder with a uni-gram network and dynamic n-gram probability application.

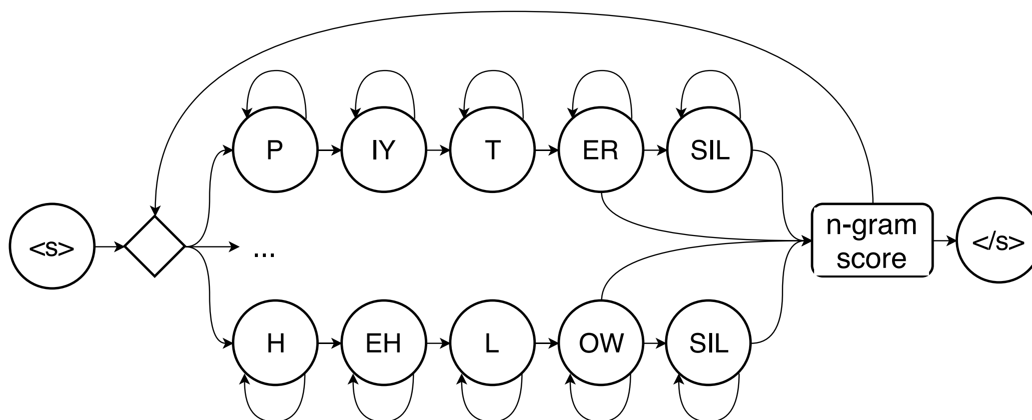

Figure 2.17: Uni-gram recognition network with dynamic n-gram probabilities including conditional silence states. Sentence „Hello Peter!“ would probably omit the silent state between the words. However, the sentence „Peter, hello!“ would presumably contain some silence between the words that would be modeled by the silent state behind word „Peter“.

2.5 Models

The statistical speech recognition implemented within this work is based on three models. Firstly, the neural network used for phoneme posteriors extraction, then the acoustic model combined with uni-gram probabilities, and n-gram probability model, which might be optional. In this chapter, we will describe the formats of these models and the way, in which they were extracted from common ASR formats.

2.5.1 Neural network for phoneme posterior probabilities

We use a five-layer ANN, including one bottleneck layer, trained on TED-LIUM corpus [6], with single state phoneme posterior probabilities on the output. For the most part, we worked with neural networks with 500 to 1500 perceptrons in the hidden layer and 80 perceptrons in the bottleneck layer. The neural network has been supplied by the project supervisor. Consequently, it was transformed into a format suitable for loading and computing using the network. A single neural network layer is represented by a matrix that

contains input weights, and bias vector. After all of the layers except the bottleneck and the last one, a logistic sigmoid is applied.

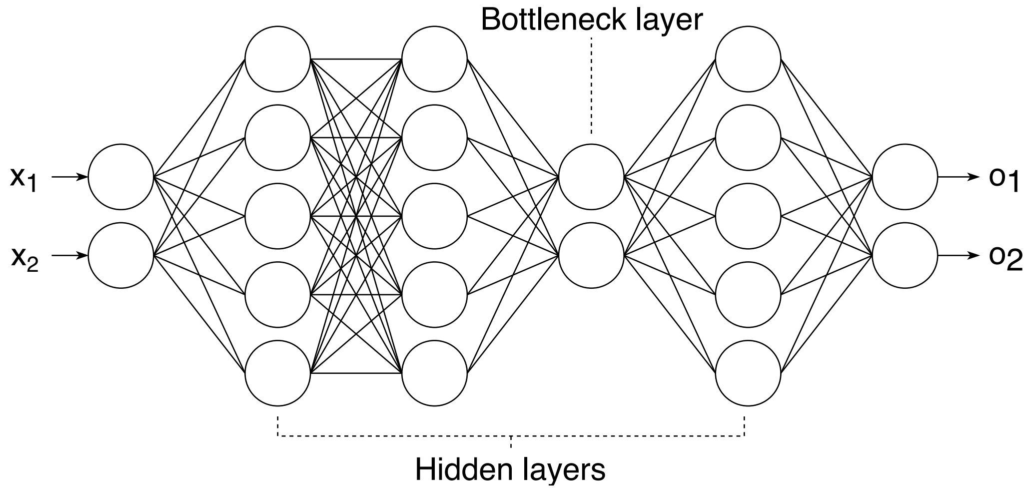

Figure 2.18: The topology of five-layer ANN with bottleneck inside, having 5 perceptrons in the hidden layers and 2 perceptrons in the bottleneck layer.

In addition to pure acoustic data, we also apply prior probabilities of individual phonemes. These prior probabilities have been extracted from the training data. Every time segment of the training set contains data about involved phoneme. The prior probability of a particular phoneme is estimated by computing its total time in the training set and dividing it by the length of the training set.

Phoneme map

Next, it is necessary to define the mapping of neural network outputs to the phoneme posteriors used by the acoustic model. The mapping is defined using a phoneme tag vector. Since the phonemes are modeled by single states, in addition to the phoneme tags, we define the probabilities of transitions between the states. These probabilities represent average phoneme lengths. Again, the average phoneme durations are computed by summing total involvement time of an individual phoneme in train set and dividing it by the total number of occurrences of that phoneme. Then the average duration is recomputed to the length of a single speech frame. If there are f frames per second of a speech and an average duration of phoneme p is d_p milliseconds, then the phoneme will on average take $a_p = \frac{d_p f}{1000}$ frames. Therefore, the probability of leaving the state is $P_{leave}(p) = \frac{1}{a_p}$

2.5.2 Acoustic model and uni-gram network

Usually, ASR systems use language models in `ARPA`[3] format, supplemented by a lexicon defining the phoneme decomposition. However, if the goal is to create an uni-gram network based decoder, it is more convenient to separate the uni-grams from n-grams and merge the lexicon with the uni-gram model. Then, the uni-gram model with integrated lexicon might be used out of the box to build the uni-gram network. This separation also empowers us to create a word index on top of the uni-gram model and build n-gram model only

[3] http://www1.icsi.berkeley.edu/Speech/docs/HTKBook3.2/node213_mn.html

using these indexes. If a single word has multiple pronunciations, then each pronunciation is stored separately – having the same transcription and scores, but different phoneme decomposition. The uni-gram model is stored in the following format:

```
WORD UNIGRAM BACKOFF NUM_OF_PHNS PHN_1 PHN_2 ... PHN_N
```

Where:

- `WORD` is the text transcription of the word
- `UNIGRAM` is the logarithmic prior probability of the word
- `BACKOFF` is the logarithmic back-off probability of the word speech
- `NUM_OF_PHNS` is the number following phonemes
- `PHN_1..PHN_N` is the phoneme decomposition of the word

For example:

```
about -2.446069 -0.5135196 4 AH B AW T
above -4.460726 -0.4070985 4 AH B AH V
```

2.5.3 N-gram model representation

Since the n-gram model is built for a specific uni-gram model joint with a lexicon, it is possible to identify the words by using only their index in the uni-gram model. This approach is faster and more memory efficient in comparison with a model that is holding full-text representations of the words. However, since a single word might have multiple pronunciations, it is possible, that it will have multiple indexes in the uni-gram model. Therefore, we will always refer to the first occurrence of the word with the same transcription in the uni-gram model. This must be taken into consideration when loading the uni-gram model. It also requires another attribute to have in the word data structure – the index of the first occurrence. N-gram models are stored in the following format:

```
NUM_OF_BIGRAMS
SCORE WORD_ID WORD_ID BACKOFF
SCORE WORD_ID WORD_ID BACKOFF
NUM_OF_TRIGRAMS
SCORE WORD_ID WORD_ID WORD_ID BACKOFF
SCORE WORD_ID WORD_ID WORD_ID BACKOFF
...
```

Where:

- `NUM_OF_[N]GRAMS` is number of entries of length N
- `SCORE` is the logarithmic probability of the n-gram stored in 32-bit float
- `WORD_ID` is an index to the uni-gram model represented by 16-bit unsigned integer
- `BACKOFF` is the logarithmic back-off probability of the n-gram

Chapter 3

Android specific implementation

While most of the Android apps are running on Java virtual machine (JVM), performance-critical real-time applications may find this high-level environment restraining. Especially, when it comes to the high load of matrix operations. For this reason, Android ships with Native Development Kit (NDK) bundle, that provides a way, how to effectively use Android resources from native low-level compiled code. The code is cross-compiled for a target platform, for example, ARMv8, using all the available platform-specific optimizations (e.g. single instruction multiple data instruction-set extensions NEON). Native code is running directly on CPU. However, NDK does not ship tools to create user interface and application on its own. Therefore, the application itself runs inside JVM and calls native code using a foreign function using Java Native Interface (JNI). This chapter introduces the library created as a part of this thesis, provides a deep insight into its implementation and evaluates different methods of approaching the problems.

3.1 Implementation goals

In the first place, the library should provide both C/C++ and Kotlin/Java interface, to allow integration of speech recognition into both native and SDK projects. Also, Android-specific parts should be separated from the speech recognition logic to empower users to use the library on other platforms as well. Consequently, modules providing the access to the platform-specific components as audio recording, or asset management, should be injected as a dependency. That will make them easily replaceable by custom implementations designed for the target platform. The library is targeted to devices using Android 4.4 (API 19) and higher. At the time of writing this work, it covers 93.5%[1] devices and it is the last version with market share among Android devices over 3%. Although Android SDK part of the application is very thin – primary for demonstration purposes, we chose Kotlin as an implementation language (officially introduced with Android Studio 3.0, compatible with Java).

3.2 Library architecture

The library is split into several modules based on their purpose. Each module is handled by a separate thread.

[1] April 27, 2018. https://developer.android.com/about/dashboards/

- Controller (Android specific) – handles basic application logic. Imports the rest of the modules and injects platform-specific dependencies.
- Recorder – handles communication between recorder, preprocessing, spectral analysis and Mel filters. Normalized Mel bank features are pushed into a queue shared with a phoneme posteriors estimation module.
- Acoustic feature extraction – estimates phoneme posterior probabilities given the Mel bank features and pushes the results into queue shared with a decoder.
- Decoder – processes the acoustic features to words, performs partial results callbacks to the user-defined functions.

The relationships between modules and objects are described in Figure 3.1.

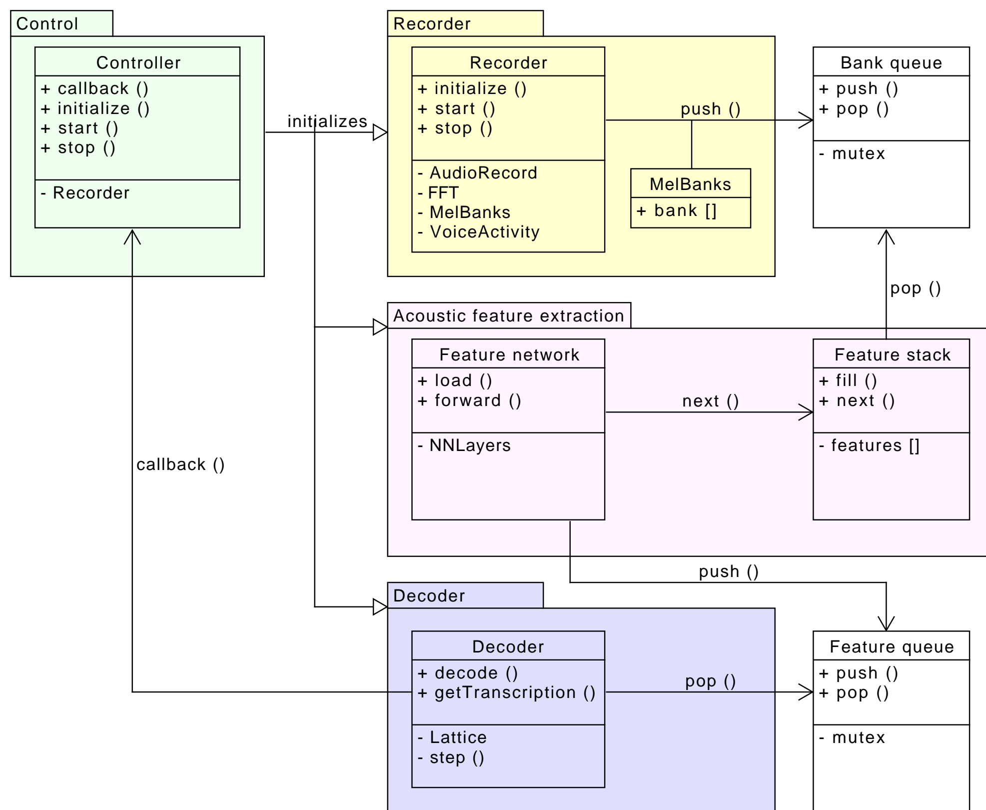

Figure 3.1: The architecture of the library, with color distinguished elementary modules, their properties, and interactions with other modules. Control module initializes Recorder, Acoustic feature extraction and Decoder modules. Module threads run as daemons and communicate using shared queues. The results are returned using controller callback.

Control (Base) module can be replaced by a custom implementation. The Base class is injected to other modules using template-based dependency injection technique. The class has to implement several methods used for callbacks and logging on various levels. That empowers a user to use his own debugging channels and error handling mechanisms.

3.3 Recorder module

Recorder module is the entry point of the speech recognition. It handles communication with system audio recorder, retrieves audio data and prepares them for feature extraction. System audio recorder is wrapped in an abstract class serving as a compatibility layer between the library and a system.

3.3.1 Audio recording

Audio recording in the native Android environment is done using Open Sound Library for Embedded Systems (OpenSL ES). OpenSL ES is hardware accelerated and provides implicit support of down-sampling to desired sample rate using Android internal resampler. Therefore performance and output quality is device dependent. Recording is stored in 16-bit PCM format – represented as 16-bit integer. The recording itself runs on a separate thread and data are accessed using a buffer queue. When a buffer is ready, library executes callback in the compatibility layer, which notifies the Recorder module. As the Recorder module has to wait until the buffer is ready to be processed, it can perform preprocessing tasks for feature extraction. When a filled buffer is received, then sub-sampling is performed if required and data are passed to the further processing. The empty buffer is then enqueued again.

3.3.2 Audio feature extraction

The audio track is processed using a rolling window with defined step size. Since samples are linearly stored in an array, desired behavior can be effectively achieved by using a block copy. The samples from the previous frame are shifted to the left, and new samples are copied in. Then Hamming window function is applied to the samples, by manually multiplying every sample with Hamming window coefficient given by index in the frame.

Fast Fourier Transform is handled by library kissFFT[2]. Audio samples are on a single – real axis what allows to use optimized variants for real input. $N/2-1$ complex samples on the output is converted to power spectrum and used as an input for Mel filter banks. Mel bank features are computed as column wise dot-product of the input vector and a Mel bank matrix. Then the results are logarithmically transformed and dynamically normalized. Mel bank computation is powered by Eigen library, which is providing acceleration using NEON instruction set if available. For normalization, we implemented the techniques proposed in Section 2.2.5. With both of these, it is easily possible to set the initial values for individual banks, what allows to kick-start the recognition process without waiting for the normalizer to get stable.

Normalization

Each normalization technique may have its benefits based on the properties of the target area. Due to its characteristics, the exponential normalization should be a better fit in areas with volatile background noise level. On the other hand, uniform last-N normalization should be performing better in areas with constant background noise as an office or a call-center. On a test set containing recordings from conference talks (close microphone, quite constant noise) did the uniform last-N normalizer outperformed the exponential normalizer by 4% of the word error rate (see Section 4.1).

[2]BSD licensed. http://kissfft.sourceforge.net/

Implementation of the uniform last-N normalization

During the work, we designed and implemented an optimized variant of uniform last-N normalization using weighted averages. Instead of storing all the samples recorded during a specified period and recomputing the mean in each step, it is sufficient to use two vectors and one counter variable.

```
struct UniformNormalizer {
    void normalize (vector &banks);
    vector current;
    vector previous;
    uint32 position;
};
```

The normalizer works in two phases using the counter variable `position` as a weight for computing the weighted average of the vectors. Using the counter variable provides a continuous transition between the phases. At first, the value of `counter` is zero, that means that there are no frames in the current phase. Therefore, the frames in this phase should be included with weight 0 and frames from the previous phase with weight $N - 0$. N is the number of frames considered for normalization (e.g. 500 for 5 seconds of speech). Next, with every frame increment the position by one, add new Mel filter bank values to `current` vector and compute the weighted average. When the counter reaches N, `previous` is replaced by `current` and `current` and `position` are zeroed.

Algorithm 1 The algorithm used in uniform last-N normalization.

1: **procedure** NORMALIZE($banks$) ▷ banks = Mel filter banks vector
2: position $\leftarrow position + 1$
3: **for** i from 0 to length(banks) **do**
4: current[i] $\leftarrow current[i] + banks[i]$
5: banks[i] $\leftarrow banks[i] - (current[i] + (N - position) * previous[i])/N$
6: **if** $position \geq N$ **then** ▷ Next phase
7: previous $\leftarrow current/N$
8: current $\leftarrow (0, 0, ...0)$
9: position $\leftarrow 0$

Implementation of the exponential normalization

We also experimented with using an appropriate exponential function for the normalization. This is beneficial since the weight of the older samples is decreased over time. That makes the transition smoother and still, there is no need to remember individual values of the previous samples. The implementation of such normalization is straightforward: provided with suitable parameters b and $d = \int_0^\infty b^x dx$, formulas 2.9 and 2.10 are applied to every Mel filter bank feature. In this case, only a single vector `previous` is required to hold mean values of previous frames.

Algorithm 2 The algorithm for normalizing a single frame using exponential normalization.

```
1: procedure NORMALIZE(banks)                      ▷ banks = Mel filter banks vector
2:     for i from 0 to length(banks) do
3:         previous[i] ← previous[i] * b + banks[i]/d
4:         banks[i] ← banks[i] − previous[i]
```

3.4 Acoustic feature extraction module

Normalized Mel banks are pushed into a queue shared with feature extraction module. The extraction thread is running as daemon waiting for features to come. Firstly, withdrawn features are stacked into a vector as linearly aligned rolling windows – for each sample separately. Then the vector is forwarded through a neural network, which is represented as a linear model with the definite number of layers. The result of the forward pass is pushed into a queue shared with the decoder module as a feature vector. Vector and Matrix operations in the neural network are done using Eigen library methods, especially vector summation, matrix-vector multiplication, and coefficient-wise product.

Properties of the neural network are given by a model. The model is packaged with an application and loaded at run-time. Although the model is serialized in a simple custom binary format, Android application assets are compressed and must be accessed using specialized Android NDK methods. Each layer is serialized as a vector and weight matrix. The layers are loaded at the run-time to the neural network object that implements methods for forward-passing the input vector through the network.

3.4.1 Phoneme posteriors estimation performance

The performance of feature extraction, for the most part, depends on the size of a neural network. For testing purposes, we used a feed-forward neural network with five layers, including one bottleneck layer. Since the size of a single layer can be more than 1000 perceptrons, the task might have to be parallelized among several CPU cores. We discovered, that for the bigger networks (more than 1000 perceptrons per layer) it is necessary to use appropriate single instruction multiple data instruction set with a single-precision floating point representation (32 bit). Specifically, we used NEON[3] instruction set extension on 64-bit ARM architecture (compatible with the most of modern Android devices).

The most significant difficulties of the feature extraction process are linear algebra operations, especially vector-matrix multiplication. For this reason, it is important to choose a suitable linear algebra library with the support of vectorization on a given platform. Thus, we decided to use C++ library Eigen[4] with native NEON support. We also examined the time requirements for the feature extraction using RenderScript[5] framework. In Table 3.1 we compared the time required to perform feature extraction of 30 second long recording – using chipset Qualcomm MSM8953 Snapdragon 625 (octa-core 2.0 GHz Cortex-A53).

[3] https://developer.arm.com/technologies/neon
[4] http://eigen.tuxfamily.org
[5] https://developer.android.com/guide/topics/renderscript/compute

Table 3.1: The time required to perform a feature extraction from 30-second long recording using a 5-layer neural network. Perceptrons is the width of the hidden layer. In NEON and no vectorization variants, the extraction is done in the native C++ environment using Eigen library parallelized among 3 CPU cores. Parallelization of the RenderScript variant is handled by the operating system, utilizing all the available resources if possible. RenderScript evaluation on larger networks was not successful.

	Single precision (32b)			Double precision (64b)	
Perceptrons	NEON	no vectorization	RenderScript	NEON	no vectorization
500	2.33 s	2.51 s	7.73 s	5.53 s	5.59 s
1200	14.34 s	16.32 s	N/A	28.66 s	32.23 s

3.4.2 Phoneme posteriors estimation parallelization

Due to high requirements of the feature extraction process, it is necessary to provide suitable parallelization design. Since our goal is real-time processing, we introduced layer-wise parallelization. Particular neural network layers are processed by separate threads, that are interconnected by shared queues. Therefore, the most significant bottleneck is the single-core performance, which must be sufficient for real-time processing of a single neural network layer. The size of the network is also limited by the number of cores on which the task might be parallelized. However, small layers (less than $\approx$ 600 perceptrons) may be processed on a single core.

In Figure 3.2 we compared the time required to process a single layer of a neural network on a single core. According to these measurements, it can be seen, that a reasonable number of perceptrons in a hidden layer of a five-layer neural network would be around 1200. This configuration would lead to utilization of approximately two CPU cores in an active state.

3.5 Decoder module

When implementing a dynamic decoder module, it is necessary to carefully design its major components. Most importantly the uni-gram network, n-gram storage, methods for execution of the token passing algorithm and relevant data structures. Additionally, it is essential to efficiently implement the pruning techniques used to restrain the number of tokens in the network and develop an eligible way how to store the history of tokens.

3.5.1 Uni-gram network representation

The most performance critical part of a dynamic decoder is a token passing algorithm, which is principally affected by representation of the uni-gram network. Probably the most straightforward representation of such structure would be a matrix of tokens, with word identifiers on one axis and position in the word on the other one. However, this representation meets with two significant disadvantages. At first, words have different lengths thus a substantial part of the matrix would be empty. Secondly, we need to restrain the number of active tokens in the matrix. That would make the matrix sparse and hard to efficiently iterate over.

During the development process, we experimented with several implementations of the uni-gram network, including red-black trees, hash-maps, and custom implementation inspired in a hash-map – vector of reversed singly linked lists indexed by word identifiers.

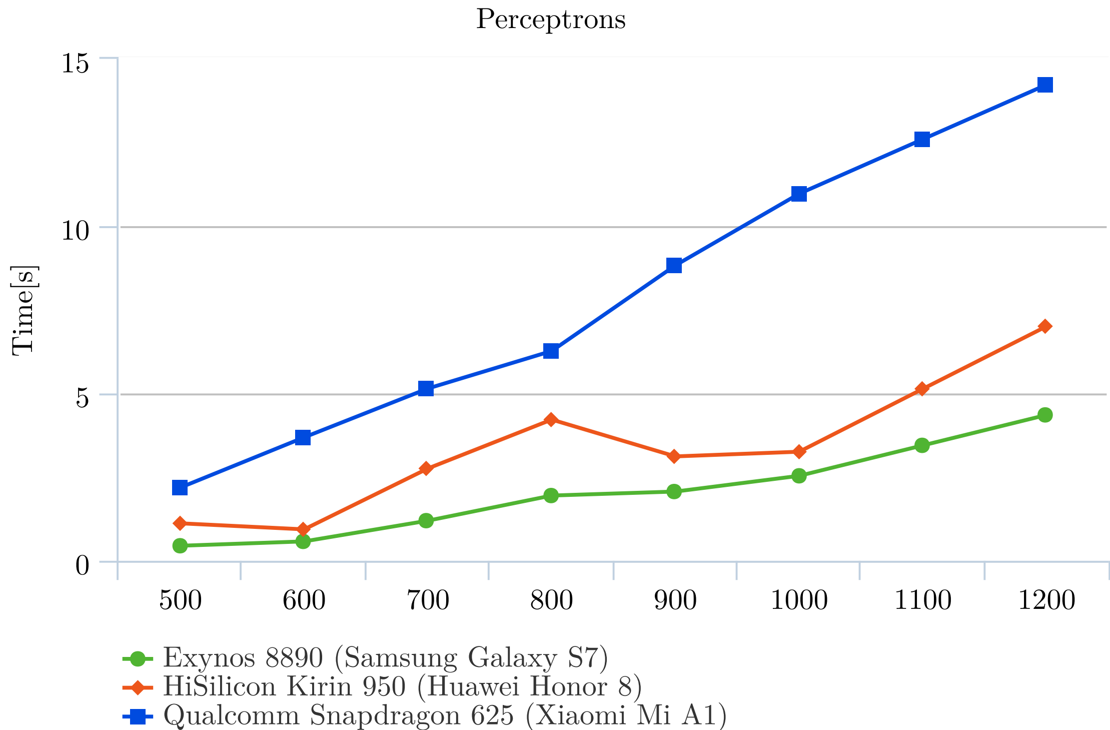

Figure 3.2: Comparison of the time required to extract features from 30 seconds of recording using various widths of the hidden layer in the neural network on recent Android devices. The number of perceptrons in a layer is shown on the bottom axis.

A single item of the reversed singly linked list might be represented by the following data structure:

```
struct Bucket {
    Token token;
    uint16 position;
    Bucket *previous;
};
```

Then, a reference to the last item in a word is stored for each word in the lexicon. This can by done using C++ STL container `std::vector`. If there is no item in a particular word, a zero reference (`nullptr`) is stored at the corresponding position. Thus, the network might be represented by the following structure:

```
struct Network {
    std::vector<Bucket *> rows;
};
```

The custom implementation makes it possible to update tokens in-place, avoiding unnecessary copies, and also providing support for row-wise parallelization, unlike the other dynamic structures. Table 3.2 shows the performance and memory comparison of the particular data structures.

Table 3.2: The time and memory required to decode features extracted from 75-second long recording on a single core using the uni-gram network only, a dictionary of 12000 words, 2000 live tokens and a pruning beam set to 100.

Structure	Time[s]	Memory[MB]
Custom	31.78	1.12
std::unordered_map	95.23	1.43
std::map	180.52	1.94

Using the dynamic decoder might require additional changes to the token passing algorithm. Since n-gram probabilities are applied after the word is emitted, it is possible, that a token with slightly better acoustic likelihood will kill a token, that would be better after the application of n-gram probabilities. That might compromise the results of the recognition process. For this reason, as proposed in work [9], we introduced modified uni-gram network representation, that permits having multiple tokens with different paths in a single HMM transducer.

Again, this behavior might be achieved by using an appropriate dynamic container (like C++ standard templates `multimap` or `unordered_multimap`). In order to make it more efficient, we modified our custom implementation by adjusting the behavior of the reversed singly linked list. More precisely, tokens in the same position will not be replaced unless their history is the same. The performance of these data structures is reviewed in Table 3.3.

Table 3.3: The time required to decode features extracted from 75-second long recording on a single core using uni-gram network, a vocabulary of 12000 words, 2000 live tokens and a 100 pruning beam having multiple tokens (with different word path) per transducer.

Structure	Time[s]	Memory[MB]
Custom	32.62	1.32
std::unordered_multimap	109.04	1.89

3.5.2 Token structure

The effective implementation of a token is crucial since thousands of tokens are both constructed and destructed in every step. Therefore, the construction and destruction operations have to be well optimized. Additionally, the structure should be as small as possible, to allow efficiently creating copies of the tokens. Each token has to store its score and path through the network. Whereas the score can be easily represented by 32-bit floating point value, the path through the network is a bit more complicated.

3.5.3 Word link record

Since the final transcription is chosen at the end of the decoding, it is necessary for a token to remember its history – words emitted on its path through a network. Whilst it could be possible to store word identifiers in a dynamic array, this approach would cause significant performance problems with copying and destructing tokens, especially during later phases. This problem is usually solved by creating a linked list of tokens. Each token only stores a pointer to the last word in a list. The token structure might be represented as follows:

```
struct Token {
    float32 score;
    void *path;
};
```

When copying such token, one can only copy a reference to the last word. Later, when forked tokens will have their own history, different from the original token, they will create tree-like structures (depicted in Figure 3.3).

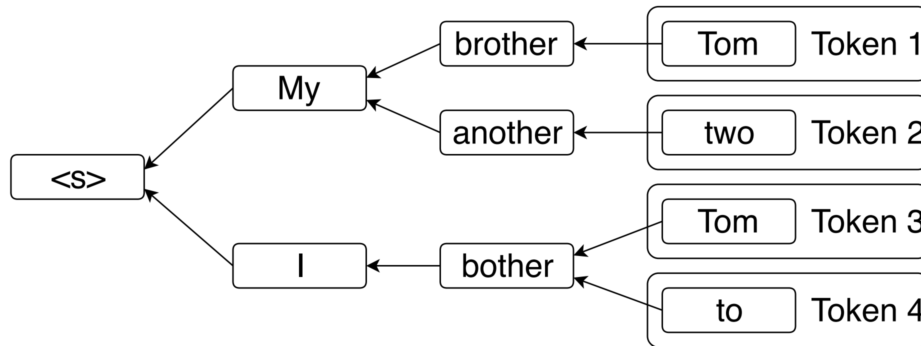

Figure 3.3: Word link record tree structure example.

This approach is memory efficient and fast for creating copies. However, when it comes to memory deallocation caused by a token removal, it is not clear whether the list might be released, since other, forked tokens might still hold a reference to certain elements in the list. Hence, it is necessary to introduce a reference counting mechanism. Each time a token is copied, the reference count of the last element in the list must be incremented. Also, it is important to implement copy constructors for list elements, to catch every copy of the instance. Accordingly, when a token is being released, the reference count of the last element in the list is decreased. If the reference count of a list element drops to zero, then it is released and the reference count of its predecessor is decremented as well (and recursively). Moreover, if we want to parallelize the process of decoding, the reference counting must be thread-safe. That requires either introduction of mutual exclusion locks or preferably, using atomic types (introduced in `C++11`).

Having these requirements, it is felicitous to use `std::shared_ptr`[6] – a smart pointer that retains ownership of an object through a pointer while using atomic functions for reference counting, copy constructors, and copy assignments. We will define the list item as:

```
struct WordLink {
    const uint16 wordID;
    std::shared_ptr<WordLink> previous;
};
```

3.5.4 Token passing

In each step, each token in the recognition network needs to be updated. Therefore it is necessary to efficiently iterate over the active tokens. By using the uni-gram network

[6] http://en.cppreference.com/w/cpp/memory/shared_ptr

representation we proposed, it is possible to perform row-wise iteration over the network. Updating the tokens in a single row has no impact on the rest of the network. This feature is obligatory for implementation of parallelized decoding.

For each row, it is iterated over the singly liked list, while each token is processed in the following way. Firstly, its score is compared with the chosen threshold. This threshold is determined by pruning techniques explained later. If the score is worse than a chosen threshold, the token is dismissed and removed from a list. The deletion operation is performance critical since the number of live tokens is at least doubled in each step of a decoder. Here, the removal can be done by setting the pointer to the next element of the previous element to the successor of the element being removed. Subsequently, the original element may be released. Find the deletion operation is depicted in Figure 3.4.

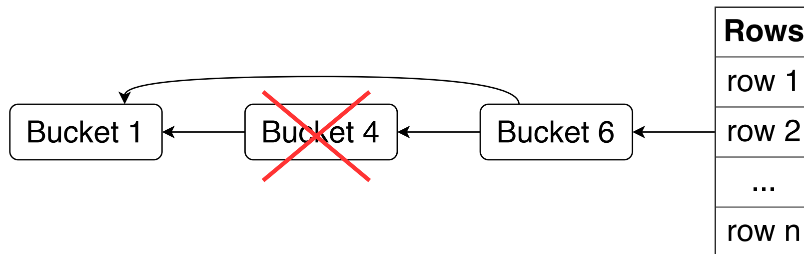

Figure 3.4: The deletion of a single element (bucket – see Section 3.5.1) from the reversed singly linked list. Structure `Rows` holds pointers to the last bucket in a word, or a zero reference if there is no bucket.

Otherwise, if the token's score is satisfactory, it will be updated. By the update, we understand duplicating the token, moving its copy to the succeeding transducer and correspondingly recalculating the scores. If there is a succeeding transducer, the score of the copy is incremented by the logarithmic probability of a particular feature in a speech, summed with the logarithmic probability of leaving the state and multiplied by acoustic scale. On the contrary, if there is no succeeding transducer, then the token is marked to be emitted from a word. The original token is updated respectively, but using a logarithmic probability of staying in a state.

Algorithm 3 The token passing algorithm using the custom data structure based on the array of reversed singly linked lists.

```
 1: procedure TOKEN PASSING
 2:     for i from 0 to size(rows) do
 3:         token = rows[i]
 4:         previous = &rows[i]
 5:         while token ≠ nil do
 6:             if token→score<threshold then
 7:                 *previous = token→previous
 8:                 delete(token)
 9:                 token = *previous
10:                 continue
11:             else
12:                 update_token(token)
13:             previous = &node→previous
14:             node = node→previous
```

Viterbi criterion

While iterating the network and emitting tokens to a next state, is possible to apply Viterbi criterion in real-time. If there already is a token in the next state (in the bucket with the position of the next state) its score is compared with the score of the emitted token. In case that the emitted token's score is better, the original token in the following bucket is replaced by the new one. Otherwise, the emitted token is dismissed.

Modified Viterbi criterion

To introduce the support for dynamic decoding, the modified Viterbi criterion has to be implemented. This modification also requires the support for having multiple tokens in a single state. However, having tokens in a word represented as a linked list, it is not a problem. With this in mind, it is only necessary to compare the history of words emitted by a token and use this information when comparing tokens in a word against each other. Though, comparing histories (to the depth of the longest n-gram) might be performance demanding for longer n-grams. Therefore, we interposed another member to the token structure, carrying a hash of the tokens history.

```
struct TokenWithHash : public Token {
    uint32 pathID;
};
```

As a hashing function we chose `sdbm` algorithm[7]. which provides a simple and fast approach to hashing integers with a good distribution. Yet, the hashing algorithm is only used when a new word is pushed to the token's history. The hash of the history is updated with every new word as follows:

```
pathID += wordID + (pathID << 6) + (pathID << 16) - pathID
```

Using this technique, paths of tokens might be compared just by comparing the hashes, rather than traversing their word link records.

[7] http://www.cse.yorku.ca/~oz/hash.html

Best N live states

In spite of the growing number tokens in each step, it is necessary to effectively reduce them, securing consistent system load that can be handled by available resources. The most constraining factor is the maximal amount of live tokens in the system. Therefore, we must implement methods to assure that the upper bound will not be exceeded. At first, we proposed a method for limiting the number of tokens in real time using a red-black tree holding N best values. However, frequent insertions and removals of the element with the worst score led to frequent rebalances of the tree and required to backwardly remove a token with the worst score from the network. In fact, it is not so important how many tokens are emitted in a single step, but how many tokens are processed. Thus, it is sufficient to reduce the tokens after the whole network is updated, or before a step is executed. In the end, we found the following method to be most efficient. Before executing the decoder steps, scores of tokens are sorted and the N-th element is selected. Consequently, while iterating over the network and updating tokens only ones with a score better than a picked one are updated, others are released.

As a sorting method, we compared standard `C++` library template functions `std::sort`, `std::nth_element` and `std::make_heap` combined with `std::pop_heap`. While the first method sorts the whole container, the second one only sorts N best elements, what is sufficient for this application. In contrast, the third method at first arranges the elements into a heap and then removes the best element N times. The comparison of these methods is shown in Table 3.4.

Table 3.4: Time required to obtain 4000-th best element from the vector of 15000 elements, 5000 times, using different methods

Method	Time[s]
std::nth_element	0.64
std::make_heap + std::pop_heap	2.42
std::sort	4.15

Beam pruning

Using a sorting method for best N live states also simplify the process of beam pruning. When a container with token scores is sorted, then the element with the best score is selected. Then the width of the beam is subtracted. If the threshold is more constraining than the one from best N live states, then the one from beam pruning is used.

3.5.5 N-gram storage implementation

Since decoding with large vocabulary requires a significant amount of n-grams, it is important to design the n-gram storage to be memory efficient, but also provided with a reasonable lookup speed. The simplest solution would probably be to store these n-grams in a hash-map indexed by a tuple of word indexes. However, this approach might be problematic when using n-grams of various lengths – would require implementing a custom hashing function for tuples of various sizes. We used the solution for n-gram storage design proposed in work [9]. Here, n-grams are stored in an array ordered by word identifier. Additionally, there is another array for storing ranges of n-gram indexes for particular words.

Specific n-gram is then found by a binary search within the range of a single word. Table 3.5 shows the performance and memory comparison of these n-gram storage designs.

Table 3.5: The time required to perform 25 million random lookups in n-gram storage filled with 5 million bi-grams.

Structure	Lookup[s]	Load[s]	Memory[MB]
Custom	9.78	2.43	40.40
unordered_map	14.17	6.87	140.45

3.5.6 Decoder parallelization

The most performance-demanding components of the decoding process are updating each token in the network, expanding the best token to the network and evaluating n-gram probabilities of emitted words. Fortunately, thanks to the custom data structure, each of these might be row-wise parallelized with no data races. However other parts, like computing the pruning threshold, are still done on a single thread. Moreover, considering the parallelization overhead, the performance gain might be not that significant on smaller models. As an implementation, we used standard C++ STL `std::thread` for slave thread and `std::mutex` with `std::condition_variable` for thread synchronization. In Table 3.6 we compared the time required to process a recording with and without the parallelization.

Table 3.6: The time required to decode 30 seconds of recording with and without parallelization, using various tri-gram models (4000 live tokens, 150 beam pruning).

Words	Bi-grams[M]	Tri-grams[M]	1 core[s]	2 cores[s]
15000	1.79	1.94	49.27	39.77
15000	0.98	0.78	48.75	38.68
12500	1.69	1.91	42.47	29.52
12500	0.93	0.77	41.82	27.05
10000	1.56	1.87	35.09	24.81
10000	0.86	0.76	34.34	26.13
7500	1.38	1.80	27.84	21.86
7500	0.76	0.75	26.23	19.66
3000	0.82	1.52	10.66	8.73
3000	0.50	0.68	9.00	7.27

3.6 Binary representation of the models

Due to the considerable size of models used, it is necessary to provide reasonable loading performance. Whilst the load speed of strings from a plain text does not differ much from loading it from a binary file, the load speed of numeric types is notably worse. Especially, this is the problem of neural networks and n-gram language models, which are almost entirely numeric. Storing a real number in plain text with a reasonable accuracy might

take more than 12 characters – bytes. On the other hand, storing a real number in a binary, 32-bit float representation takes only 4 bytes and provides a much better accuracy. Moreover, reading numbers from plain text and parsing them to numeric representation is much slower, than reading them directly from a binary format. Still, endianness of the target architecture must be taken into consideration, supposing that data are generated on x86_64 – little-endian system. In Table 3.7 we compare the time to load models from plain text and binary form on an Android device. Binary formats are tailored for the individual models, with respect to the dominant data types.

Table 3.7: The load time and memory required for a neural network and n-gram language model on Snapdragon 625 based Android device. The neural network has 5 layers and 500 perceptrons in the hidden layer. The n-gram language model contains 0.82M bi-grams and 1.52M tri-grams.

Model	Load time[s]		Size[MB]	
	binary	plain text	binary	plain text
Neural network	0.23	2.01	2.1	6.6
Language model	1.69	24.16	54	24

3.6.1 Binary representation of a neural network

A neural network can be serialized as a series of vectors and matrices. Therefore, the major data type in a neural network model is the floating point representation of a real number. On the first byte, the type of the serialized object is indicated by the initial character, which may be either „v“ or „m“ (vector or matrix). Then, the size of the entity is stated on 4 bytes, represented by unsigned, 32-bit integer (one dimension for vector, two dimensions for matrix). The size of followed by the declared amount of serialized 32-bit float values.

Table 3.8: The binary representation of a single neural network layer with weights in the first row and bias in the second one.

0	1	2	3	4	5	6	7	8	$9 \rightarrow n$
type (m)	matrix rows				matrix columns				matrix data
type (v)	vector length				vector data				

3.6.2 Uni-gram model binary representation

Since the uni-gram model includes the lexicon, it contains both text and numeric data types. The model is serialized in the format described in Section 2.5.2. Firstly, the word's transcription is written. In order to make deserialization more efficient, the length of a word is defined in advance. Thus the word might be read as a block. After the transcription, the uni-gram and back-off probabilities are provided in 32-bit float format. In the end, the word phoneme decomposition described. Similarly to text, we define the number of phonemes in advance and then provide indexes of involved phonemes. These could be stored on a single byte. However, in order to support larger acoustic models (e.g. tri-state phonemes), we used a 16-bit unsigned integer representation.

Table 3.9: The binary representation of a single uni-gram model entry, where L is the length of the word's transcription and P the number of phonemes/states. The first row shows byte indexes. The second one shows the data stored on the particular indexes.

0	$1 \to L$	$L+1 \to L+5$	$L+6 \to L+10$	$L+11$	$L+12 \to L+11+2P$
L	text	uni-gram score	back-off score	P	phonemes

3.6.3 N-gram model binary representation

Using the indexes to uni-gram model in n-grams instead of full transcription allows us to make the n-gram model entirely numeric. Therefore, the binary representation is more than appropriate. The fixed form of the n-gram model does not require any specific adjustments. It is sufficient to serialize the model in the format defined in Section 2.5.3 in suitable data types – a 32-bit float for probabilities and a 16-bit unsigned integer for uni-gram model word indexes.

Table 3.10: The binary representation of a single bi-gram and tri-gram entry in a tri-gram model (no back-off for tri-grams).

Byte	0	2	4	6	8	10
bi-gram data	score		word 1 ID	word 2 ID	back-off	
tri-gram data	score		word 1 ID	word 2 ID	word 3 ID	

3.7 Implementation summary

During this work, we implemented a fully functional automatic speech recognition system based on a dynamic decoder. The audio recording is done using `OpenSL ES` library available in Android Native development kit. Consequently, the recorded audio is preprocessed and the spectral analysis is performed using `kissFFT` library. Then, the custom voice activity detection is performed, Mel filter banks are computed, and the features are forward to the input of a neural network. The neural network computations are powered by `Eigen` library, using instruction set extension `NEON` and it is layer-wise parallelized with use of shared queues. Phoneme-state probabilities are processed by the dynamic decoder that is based on the custom implementation of the token passing algorithm. The token passing algorithm is performed on top of the uni-gram network that is represented by the proposed structure. N-gram probabilities are applied dynamically, while the n-grams are stored in n-gram storage based on the structures proposed in work [9]. The process of decoding is, for the most part, parallelized among two CPU cores. Both language and neural network models are stored and loaded in a custom binary format. Such configuration is able to handle a language model with a vocabulary of 12500 words in real-time on a recent mid-end Android device. The functionality of the solution is presented by a demo application, which is using the library interface from JNI environment.

Chapter 4

Evaluation and testing

This chapter reviews the techniques used to measure the correctness of the speech recognition process implementation, as well as to evaluate the actual results of the recognition.

4.1 Evaluating the results

The results of automatic speech recognition are usually evaluated using so-called *Word Error Rate* (WER) computed as:

$$WER = \frac{S + D + I}{N} \tag{4.1}$$

Where:

S is the number of substitutions

D is the number of deletions

I is the number of insertions

N is the number of words in reference

To evaluate ASR hypothesis, it is necessary to compute minimal edit distances between arbitrary sequences. For this task, we used open-source utility `asr-evaluation` available online[1].

4.1.1 Test set

As a test set, we used a subset of TED-LIUM [5] corpus. Firstly, we transformed the test recordings to an eligible format and sub-sampled them to the sample rate used by our ASR system. Then, we extracted first 90 seconds from each recording (aligned to whole sentences) and prepared plain text transcriptions of those segments. In total, the score was evaluated on 978 seconds of audio in 11 recordings, containing 2775 words.

4.1.2 Test results

We compared word error rates achieved with various sizes of neural networks and language models. Table 4.1 shows the results achieved on recordings from the test set.

[1] `https://github.com/belambert/asr-evaluation`

Table 4.1: The word error rate computed on the test set using a dynamic decoder with various language model and neural network sizes. Words, bi-grams and tri-grams define the used language model. Perceptrons is the number of perceptrons in the hidden layers of ANN used for posterior phoneme state probability estimation. Configuration: 4000 live states, 150 beam pruning.

Words	Bi-grams[M]	Tri-grams[M]	Perceptrons	Average WER[%]
3000	0.49	0.67	500	73.74
			800	70.89
			1200	68.75
3000	0.83	1.52	500	74.04
			800	70.91
			1200	69.03
5000	0.65	0.71	500	72.84
			800	69.10
			1200	66.92
5000	1.13	1.69	500	72.59
			800	68.77
			1200	67.03
7500	1.38	1.80	500	72.77
			800	70.39
			1200	67.53
10000	1.55	1.87	500	73.54
			800	70.00
			1200	68.25

4.1.3 Comparison with a static decoder

The implemented dynamic decoder performed slightly worse in comparison with a static decoder, using the same features extracted from TED-LIUM recordings – the results are shown in Table 4.2. However, the implemented dynamic decoder should be able to compete with conventional static decoders. There is still a lot of work to be done fine-tuning the phoneme posterior probabilities extraction and language models for use with mobile device's microphone in the desired domain. For testing purposes, we used a five-layer ANN with 500 perceptrons in the hidden layer, including one bottleneck layer, with single state phoneme posterior probabilities on the output. The neural network was trained on TED-LIUM corpus [6].

4.1.4 Three-state phonemes

We also tried to extend the acoustic model by adding multiple states for each phoneme. For phoneme sub-state posteriors estimation, we used an artificial neural network with 5 layers and no bottleneck. In this experiment, we modeled each phoneme by three sub-states, having 152 sub-states in total. Conditional silence at the end of the words has been modeled by a transducer, that is accepting the maximum of all the non-speech states (silence, hesitation, coughing). This approach should lead to better acoustic results, however, might

Table 4.2: The word error rate computed on the test set using a dynamic and static decoder with the same configuration (4000 live tokens and 150 beam pruning) and a vocabulary of 10k words supplemented by a 3-gram language model.

Recording	Static decoder WER[%]	Dynamic decoder WER[%]
1	86.09	83.44
2	55.85	66.66
3	85.62	87.43
4	66.92	74.41
5	65.77	67.45
6	62.63	62.63
7	61.60	76.79
8	64.07	72.59
9	86.9	85.12
10	71.32	74.71
11	56.79	63.21
Average	69.42	74.04

need more tokens for decoding, since the number of available states in the unigram model has tripled.

Table 4.3: The word error rate on a subset of TED-LIUM test set using three-state phonemes. Perceptrons is the number of perceptrons in the hidden layers of the ANN. Parameters words, bi-grams and tri-grams define the properties of used language model. Configuration: 8000 live tokens, 150 beam pruning.

Words	Bi-grams[M]	Tri-grams[M]	Perceptrons	Average WER[%]
10000	1.56	1.87	1500	59.51
			700	63.58
7500	1.38	1.80	1500	58.20
			700	62.81
5000	1.13	1.69	1500	57.94
			700	61.58

4.2 Demo application

We created a simple demo application for library demonstration and testing purposes. The application contains a single Android activity, with a button which controls the recognition process. Additionally, the application provides a simple interface for choosing the used language model, by default the language model, numbers model and pure acoustic mode. The hypothesis development of the current sentence is shown in real-time. Once the hypothesis is confirmed by a silence at the end of a sentence, then the whole sentence is added to the transcription. Additionally, ongoing voice activity is indicated by a color of the control

button – green = speech; red = silence; blue = turned off. Find the demo application described in Figure 4.1.

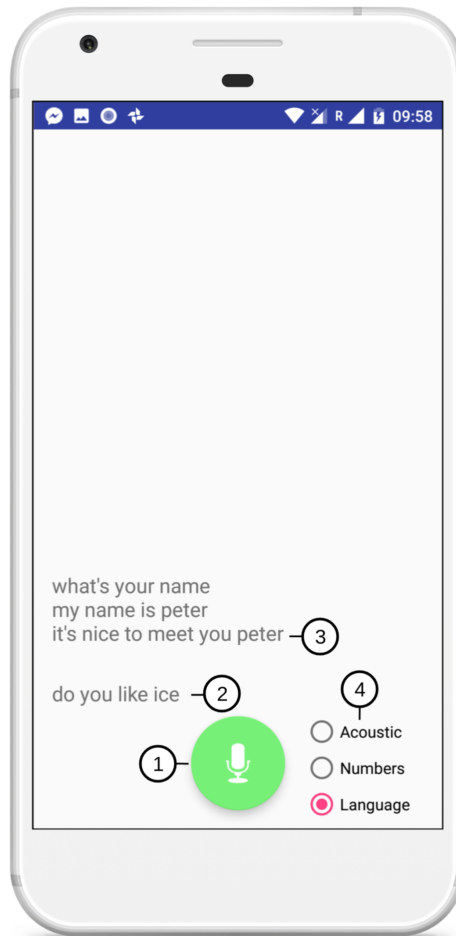

Figure 4.1: A phone mock-up with a screenshot from the demo application, where 1 is the control button, 2 is the hypothesis view, 3 is the transcription view and 4 is the model picker.

We found, that most successful configuration for online demonstration is to use a neural network with three-state phoneme posteriors on the output with 700 perceptrons in the hidden layers combined with a 7500-word dictionary. This configuration worked smoothly on a mid-end smartphone using approximately 100 megabytes of random access memory.

4.3 Unit testing

To make sure that all steps are performed correctly and will remain consistent in further development, it is necessary to introduce unit testing. For this reason, unit tests compare the results of optimized Android implementation for Android with an accurate referential implementation.

Designing library modules in a separable and testable way is an important request of the library architecture. Therefore, units have to be independent and generic enough to allow usage in a mocked environment. In software engineering, this problem is often resolved by

using the dependency injection construct. A module defines its requests to a sub-module (dependency) using an abstract interface, while the specific implementation of a sub-module is supplied either at compilation, or runtime. C++ supports using both. Whereas dynamic binding can be achieved by virtual methods, static one might be done using templates. For performance critical applications it is better to use static binding, rather than dynamic one, to avoid late binding overhead. In this case, the sub-module is known at the compilation time, thus the static binding may be used.

The major requirement of unit testing is getting the same results every time, that is why the audio recorder needs to be replaced with a static audio recording. For this reason, the system audio recorder is replaced with `wav` format loader. A recording is loaded using modified AudioFile[2] library. Then, samples are framed using the method as the original audio recorder and passed to preprocessing. Finally, the output of every step is compared with results produced by the reference implementation.

[2]By Adam Stark, GPL licensed. https://github.com/adamstark/AudioFile

Chapter 5

Conclusion

In this work, we designed the fundamental components of a speech recognition engine for Android and compared the particular approaches. Consequently, we created a library that might be used with various acoustic and language models trained for the desired purpose. The accuracy of the recognition system is admittedly determined by the used models. Therefore, the most constructive merits for the evaluation of the results of this work is the size of models supported for the real-time recognition.

Using a recent, mid-end smartphone equipped with chipset Qualcomm Snapdragon 625, the recognizer can handle a dictionary of 12500 words, 1.69 million bi-grams and 1.91 million tri-grams with 4000 live tokens utilizing two CPU cores. While using the proposed, custom implementation of the uni-gram recognition network, the task might be parallelized among more CPU cores if required. With this configuration, it is possible to support basic conversational models of spoken language. Using these models and the same set of features, the error rate of the dynamic decoder is comparable with the static implementation.

We believe that the results of this studies, as well as the implemented library, will serve as a valuable resource for further investigation and deployment of automatic speech recognition on mobile phones and embedded devices. This work, along with the results and the live demo has been presented and awarded at student conference Excel@FIT 2018[1].

5.1 Further work

Considering the complexity of the project, there is still a lot of space for further optimizations. Including data structure and algorithm optimizations as well as the platform-specific ones, what might be the subject of the future studies. Speaking of Android, it would be worth it to consider using Neural Networks API[2] introduced in Android 8.1. Such interface could provide better performance in phoneme posteriors estimation and support for bigger networks. Outwith the implementation part, there is admittedly a lot of work to be done on acoustic modeling and developing the neural network to provide better features for the decoder, but still preserving the reasonable computational complexity. For this purpose, it would be convenient to provide support for standardized neural network formats (e.g. ONNX[3]).

[1] http://excel.fit.vutbr.cz/
[2] https://developer.android.com/ndk/guides/neuralnetworks/
[3] https://onnx.ai/

Bibliography

[1] Chassaing, R.; Reay, D.: *Fast Fourier Transform*. John Wiley & Sons, Inc.. 2007. ISBN 9780470238141. pp. 255–318. doi:10.1002/9780470238141.ch6. Retrieved from: http://dx.doi.org/10.1002/9780470238141.ch6

[2] Grézl, F.; Karafiát, M.; Burget, L.: Investigation into bottle-neck features for meeting speech recognition. In *Proc. Interspeech 2009*. 9. International Speech Communication Association. 2009. ISBN 978-1-61567-692-7. ISSN 1990-9772. pp. 2947–2950. Retrieved from: http://www.fit.vutbr.cz/research/view_pub.php?id=9038

[3] Jr., L. E. F.: Chapter 10 - Audio Electronics: Digital Voice and Music Dominate. In *Electronics Explained*, edited by L. E. Frenzel. Boston: Newnes. 2010. ISBN 978-1-85617-700-9. pp. 229 – 246. doi:https://doi.org/10.1016/B978-1-85617-700-9.00010-2. Retrieved from: https://www.sciencedirect.com/science/article/pii/B9781856177009000102

[4] Katz, S.: Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE Transactions on Acoustics, Speech, and Signal Processing*. vol. 35, no. 3. Mar 1987: pp. 400–401. ISSN 0096-3518. doi:10.1109/TASSP.1987.1165125.

[5] Rousseau, A.; Deléglise, P.; Estève, Y.: TED-LIUM: an Automatic Speech Recognition dedicated corpus. In *LREC*. 2012.

[6] Rousseau, A.; Deléglise, P.; Estève, Y.: Enhancing the TED-LIUM Corpus with Selected Data for Language Modeling and More TED Talks. In *LREC*. 2014.

[7] Russell, S.; Norvig, P.: *Artificial Intelligence: A Modern Approach*. Upper Saddle River, NJ, USA: Prentice Hall Press. third edition. 2009. ISBN 0136042597, 9780136042594.

[8] Schwarz, P.; Matějka, P.; Černocký, J.: Towards Lower Error Rates In Phoneme Recognition. *Lecture Notes in Computer Science*. vol. 2004, no. 3206. 2004: pp. 465–472. ISSN 0302-9743. Retrieved from: http://www.fit.vutbr.cz/research/view_pub.php?id=7483

[9] VESELÝ, M.: *Dynamic decoder for speech recognition*. Master's Thesis. Brno University of Technology, Faculty of information technology. Brno. 2017. supervisor Schwarz Petr.

[10] Young, S.; Russell, N.; Thornton, J.: Token Passing: a Simple Conceptual Model for Connected Speech Recognition Systems. Technical report. 1989.