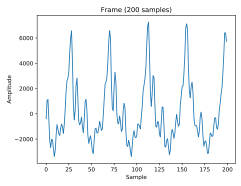
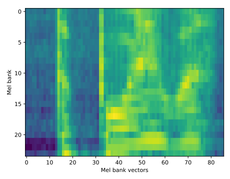
BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

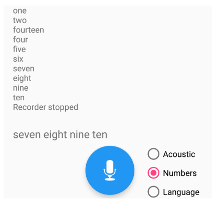
Excel@FIT 2018
http://excel.fit.vutbr.cz

Implementation of a simple speech recognition engine for Android

Eduard Čuba






one
two
fourteen
four
five
six
seven
eight
nine
ten
Recorder stopped

seven eight nine ten

Acoustic [✓ Numbers] Language

Abstract
The aim of this project is to implement speech recognition software for Android platform. This paper outlines fundamental components of a speech recognizer and reviews techniques used to optimize the process of speech recognition on Android devices. At first, it examines the implementation of the audio and speech feature extraction process. Then, it describes the design and implementation of a decoder used to process speech features into transcription utilizing only limited resources of a mobile device. The project is split into modules formed into an Android library, which should be easily expandable and equipped 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 ways of further development and applications of this project.

Keywords: Speech recognition – Dynamic decoder – Android – NDK – mobile devices

Supplementary Material: Demonstration Video

xcubae00@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

The domain of automatic speech recognition is one of the most discussed topics in the world of computer science. However, most of the speech recognition is still performed in the cloud by powerful and often specialized hardware. 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 project 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 concern is the limited amount of resources of conventional 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 feature extraction and decoder 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.

From the existing solutions, it is worth mentioning the Android internal speech recognizer, which may work in offline mode. However, it is more suitable for voice commands rather than continual recognition, and it is not possible to equip the recognizer with custom models. On the other hand, there is an open-source alternative PocketSphinx[1], a lightweight implementation of CMUSphinx toolkit. But still, being quite complex and complicated makes it hard to use for not involved developers, and it is mostly used with small models for application control purposes.

In this project, we focus on an implementation of a simple speech recognition engine, including all the mandatory components for reasonable recognition results. The acoustic model is based on a feed-forward neural network, which might be trained for a particular purpose. Similarly, we take the statistical approach to language modeling, enabling the user to deliver his own language model based on recordings and transcriptions from a 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 resources for the desired use.

Particularly, we focused on an implementation of a phoneme posterior probability extraction based on a neural network and dynamic decoding, that might run in real-time on a mobile device. 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. Audio processing

The very first step of speech recognition itself is to obtain a source recording. In order to use full computing potential of a target platform, the whole process is implemented in a language, that might be compiled to low-level code for the selected architecture. Having Android as a target platform, we used toolkit Android Native Development Kit (NDK) with the C++ programming language. NDK toolkit provides an interface to system audio devices using OpenSL ES framework, including the system audio recorder.

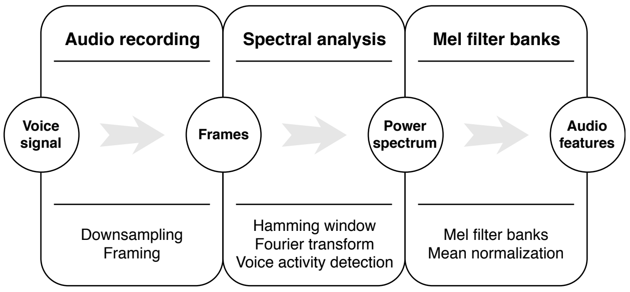


Figure 1. A block diagram of audio processing

For the speech recognition system, we use a frequency band from 64Hz to 3800Hz. 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 8000Hz a good 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 44100Hz. 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 sample rate of 8kHz is supported on most of the devices, the actual list of supported sample rates is vendor specific. Therefore, in order to support all the devices, it is necessary to provide the recorder with a custom resampler including an appropriate low-pass filter to avoid aliases.

2.1 Spectral analysis

To get an estimation of frequencies involved in a speech over time, the speech has to be cut into smaller frames, on which spectral analysis will be performed. A common approach to audio framing for speech recognition is to use 25ms long frames with 10ms step. Cutting down 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.

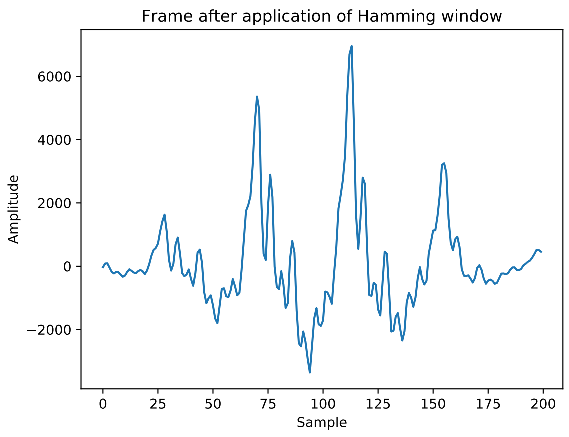


Figure 2. A speech frame after application of Hamming window function

Later on, data about frequencies involved in frames are used for audio feature extraction. Fundamental frequencies in a frame are accentuated by computing a power spectrum of the signal.

2.2 Voice activity detection

In order to save device resources and enhance the results of mean normalization, we use simple voice/silence detection. If the recorder is in a silence state, it is possible to suspend both feature extraction and decoding, keeping CPU usage by the recognizer close to none.

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

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

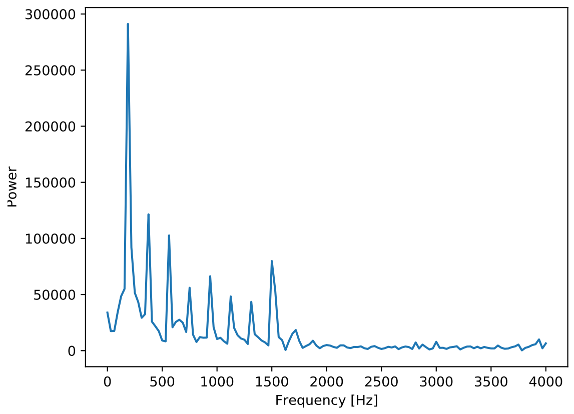


Figure 3. Power spectrum of the frame

This detection is performed by computing the energy of each frame and comparing against a statistically chosen threshold. After series of silent frames of a specified length, the recognizer enters a suspended mode, in which incoming frames are stored to a queue of restricted length. This queue is used for smoothing the transition to a active state. Since the threshold might be crossed by short, but intensive sounds like clapping or typing on a keyboard, it is convenient 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 to be 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 a rolling window width on feature extraction model input, increased by a number of frames required for speech activation.

2.3 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. Thus, the power spectrum is divided into 26 overlapping bands, on which 24 triangular filters called banks are applied.

2.4 Mean normalization

To find out whether specific Mel banks are significant in particular frames, their values have to be normalized – showing the difference between bank mean and its current value. Since the background noise may change over time, it is necessary to normalize against recent samples only.

We used exponential function $f(x) = b^x$ with base $b \in (0,1)$ to restrain the negative impact of older frames by decreasing their weight over time. For example, if a 5 second old sample should be included with weight 0.5, using sample rate 8000, and frame step

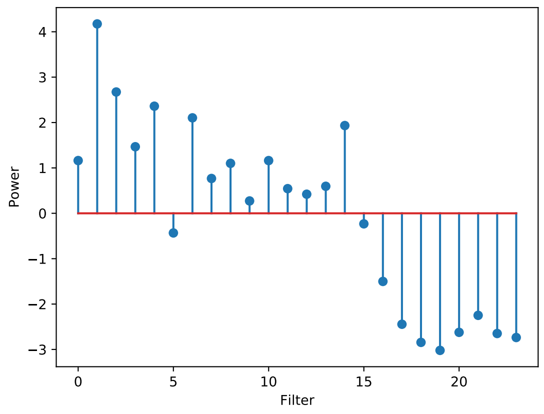


Figure 4. Normalized Mel filter bank values of the frame

80 samples, than 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 can be expressed by the following formulas:

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

$$M_{n+1}[m] = \frac{S_{n+1}[m]}{\int_0^\infty b^x dx} \tag{2}$$

Formula 1 computes a weighted sum of the samples in a new time step. Subsequently, formula 2 computes new mean value M by dividing sample sum by total weight represented by area under the exponential curve of $f(x) = b^x$ from zero (new sample) to infinity.

3. Speech feature extraction

The process of speech feature extraction describes a transformation of acoustic features retrieved from the audio signal to speech features used for language modeling. In this work, these features are represented by single state posterior probabilities of phonemes involved in particular speech frames. The speech feature extraction model is defined by a feed-forward artificial neural network (ANN), which may be trained for a task-specific purpose. We use split context ANN [1] with a bottle-neck inside [2]. For the most part, the performance of feature extraction depends on the size of a neural network. Stacked filter banks are used as an input to the neural network. Each input vector represents the progress of individual Mel filter banks over a time period (15 frames).

Admittedly, this part is one of the most significant bottlenecks in the speech recognition process. Conventional speech recognizers often use more than 2000 perceptrons in hidden layers of the neural network, what is hardly computable in real-time on a mobile device. The practical output of this part is a system built

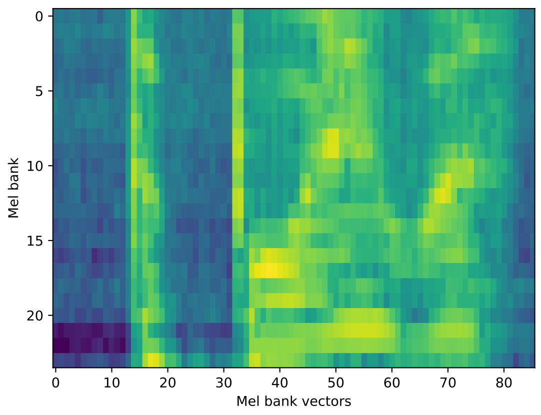


Figure 5. Visual representation of stacked Mel filter banks over period of 90 samples.

for layer-wise parallelized computation of phoneme posterior probabilities using a neural network.

3.1 Performance comparison

For testing purposes, we used an five-layer ANN, including one bottleneck layer, trained on TED-LIUM corpus [3]. Since the size of a single layer can be more than 500 perceptrons, the task might have to be parallelized among several CPU cores. We discovered, that for bigger networks (more than 500 perceptrons in a layer) it is necessary to use appropriate single instruction multiple data (SIMD) instruction set with single-precision floating point representation (32 bit). Specifically, we used SIMD instruction set extension NEON on 64-bit ARM architecture (used in 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 support of vectorization on a platform given. Since most of the Android devices use ARM architecture, we decided to use C++ library Eigen with native NEON support. In table 1 and figure 6, we compare the time required to perform feature extraction of 30 second long recording on a single-core – using chipset Qualcomm MSM8953 Snapdragon 625 (Octa-core 2.0 GHz Cortex-A53).

Table 1. The time required to perform a feature extraction from 30-second long recording on a single core using a 5-layer neural network with 500 perceptrons in the hidden layer.

Single precision (32b)		Double precision (64b)	
NEON	no vectorization	NEON	no vectorization
29.47s	55.49s	56.17s	61.37s

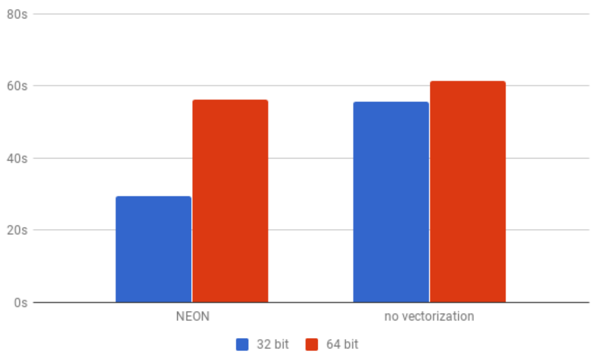


Figure 6. Comparison of time required to perform a feature extraction using 32-bit (blue) or 64-bit (red) data types with and without vectorization.

Since the process might be layer-wise parallelized, the main restriction is the single core performance, which must be taken into consideration when designing a neural network for this purpose. In figure 7 we compare the time required to process a single layer of a neural network on a single core. According to these measurements, we can see, that a reasonable number of perceptrons in a hidden layer of a five-layer neural network would be about five to six hundred. This configuration would lead to utilization of approximately two CPU cores in an active state.

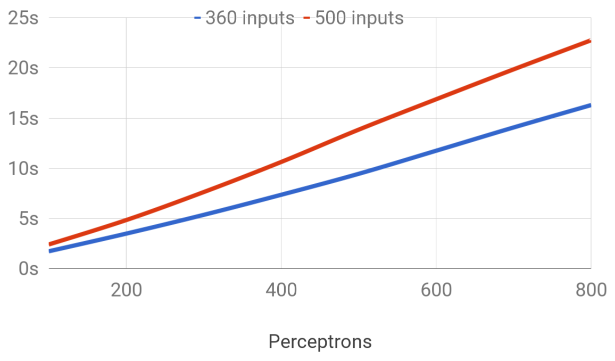


Figure 7. Comparison of the time required to process a single layer of a neural network in relation to the size of the layer. The number of perceptrons in a layer is shown on the bottom axis, layers with 360 inputs are drawn by a blue color, whereas the ones with 500 inputs are drawn by a red color.

4. Design of the decoder

The major problem regarding 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 the price of making the recognition procedure more demanding on computing power. However, considering the performance of modern smartphone processing units, this approach may be applicable to even quite big dictionaries and n-gram models.

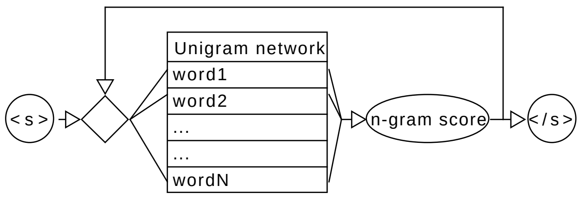


Figure 8. A sample representation of a dynamic decoder with unigram network and dynamic n-gram probability application.

The goal of this part of the work is to create an implementation of the dynamic decoder, that will be able to handle decoding with both limited-vocabulary and larger – conversational models.

4.1 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 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. The custom implementation makes it possible to update tokens in-place, avoiding unnecessary copies, and also provides support for row-wise parallelization, unlike the other dynamic structures. Table 2 shows the performance and memory comparison of the particular data structures.

Usage of a 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. This might

Table 2. The time and memory required to decode features extracted from 75-second long recording on a single core using a uni-gram network only, a vocabulary of 12000 words, 2000 live tokens and a 100 pruning beam.

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

compromise the results of a recognition process. For this reason, as proposed in work [4], 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 this more efficient, we modified our custom implementation by adjusting the behavior of a reversed singly linked list in a way, that tokens in the same position will not be replaced unless their word path is the same. The performance of these data structures is reviewed in table 3.

Table 3. The time required to decode features extracted from 75-second long recording on a single core using a 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

4.2 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 provide 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 to implement custom hashing function for tuples of various sizes. We used a solution for n-gram storage design proposed in work [4]. Where n-grams are stored in an array ordered by word identifier, and there is an additional array to store ranges of n-gram indexes for particular words. Specific n-gram is then found by binary search within the range of a single word. Table 4 shows the performance and memory comparison of these n-gram storage designs.

Table 4. The time required to perform 25M random lookups in n gram storage filled with 5M bi-grams.

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

4.3 Comparison with a static decoder

In comparison with a static decoder using the same features extracted from TED-LIUM recordings, the dynamic decoder performed slightly worse – the results are shown in table 5. However, the implemented dynamic decoder should be able to compete with conventional static decoders. Assuredly, 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.

Table 5. The word error rate computed on a subset of TED-LIUM test set using a dynamic and static decoder with the same configuration (4k tokens and 150 beam pruning) and a vocabulary of 12k words supplemented by a 3-gram language model.

Decoder type	WER[%]
Static	69.42
Dynamic	74.04

5. Conclusions

In this work, we designed and implemented fundamental components of a speech recognition engine for Android and compared various approaches. Particularly, we focused on the efficient implementation of neural network computations used for phoneme posterior probability estimation, representation of data structures used by the dynamic decoder and optimizations of decoding process itself. Consequently, we created a library that might be used with various acoustic and language models trained for the desired use. The accuracy of the recognition system is admittedly determined by used models, therefore, the most constructive metric for evaluation of the results is the size of the models supported for the real-time recognition.

Using a recent mid-end smartphone equipped with chipset Qualcomm MSM8953 Snapdragon 625, the recognizer can handle a vocabulary of 3700 words, 0.835M bi-grams and 1.52M tri-grams with 4000 live tokens on a single core during constant voice activity. While using the proposed custom implementation of a uni-gram recognition network representation, the task might be fully parallelized among several CPU cores. With this configuration, it is possible to process the given amount of workload per core and support basic conversational models of spoken language with a vocabulary of appropriately 12000 words utilizing 4 CPU cores.

Considering the complexity of the project, there is still a place for further optimizations, including data structure and algorithm as well as the platform-specific ones, what might be a subject of the future work. Speaking of Android, it is worth to mention RenderScript[3] framework, which can handle parallelization of linear algebra computations and delegate some parts of the process to graphics processing unit (especially the vector and matrix operations). Outside of the implementation part itself, it will be convenient to provide support for standardized neural network (e.g. ONNX[4]) and language model formats to make it easier for developers/researchers to train their own models and integrate this speech recognition library with their application.

Acknowledgements

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

References

[1] Petr Schwarz, Pavel Matějka, and Jan Černocký. Towards lower error rates in phoneme recognition. *Lecture Notes in Computer Science*, 2004(3206):465–472, 2004.

[2] František Grézl, Martin Karafiát, and Lukáš Burget. Investigation into bottle-neck features for meeting speech recognition. In *Proc. Interspeech 2009*, number 9, pages 2947–2950. International Speech Communication Association, 2009.

[3] Anthony Rousseau, Paul Deléglise, and Yannick Estève. Enhancing the ted-lium corpus with selected data for language modeling and more ted talks. In *LREC*, 2014.

[4] Michal. VESELÝ. Dynamic decoder for speech recognition. Master's thesis, Brno University of Technology, Faculty of information technology, Brno, 2017. Supervisor Schwarz Petr.

[3]https://developer.android.com/guide/topics/renderscript/index.html
[4]https://onnx.ai/