Analysis and implementation of a web-based Speech-to-Text system with DeepSpeech using freely available software components

Bachelor Thesis WS2019/2020

im Studiengang

Medieninformatik

Hochschule der Medien Stuttgart

Erstprüfer - Professor Dr. Fridtjof Toennissen
Zweitprüfer - Frederik von Berg

Vorgelegt von Johannes Beiser, Matrikel-Nr. 33570

12.02.2020

Contents

List of abbreviations **7**

Ehrenwörtliche Erklärung **8**

Abstract **9**

Kurzfassung **9**

1 Introduction **10**

1.1 Motivation and relevance . . . 10

1.2 Objective . . . 10

1.3 Structure . . . 11

2 Analysis and technical foundations for speech recognition systems **12**

2.1 Requirements and comparison between existing technologies . . . 12

2.1.1 Requirements for a Speech-to-Text engine . . . 12

Classification . . . 12

Transcribing Audio . . . 12

2.1.2 Comparison of the existing speech recognition technologies . . . 12

Google Cloud Speech API . . . 12

TensorFlow.js . . . 13

DeepSpeech . . . 14

Kaldi . . . 16

Evaluating the effectiveness of a speech recognition system . . . 16

2.1.3 Corpora for training a speech recognition model . . . 18

LibriSpeech . . . 18

CommonVoice . . . 18

Selecting the corpus . . . 20

2.2 Technical foundations for DeepSpeech . . . 20

2.2.1 Artificial neural networks . . . 20

Abstract functionality of a neural network . . . 20

Recurrent neural network . . . 21
Supervised learning . . . 22
End-to-End automatic speech recognition . . . 22
2.3 The DeepSpeech architecture based on Mozilla . . . 23
2.3.1 Abstract architecture . . . 23
2.3.2 Required input format . . . 24
2.3.3 The acoustic model of DeepSpeech . . . 24
Structure & task of the RNN . . . 24
Beam search decoder . . . 25
Training of an acoustic model . . . 27
Pre-trained acoustic model . . . 27
2.3.4 The language model of DeepSpeech . . . 28
Language model provided by Mozilla . . . 29
Creating a custom language model . . . 29
2.3.5 Overview . . . 30
2.3.6 DeepSpeech in other programming languages . . . 30

3 Analysis of suitable web technologies for DeepSpeech 31
3.1 Language choice for the backend . . . 31
3.2 Requirements for DeepSpeech . . . 31
3.3 Choice of framework and browser API's for the frontend . . . 31
3.3.1 Choice of the framework . . . 32
3.3.2 Choice of browser API's . . . 32
Media Capture and Streams API . . . 32
MediaRecording API . . . 33
Web Audio API - Client-side resampling . . . 34
3.4 Transmission of audio from client to server . . . 35
3.4.1 WebRTC . . . 35
3.4.2 XHR . . . 36
3.4.3 WebSocket . . . 36

3.5 Server side stream handling . . . 37
3.5.1 FFMPEG . . . 37
3.5.2 VAD . . . 38
3.5.3 DeepSpeech streaming . . . 39
3.5.4 SOX . . . 39
3.6 General summary . . . 40

4 Implementation 42
4.1 Introduction demo application . . . 42
4.1.1 Implementation of the elaborated concepts in a demo application . . . 43
4.2 Implementation Frontend . . . 44
4.2.1 Access to the user's microphone . . . 44
4.2.2 Obtaining binary data from the audio stream . . . 45
4.2.3 Client side resampling . . . 46
Resampling audio frames . . . 46
Buffering of the resampled audio frames . . . 47
Overview resampling . . . 49
Speech synthesis . . . 50
4.3 Client-Server communication . . . 50
4.3.1 XHR . . . 50
Client . . . 50
Server . . . 51
4.3.2 WebSockets . . . 52
Client . . . 52
WebSocket Server . . . 53
4.3.3 Connection establishment . . . 54
4.4 Implementing the web servers . . . 56
4.4.1 General setup . . . 56
4.4.2 Pipelining and transcoding the audio stream . . . 56
Creating a ReadStreams . . . 57
FFMPEG . . . 57

VAD 58
4.4.3 Integrating DeepSpeech in Node.js 59
4.4.4 Transcribing the buffer with DeepSpeech 60
Basic transcription of a buffer from a finished recording 60
Streaming in DeepSpeech 61
4.4.5 Creating a custom language model 62
4.5 DeepSpeech client-only 63

5 Conclusion **64**

References **66**

List of Figures

1 TeachableMachine demo project . . . 13
2 DeepSpeech and CommonVoice . . . 15
3 Topology of an artificial neuronal network . . . 21
4 Topology of a recurrent neural network . . . 22
5 Topology of the RNN of DeepSpeech . . . 25
6 CTC process . . . 26
7 Proportion of corpora used for the pre-trained acoustic model of DeepSpeech . . . 28
8 Internal process for inference in DeepSpeech . . . 30
9 Browser compatibility of MediaStream API . . . 33
10 Browser compatibility of MediaRecorder API . . . 34
11 Web architecture roundtrip with server side resampling . . . 40
12 Web architecture roundtrip with client side resampling . . . 40
13 Design of the cocktail machine . . . 42
14 User interface of the demo application . . . 43
15 Overview resampling process . . . 49
16 Timing of connection establishment for streaming with WebSockets . . . 54

List of Tables

1 Word error rate of DeepSpeech . . . 17
2 Word error rate of different ASR-Systems . . . 17

List of abbreviations

AI - Artificial intelligence

STT - Speech-to-Text

ASR - Automatic Speech Recognition

SNR - Signal to noise ratio

ANN - Artificial neural network

RNN - Recurrent neural Network

E2E - End-to-End

CTC - Connectionist Temporal Classification

WER - Word Error Rate

BLOB - Binary Large Object

XHR - XMLHttpRequest

P2P - peer-to-peer

VAD - Voice activity detection

PCM - Pulse code modulation

EOF - End of File

MFCC - Mel Frequency Cepstral Coefficients

Ehrenwörtliche Erklärung

Hiermit versichere ich, Johannes Beiser, ehrenwörtlich, dass ich die vorliegende Bachelorarbeit mit dem Titel: „Analysis and implementation of a web-based Speech-to-Text system with DeepSpeech using freely available software components“ selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden. Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§ 24 Abs. 2 Bachelor-SPO (7 Semester) der HdM) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.

Stuttgart, den 12.02.2020

Abstract

In recent years, increasingly more software solutions have been developed for the web due to the popularity of web browsers and ease of use. The use of voice assistants has also increased significantly. The main objective of this thesis was to unite these two worlds by using the Speech-to-Text engine DeepSpeech and other freely available open-source components. It will be explained how DeepSpeech works and how the engine can be adapted to individual needs. The decision for the different components will be evaluated on the basis of a prototypical implementation, so that at the end a demo application is available which can transcribe the user's speech inputs in real-time. Client and server-side approaches will be evaluated to allow different application scenarios.

Keywords: *Speech recognition, Speech-to-Text, DeepSpeech, Open-source*

Kurzfassung

In vergangenen Jahren werden aufgrund der großen Verbreitung von Webbrowsern immer mehr Softwarelösungen für das Web entwickelt. Auch die Verbreitung von Sprachassistenten hat stark zugenommen. Gegenstand dieser Arbeit soll es sein, diese beiden Welten durch die Verwendung der Speech-to-Text Engine DeepSpeech miteinander zu vereinen. Dabei wird das Ziel verfolgt ausschließlich frei zugängliche open-source Komponenten einzusetzen. Es wird erläutert wie DeepSpeech grundlegend funktioniert und wie man die Engine eigenständig den eigenen Bedürfnissen anpassen kann. Die Wahl der dafür benötigten Komponenten wird anhand einer prototypischen Implementierung auf ihre Eignung und Einsatzmöglichkeiten geprüft, sodass am Ende eine Beispielanwendung bereitsteht welche Spracheingaben des Nutzers in Echtzeit transkribieren kann. Es werden dabei client- und serverseitige Ansätze evaluiert um unterschiedliche Einsatzmöglichkeiten zu ermöglichen.

Schlagwörter: *Spracherkennung, spache zu text, DeepSpeech, Open-Source*

1 Introduction

1.1 Motivation and relevance

In recent years the world of software development has been revolving more and more around the Web. Requirements for web applications are increasing, which is why new solutions are constantly being developed. Especially at times when voice assistants are gaining more and more ground through smart homes, the desire to integrate voice control into one's own web application is also increasing. So far, this has mainly been possible via external API's from large companies like Google or with more complex open source systems. For privacy and security reasons this is not an option for all projects and more complex open source systems require more powerful server side hardware and maintanance. Hence, a simple open source solution was needed. Recently, a Speech-to-Text (STT) engine[1] developed by Mozilla based on an architecture introduced by Baidu[2] called "DeepSpeech" has been released, which can be used by everyone for free. If and how this new technology can be used for the web and which new possibilities it offers will be examined in this thesis.

1.2 Objective

The goal of this thesis was to allow the user to convert speech to text in real time in a demo application. Only freely accessible software components may be used, so that the developer has full control over the entire process - from recording the speech via the user's microphone to transcribing the speech via a neural network[3]. The main challenge is to identify the most suitable components and to make the communication between them as efficient as possible.

In particular, the following questions are discussed in this thesis:

- How does DeepSpeech work?
- Where does the data come from that is used for the training of the STT-engine?
- Which software components can be used to integrate DeepSpeech into a web application?
- In what different ways can DeepSpeech be used?

[1] An API based software component capable of transcribing speech into text
[2] One of the largest AI and internet companies in the world based in china
[3] An artificial intelligence component which is capable of making a prediction based on some kind of input

1.3 Structure

Chapter 2 of this paper will first analyze how DeepSpeech compares to other STT engines and what requirements need to be met. It also explains the exact architecture and components of DeepSpeech, how they work together and how it differs from traditional solutions.

After the basics to understand DeepSpeech are covered, chapter 3 analyzes how DeepSpeech interacts with suitable web technologies and how it can be integrated into a web application.

In the last - the practical - part of this thesis the chosen technologies will be used and implemented in a demo application so that at the end a prototype is available which is ready to convert speech input via the user's microphone from the browser to text via the self-implemented web server with the DeepSpeech.

2 Analysis and technical foundations for speech recognition systems

This chapter will analyze which technologies are suitable to ensure robust[4] speech recognition. In particular explaining the individual components of a speech recognition system and how they interact with each other to make the decisions for the different options clearer.

2.1 Requirements and comparison between existing technologies

2.1.1 Requirements for a Speech-to-Text engine

Classification

Probably the most basic requirement of a speech recognition system is to be able to distinguish between different words. This can be important if it only needs to recognize words within a domain of words, for example the recognition of control commands for a game (“up”, “down”, “left”, “right”).

Transcribing Audio

More complex speech recognition systems are expected to convert the speech directly into text. In terms of complexity, this differs from simply transcribing already fully captured recordings in clear spoken language to transcribing difficult to understand accents and dialects with background noise and poor SnR (signal-to-noise-ratio). In addition, an advanced STT engine should be able to process a continuous stream of audio in real-time. During speech, the recognized words should already be transcribed and output by the STT engine.

2.1.2 Comparison of the existing speech recognition technologies

Google Cloud Speech API

Even though the Google Cloud Speech API is not an option for this thesis because it is a paid API which is not owned by the open-source community but a product of Google, this option is still listed because it is a very popular and widely used way to convert speech to text on the web.

[4] speech recognition capable of working very well in various kinds of environments

TensorFlow.js

Tensorflow.js is a machine learning library for javascript. It is a javascript based implementation of the TensorFlow framework from Google for Python. It can be used "[...] for training and deploying machine learning models in the browser and in Node.js" [1]. If trained to create a speech recognition model it is able to classify words. To create a classification model the web interface of the project "TeachableMachine" can be used by speaking samples of words which the model should recognize [2]. This model has only a few megabytes and can therefore be provided on the client side which has the great advantage that it can also be used offline.

The figure shows an demo project which can distinguish between background noise and the words "Hello" and "World".

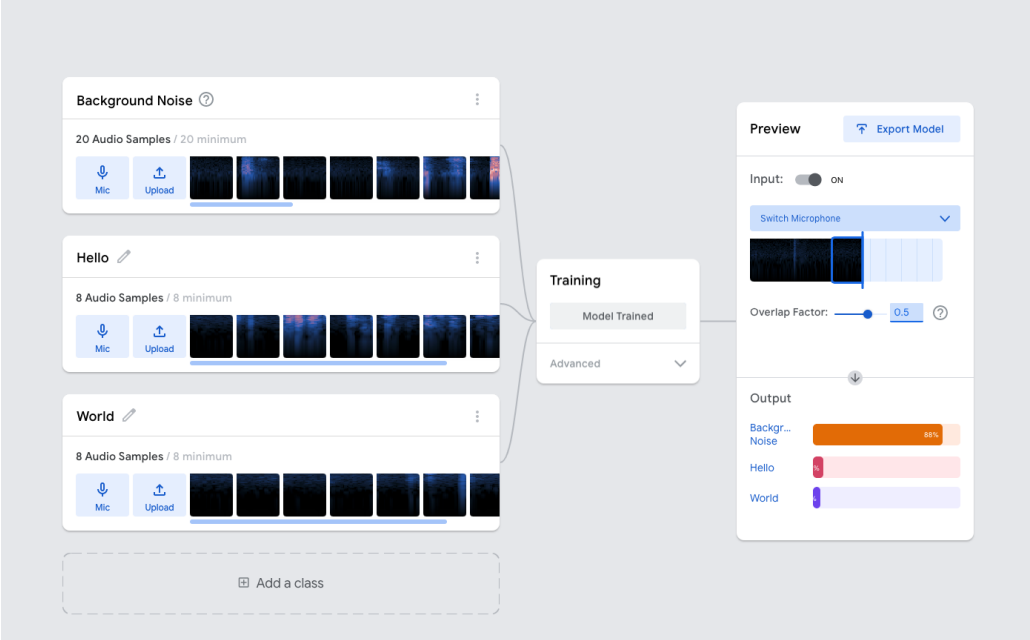

Figure 1: TeachableMachine demo project

Tensorflow.js is a low level solution with which neural networks can be trained easily and quickly. For more complex applications it is not suitable due to the problems mentioned in the following section.

At the moment it is only possible to recognize words with a duration shorter than one second. Using the web interface from figure 1, samples of the sound/word must be recorded. Once a sufficient number of recordings have been obtained, the model can be trained. The model may be sufficient for a particular case, but its great weakness is that it is exactly adapted to the pronunciation of the training material.

There is no corpora - a set of recordings - that specifically consists of recordings of individual words, so it is necessary that all words are recorded for training the model itself. For a robust model, it would not be sufficient to provide only the minimum number of eight example recordings [1]. Furthermore, the model is only able to compare how similar an input sounds to one of the possible outputs without having an understanding of letters or words. Therefore Tensorflow.js cannot be used for transcribing speech, just very basic classification.

If the machine-learning model should be able to understand new words it is necessary to train it further, which is a disadvantage compared to a more sophisticated system where it is often sufficient to enter the new vocabularies as a string, because the model already has a fundamental understanding about the pronunciation of letters and words.

Another problem is the task of classification itself. Ideally, the end user should only speak words that can be understood by the model. If a unknown word is spoken, the most similar sounding word is taken instead of ignoring it. Ideally, a model should be based on all words of a language and output only when certain words are recognized.

DeepSpeech

DeepSpeech is an open-source implementation by Mozilla based on the DeepSpeech speech recognition architecture introduced by the company Baidu. It is to be distinguished between the implementation “DeepSpeech” by Mozilla and the architecture “DeepSpeech” by Baidu. There are also many other implementations of this Architecture, though Mozillas implementation is the most sophisticated one [3]. Mycroft for example, a company developing an open-source voice assistent has switched to using DeepSpeech internally [4].

Although the implementation of Mozilla follows the DeepSpeech architecture quite closely, it deviates in some places. In addition, Baidu has meanwhile published another paper called “Deep Speech 2 : End-to-End Speech Recognition in English and Mandarin” , which also plays into the implementation of Mozilla in parts [5]. In the further course of this paper “DeepSpeech” refers to the implementation of Mozilla. In the research paper “Deep Speech: Scaling up end-to-end speech recognition”, published in 2014 by Baidu, an architecture is presented that allows to use a single artificial neural network to directly infer[5] text from an audio file. Why this is beneficial and what exactly neural networs are, will

[5] Inference is the process of making prediction based on a artificial neural network

be explained in chapter 2.2.1.

In the end of 2017 Mozilla released the first version 0.1.0 of this implementation. Currently, DeepSpeech is in version 0.6.0 which is still a very early stage of development. DeepSpeech supports both fixed-length audio transcription and an audio-stream interface that outputs the transcribed text of an audio stream in real time. DeepSpeech is implemented with TensorFlow, a machine learning framework from Google in Python.

The main idea behind DeepSpeech is to create a simple, yet effective Speech-to-Text engine that does not require expensive server hardware and runs efficiently even on devices with limited hardware resources. This is to be made available on different platforms and for various languages free of charge under the “Mozilla Public License” [6].

In 2017, Mozilla also started the crowdsourcing project CommonVoice in parallel to DeepSpeech in order to create a constantly growing corpus for DeepSpeech. Corpora are datasets containing speech and the corresponding transcription and are mainly being used for the training of automatic speech recognition (ASR) Systems. A further analysis of Corpora will be made in chapter 2.1.3.

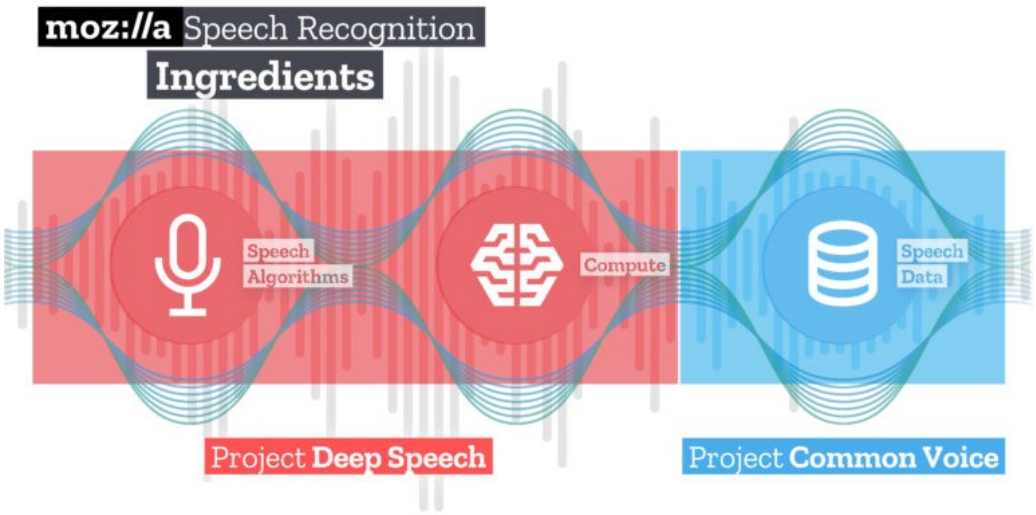

Figure 2: DeepSpeech and CommonVoice

This means that DeepSpeech can be used to transcribe any language as long as the training data is available for that language to train a model with. As an example, DeepSpeech is already being used to

develop a voice assistent for Maori [7].

DeepSpeech is available in several programming languages including Node.js. In future versions it is planned to be made available for the browser on the client side. On various platforms such as Windows, MacOS or Rasberry Pi, DeepSpeech is already available as a native client, but for web applications a web server is still required to run DeepSpeech.

Kaldi

Kaldi is a open-source toolkit for speech recognition targeted for researchers. It can be used to train speech recognition models and to decode audio [8]. While there are a several pretrained models provided for different languages, kaldi is mostly targeted for developers who want to create or improve their own model, with their own or another publicly available set of data. It is written in C++ and generally more complex and difficult to use compared to DeepSpeech, since it doesn't offer language-bindings for other programming languages [9]. However it is currently still the Speech-to-Text engine of choice for many companies who work in the field of open-source speech recognition since it was continuously improved since it was released in 2009 [10] [11]. In contrast to DeepSpeech developers can create different models and STT architectures with it. This is why Kaldi is more to be seen as a actual toolkit rather than a ready-to-use Speech-to-Text engine like DeepSpeech.

Evaluating the effectiveness of a speech recognition system

The accuracy of a speech recognition system is measured by the word error rate (WER). The goal of any speech recognition system is to get close to the WER of a human. Contrary to the expectation this is not 0% but about 5.8% [5]. To compare the different speech recognition systems there are tests based on different corpora which measure the WER. A corpus is the set of data a speech recognition system is trained on. For testing purposes it is possible to test a speech recognition system on a different set which it was not trained on. These tests result in different WER depending on the test set. Only at a deviation in WER around 5-10% a person will notice a quality gain or loss [12]. There are numerous comparisons between word error rates among the different speech recognition systems and it does not necessarily mean that one is superior to the other if it achieves a better result for one test set. For competitive reasons each system tries to present itself better than the other, which is achieved by choosing a special test corpus. This is the reason why many reported word terror rates are often

quite far from a real life scenario and lack meaningfulness, as the value simply states that this speech recognition system is very good at recognizing a special accent in a special scenario with a limited list of words known to the system [13]. In individual cases, such tests can make sense if the application area is isolated and consistent. To be able to draw a real conclusion one has to look at the WER of different test sets. The following table compares the results of different tests on the DeepSpeech 2 implementation of Baidu with the WER of a human for each test. The values of human WER are measured and provided by the respective test sets. The DeepSpeech implementation of Mozilla has only been measured by one test (LibriSpeech - clean) at 7.5%, which is very close to the WER of Baidu's implementation, therefore the measured values of Baidu are shown below as an example [5].

Category	**Test Set**	**DeepSpeech by Baidu**	**Human WER**
Clean Read	WSJ eval'92	3.10	5.03
Clean Read	WSJ eval'93	4.42	8.08
Clean Read	LibriSpeech test-clean	5.15	5.83
Clean Read	LibriSpeech test-other	12.73	12.69
Accented	VoxForge American-Canadian	7.94	4.85
Accented	VoxForge Commonwealth	14.85	8.15
Accented	VoxForge European	18.44	12.76
Accented	VoxForge Indian	22.89	22.15
Noisy	CHiME eval real	21.59	11.84
Noisy	CHiME eval sim	42.55	31.33

Table 1: Word error rate of DeepSpeech

In the following table, the different speech recognition systems are compared with the corresponding WER as an example. The tests were performed with an internal test set from Baidu, which is a combination of several test sets.

System	Clean	Noisy	Combined
Apple Dictation	14.24	43.76	26.73
Bing Speech	11.73	36.12	22.05
Google API	6.64	30.47	16.72
wit.ai	7.94	35.06	19.41
DeepSpeech (Baidu)	**6.56**	**19.06**	**11.85**

Table 2: Word error rate of different ASR-Systems

2.1.3 Corpora for training a speech recognition model

A corpus (plural corpora) is a collection of audio files linked to the corresponding text it contains. They can be used to train ASR models since they provide both the audio and the corresponding transcription.

There are several different corpora which are publicly available. The choice of audio data used to train the ASR model has a direct impact on how robust the system will be and what language variations it can understand [14]. For example, if one uses only clearly spoken English without background noise, the resulting model will recognize clear English very well, but will not be able to handle accents or background noise well. Also, the different corpora have different amounts of voice data. Some systems consist of over 1000 hours of speech, while others consist of only 100 or 200 hours. To train a sufficiently robust system, according to Baidu about 10.000 hours are required [5]. Where this data comes from does not matter as long as it is all in the same format and some other basic requirements are given.

When choosing the corpus / corpora, one should therefore make sure that it is adapted to the scope of the ASR system. Ideally it should be a corpus containing data with bad signal to noise ration (SNR) and strong accents to cover every case in a real world scenario. In the further course of this chapter an analysis will be made to what extent the currently existing corpora are suitable to train a robust an adapted speech recognition system.

LibriSpeech

LibriSpeech is the corpus that emerged from the LibriVox project. LibriVox is a collection of freely available audio books in mainly plain American English without background noise. LibriSpeech contains about 1000 hours of speech and is currently one of the corpora used for DeepSpeech . LibriSpeech is not suitable as the sole source for training a robust speech recognition because the corpus contains only noise-free and clear speech.

CommonVoice

To solve the problems mentioned in previous chapters, Mozilla started the CommonVoice project in june 2018. The corpus is currently still under construction and unsuitable as the sole source for training a speech recognition system since it doesn't yet contain enough data. CommonVoice is a

crowdsourcing project where one can contribute to the corpus by recording one' s voice. The web user interface has two modes - speaking and listening. When speaking, a short sentence oaf about 3-10 seconds is given which is then recorded. Once five of these sentences have been recorded, the recordings are shared with other users for validation. These recordings are then validated by other users of the platform who have selected the "listen" mode. One listens to 5 random recordings of other users and then indicates whether they match the given sentence. The quality of the recording does not matter as long as one understands each of the spoken words. If a recording has been validated by two different users, it is included in the data set. If the recording is rejected by at least two users, the recording is included in the so-called "Clip Graveyard" which is also available for download.

The platform tries to motivate users to be active by methods like gamification[6]. For example, one can set daily or weekly goals to achieve a certain number of recordings or validations and is also placed in a worldwide raking depending on how many of one's own recordings have been validated. The result of this platform is a corpus consisting of different accents and background noise levels. It should also be noted that the project not only collects English but also offers a lot of other languages due to the way it works. English currently is the most actively contributed language with about 1000 validated hours. thesis In total of all languages, about 2500 hours have been validated.

It is important to note that of the 1000 hours only 88 hours are unique. The reason for this is that until the beginning of 2019 there was only a very small amount of sentences, resulting in duplication, which is not optimal for training. In early 2019 Mozilla started the project SentenceCollector [15] which works after the same principle as CommonVoice to collect sentences from everyday language. In addition to this, 1.000.000 sentences were extracted from Wikipedia articles, so CommonVoice now has enough sentences to avoid duplicate recordings in the future. Criticism regarding this would be that the complexity of the sentences extracted from Wikipedia to be spoken exclude some users such as children or people who have difficulties with the language. Since the aim of CommonVoice is capturing the "common voice", the voice of everyday speech and speakers, this could be counterproductive.

Additional metadata such as the speaker's origin is stored with the respective recordings, if the speaker has registered before. This way it would even be possible to filter the data of a special accent in order to develop a speech recognition model for an exact target group or specific accent.

[6] Gamification commonly employs game design elements to improve user engagement in a non-game context

Selecting the corpus

The different corpora were introduced in order to have an understanding on how they can be different. There are many other corpora, including TED-LIUM, SWITCHBOARD and Fisher Corpus.
The choice of the suitable corpora for training a speech recognition system should be based on the target users for that speech recognition system. If one wants a quite specific system for a special accent of a language, it makes sense to adapt the training material to it. On the other hand, if one wants to use it for a wide range of speakers the ASR should be able to handle any kind of background noise, different environments and accents. In such a scenario it makes sense to use multiple corpora or even all corpora available in that language. CommonVoice would be the most suitable corpus in this case, although there is no reason not to use other corpora as well.

2.2 Technical foundations for DeepSpeech

This chapter explains the basics necessary to understand DeepSpeech and introduces some technical terms.

2.2.1 Artificial neural networks

Artificial neural networks (ANN) are based on the model of neurons in the nervous system of a human brain, although the primary objective is focused on creating models for information processing purposes rather than reproducing the actual biological structures. This chapter will describe a few types of neural networks that exist, how they function fundamentally and how they are trained.

Abstract functionality of a neural network

The basic task of an artificial neural network (ANN) is to conclude the respective output on the basis of an input. It consists of a set of artificial neurons which are connected to each other on different layers [16]. There is always an input and output layer. In between there are one or more so-called "hidden layers" which are responsible for the conclusion of the input. One layer of a neural network is responsible for one task. In the example of a ANN for image recognition, one layer could for example be responsible for transmitting a signal depending on the brightness, which is then used as input by the next layer of neuron. The topology of the network depends on the exact task of the neural network. The

topology describes the structure of the network which consists of the number of layers, neurons and the weighted connections of the neurons among each other. The connections between the neurons can have different weightings which determine how much influence one neuron has on the result. The output can be of any format. It could be a simple binary classification (right or wrong) or more complex output formats such as letters.

The following figure shows such an exemplary topology.

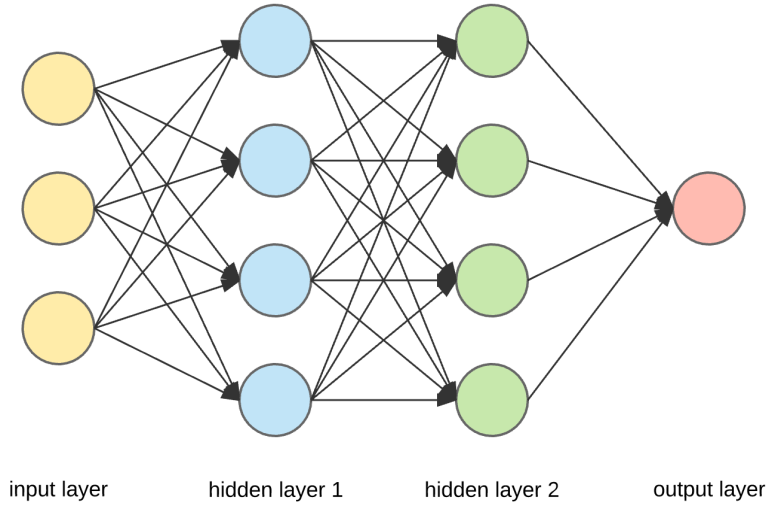

Figure 3: Topology of an artificial neuronal network

Recurrent neural network

The recurrent neural network(RNN) is a type of ANN. Compared to a simple neural network as shown in Figure 1, it differs in that the output of a neuron of a higher layer can be used as input of a neuron of a lower layer. This feedback makes sense for example if the input of the ANN changes over time [17]. An example would be a video stream as input of a ANN in which the result of a neuron at time t_1 is used as input of a neuron at time t_2. If the input of a neural network is changing over time, the topology of the network must determine which duration is used as a single input for the network. For the processing of a video stream, for example, one frame could be used.

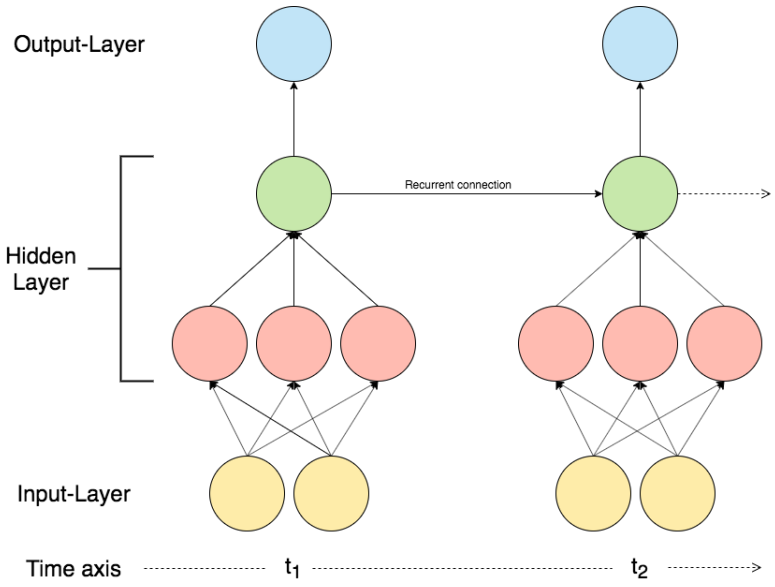

Figure 4: Topology of a recurrent neural network

The figure shows how the recurrent layer passes on information from the previous to the following iteration.

Supervised learning

Supervised learning is understood to be the training of a neural network with samples of input data and their corresponding results [18]. Using speech recognition as an example, one would feed pairs of speech recordings with their transcription into the neural network. The network first automatically infers the output value with the current neural network topology and compares this actual state with the target state which is given in the training data. Depending on the deviation from the actual to the target state, changes are made to the network topology and the next iteration starts.

End-to-End automatic speech recognition

Traditional speech recognition systems consist of different components such as the acoustic model, language model and a pronunciation model. These components are independently developed models and can be implemented differently depending on the architecture of the speech recognition system. In order to conclude an output, the input from the individual modules must be analyzed one after the other [19]. For example, so-called phonemes are used in traditional speech recognition pipelines.

A phoneme is a written representation of the corresponding pronunciation of a word. The acoustic model learns which phonemes are associated with a sound based on carefully entered training data with aligned words (words where the pronunciation of each letter of the word is aligned in time). Since it knows from the training data which sound is found at which position of the word, it is easy for the model to output a phoneme representation of the word. This output must then be passed further into the speech recognition pipeline and processed by the next model which for example converts a phoneme representation of a word into letters.
The development of such an architecture is very complex and every single model has its own problems. Each of the models must be optimized and trained independently, which can quickly lead to a bottleneck when used together. With End-to-End (E2E) ASR the goal is to limit these components to a single model so that the final output can be deduced directly from the input [19]. The advantage of E2E ASR is that only a single model needs to be trained, the output is the transcribed result, and no aligned recordings are needed for training in order to learn how to pronounce individual letters. This model is still called the “acoustic model” and will be referred to as such in the further course of this thesis.

2.3 The DeepSpeech architecture based on Mozilla

After providing some basic knowledge about speech recognition in the previous chapters, this section will explain how DeepSpeech works. The implementation of Mozilla follows Baidu's original Idea but it differs in some ways [20].

2.3.1 Abstract architecture

DeepSpeech uses a E2E recurrent neural network to infer the transcribed text directly from the audio input.
In the first step, the raw audio signal is converted into a suitable data format which is then analyzed by an acoustic model in sequences of 20ms, so that for each of these 20ms frames a probability for the letter is given. A decoder then calculates the most likely resulting word and word sequences from all the frames in conjunction with a language model.

2.3.2 Required input format

So called features are needed for further processing for the acoustic model. These features serve as input values for the RNN and are a mapping of the raw audio data to a specific data structure. As an input format, DeepSpeech produces a Mel Frequency Cepstral Coefficients (MFCC) from the raw PCM[7] audio signal, which is a type of spectrogram - a compact representation of the frequency spectrum of the audio signal. This allows DeepSpeech to make decisions with the neural network based on a few absolute values.

2.3.3 The acoustic model of DeepSpeech

The acoustic model of DeepSpeech basically consists of two parts. The RNN which outputs the probabilities of the individual letters and a beam search decoder which decides on the most probable result based on these matrices.

Structure & task of the RNN

After the MFCC is created the RNN iterates in 20ms frames over the entire length of the data or continuously over a stream. For each of these 20ms sequences a probability matrix is created which contains the probabilities of the possible symbols, for example the english alphabet. A decoder calculates the most probable sequence of symbols to determine the final word.

This described process is performed within the acoustic model so it can be used on its own to convert speech to text. Additionally, a language model can be used to gain further accuracy [20].

[7] Pulse-code modulation - the standard form of digital audio. The analog signal is sampled regularly at uniform intervals in order to oblain a uniform digital signal [21]

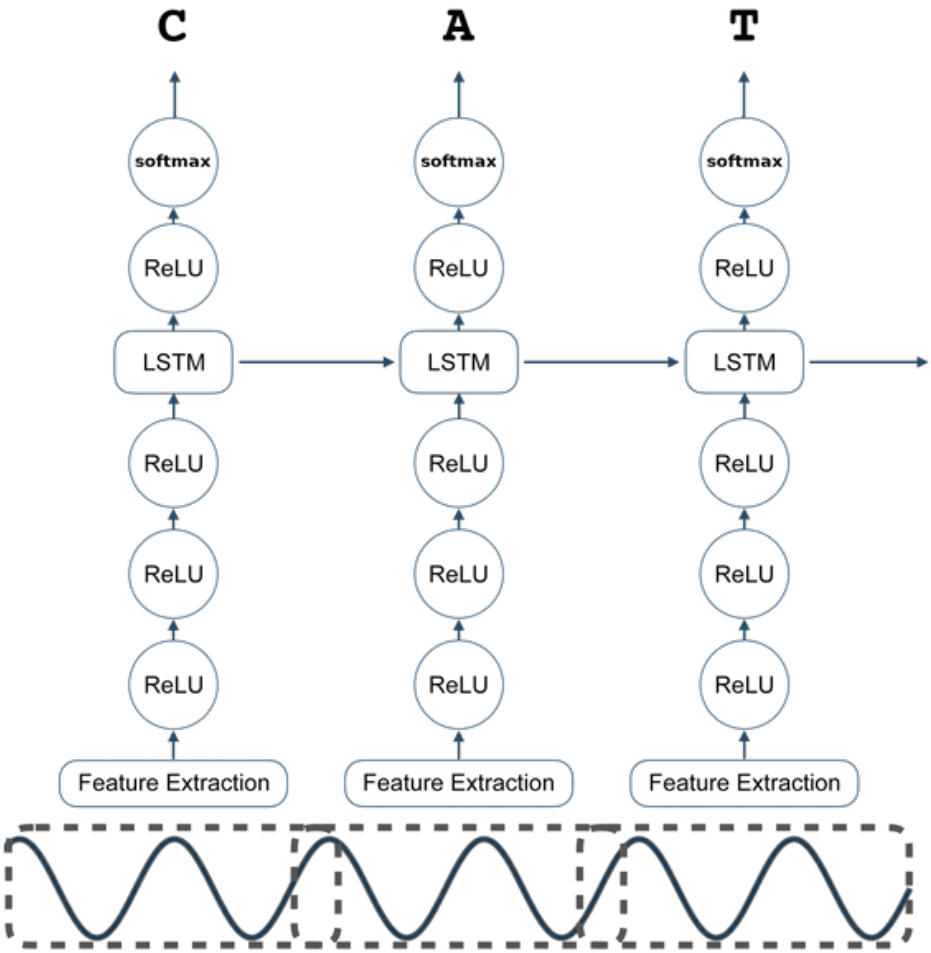

Figure 5: Topology of the RNN of DeepSpeech

Source: https://deepspeech.readthedocs.io/en/v0.6.1/DeepSpeech.html

Beam search decoder

A major challenge in converting speech to text is that words and letters can be pronounced at different speeds and do not occur sequentially in temporally equal segments that the neural network can process. This is a problem because everyday language is too unpredictable. To solve this problem DeepSpeech uses a Connectionist Temporal Classification (CTC) beam search decoder [22]. The RNN creates a transition matrix for each frame it iterates along the timeline, which determines how likely the frame is to contain the various symbols (letters & some special characters).

It would be easy to take the most probable symbol from each frame, but this would not be the most probable result. If a letter is pronounced longer, the RNN will recognize the same letter repeatedly. Example: “HHH__EL_LLL_O”.
The blank symbol “_”. is recognized by the RNN at optional letter boundaries, the blank (space) between words itself is a separate symbol. This output is reduced to “hello” with the the so called. The most probable result is therefore not the sequence of the most probable symbols, but the most probable sequence of symbols with the same result after applying the reduction.

An example would be these 3 possible results of which not “Cat” but “Hat” is the most probable result:
1. 45% probability for “C__AA_T”
2. 30% probability for “HH_A_TT”
3. 25% probability for “H__AA_T”

Since the second and third result in the same string “HAT”, alltogether it is more probable then “CAT”.

This reduction is performed through the following three steps.

1. predict a sequence of symbols.
2. repeating symbols are reduced to one if they are not separated by a blank (“_”).
3. blanks are removed.

Through these steps an example output of a RNN “HHH__EL_LLL_O” is reduced to “Hello”.

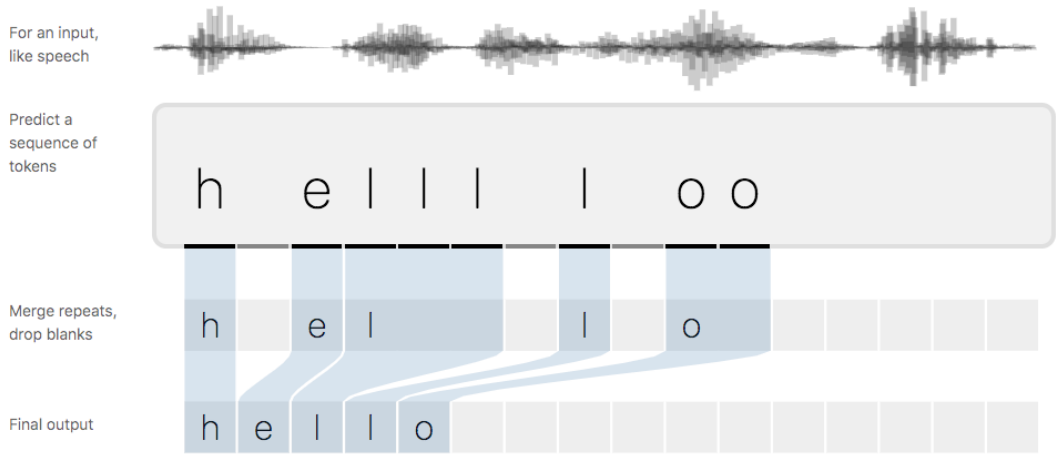

Figure 6: CTC process

In the case of DeepSpeech, the acoustic model already includes the E2E model, which infers speech directly into letters and words. This can easily be used on its own to transcribe speech, but it is useful to include a language model as described in the following chapter 2.3.4 to improve the output accuracy. Since DeepSpeech's acoustic model is already an E2E model, it directly outputs graphemes. A grapheme is the smallest semantically distinctive unit of a language. In the case of English this equals the alphabet a-z and some special characters.

Training of an acoustic model

It is possible to train a custom acoustic model or to continue training an existing pre-trained model provided by Mozilla from a checkpoint. All that is needed is the client provided by Mozilla, implemented in Python, and a corpus with sufficient data. To train an acoustic model that can be used productively, however, amounts of data sets are required that are not yet publicly available. For example, Baidu has recorded about 10.000 hours of speech in a variety of situations for its DeepSpeech2 [5] implementation. In addition to these 10.000 hours, another 100.000 hours of artificially created recordings were used. This was achieved by combining the 10.000 hours of original recordings with background noises of various kinds. For training with 1000 hours of audio alone, 20 hours are required with 4 standard graphics cards (e.g. NVIDIA GTX 1080 16gb). If this number is scaled to 100.000 hours of audio, it would take about 12 weeks for the fully trained model.

Even with 8 NVIDIA Titan X GPUs, which according to Baidu is the maximum possible parallelism for training, it still takes two weeks to train an entire model with this amount of data [5].

Pre-trained acoustic model

Mozilla provides pre-trained acoustic models that can be downloaded. These are versioned to be compatible with the different versions of DeepSpeech. Currently, the most recent model is available in version 0.6.0. When using this model, the version of this model must be compatible with the version of DeepSpeech. The 0.6.0 model was trained with the corpora LibriSpeech, Fisher, Switchboard and CommonVoice English. Altogether the corpora contain about 5000 hours of which 3816h were used as training material, 83% of which is american-english. The unused hours are either duplications or were used for development/testing purposes. Compared to the model from version 0.5.1 which was trained with only 1000 hours of LibriSpeech, the current model is based on a much larger amount of

data. Accuracy has improved greatly, although foreign accents are still problematic, as they are only present with 17% in the training data. With the LibriSpeech Clean test, the model achieves a WER of 7.5%, which is a good result but it is to be noted that the LibriSpeech test set only consists of American English without background noise. With slow and accurate pronunciation it is still possible to achieve somewhat good results with a foreign accent. There is no empirical evidence for this as there is no test set which tests for accuracy with foreign accents. This would require a corpus which consists solely of foreign accents. With CommonVoice this would be possible in the future as the corpus stores the individual recordings with additional information about the speaker's origin. In the following diagram the exact composition of the training data is summarized:

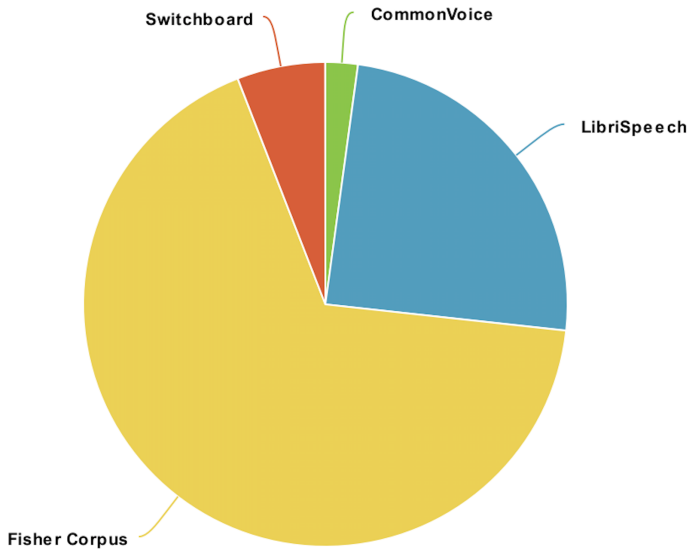

Figure 7: Proportion of corpora used for the pre-trained acoustic model of DeepSpeech

2.3.4 The language model of DeepSpeech

Transcribing sounds to individual words is often not enough to form meaningful sentences. For instance, both the word "night" and "knight" sound exactly the same, which makes it impossible for the acoustic model to determine which of the two is spoken in a particular example. The task of the language model is to solve this problem by looking at how probably one word follows the other. In most cases these are uni- or n-gram models. Unigrams only contain single words - no word sequences - and indicate how likely they are to occur in the language. Thus the word "night" could occur more often than "knight"

and would be prioritized. N-grams indicate for any number of consecutive words how likely these sequences are to occur. Thus, the Language model provides more context and understanding of a language for the result of the acoustic model, which without a language model would only output word sequences without context and understanding. An example would be the sentence “The night is dark”. Without context, the acoustic model would not know whether it is “night” or “knight”, but the language model estimates that it is more likely to be “night” than “knight”.

Language model provided by Mozilla

The language model provided by Mozilla consists of the most common words and sentences from the set of sentences/word sequences of the training material. Currently the language model of DeepSpeech is about 900 MB in size and consists of two files. The first one is the `trie`, a data structure used to search for strings and identify a specific word. During the transcription of a single word, the `trie` is used to decide whether the assumption of the acoustic model is correct or whether another similar sounding letter is more likely to occur in the sequence. For example, the letter “s” is more likely than a “z” to occur in the previously recognized letter “ha”, since “has” occurs in the English language in contrast to “haz”.

The second part is the language model itself, and in the case of DeepSpeech, it is created as a binary file called `lm.binary`. It contains 5-grams, which is a N-gram of the order five, which means that at most, word sequences of length five are given a probability. If sentences from the training material are shorter than 5 words, only the possible N-grams for the length of that sentence are calculated.

Creating a custom language model

It is possible to create a custom language model. For example, if it consists of only 10 words, each output of the acoustic model will be mapped to one of these 10 words from the language model. The size of a language model is also dependent on the number of words it comprehends. For example, a language model in version 0.6.0 of DeepSpeech created with only 20 words has a size of only 3 KB and the corresponding `trie` only 2 KB. `trie` and `lm.binary` are created together in the same step, while the `trie` depends on the `lm.binary` and is created from it.

2.3.5 Overview

The following figure shows the internal process of DeepSpeech.

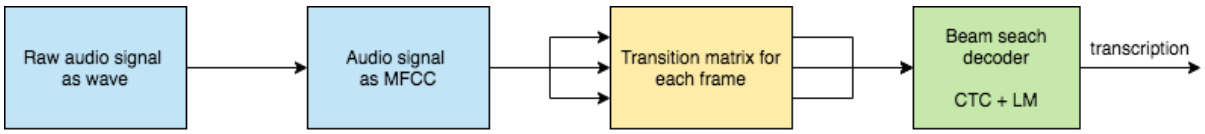

Figure 8: Internal process for inference in DeepSpeech

2.3.6 DeepSpeech in other programming languages

DeepSpeech is implemented in Python with TensorFlow. To be able to use it in other programming languages, language bindings are provided. Currently DeepSpeech supports the following environments:

- Python
- Node.js
- .NET Framework

For Node.js, packages are provided via NPM, for .NET corresponding NuGet packages. In addition, there are native clients for Linux, MacOS and Windows, which can be executed locally on the device via command-line.

3 Analysis of suitable web technologies for DeepSpeech

This chapter is aimed at working out which different web technologies can be used to integrate DeepSpeech into a modern web application. Starting from the web server, it will be examined which requirements have to be met for DeepSpeech in order to work out the necessary preceding components for the web server and client.

3.1 Language choice for the backend

DeepSpeech is available in #C in .NET Framework, Python and Node.js. Since the API of DeepSpeech is the same everywhere, one has free choice when choosing the programming language for the web server. Due to the current relevance and simplicity Node.js is a good choice, because it also shares the same language with the frontend and Node.js is known for its simplicity.

3.2 Requirements for DeepSpeech

The acoustic model of DeepSpeech is trained with files in 16kHz 16 bit mono `.wav` file format [20]. In practice, this means that the model can only transcribe data which is provided in exactly this format. At the moment this unfortunately is a weakness of neural networks the way they are currently designed. The API of DeepSpeech offers two ways to transcribe audio: Fixed length audio files or from an audio stream. Even if both use different functions of the API in the implementation, the correct format and input type must be given at both places which is a buffer[8] that contains the audio data.

3.3 Choice of framework and browser API's for the frontend

The goal of this chapter is to find out which possibilities there are to obtain the audio data in the frontend. In concrete terms, two requirements for the frontend are derived from the possibilities with DeepSpeech:

1. Creating a continuous audio stream.
2. Recording a audio file which already is in the right format for DeepSpeech.

[8] Part of a stream which is currently loaded for reading/writing

To meet these requirements, frameworks, browser API's and Javascript libraries are examined for their possibilities.

3.3.1 Choice of the framework

Nowadays, hardly any web applications are created without a framework. The question is whether one of the frameworks is more suitable than another for integrating DeepSpeech. This question can easily be answered with a "no", because none of the big frameworks like Angular, React or Vue has special functionality that makes the handling with audio easier. The chapter is nevertheless included in this thesis to answer this question. The frameworks only provide the basic architecture for a web application and use different design patterns to simplify the development of more complex applications. In contrast to libraries, they do not provide a specific implementation for handling audio streams for instance. A small advantage of Angular is the use of RxJS [9] "[...] a library for reactive programming using Observables, to make it easier to compose asynchronous or callback-based code" [23]. This makes it easier for example to handle the continuous transcriptions sent by the server when streaming continuous speech.

3.3.2 Choice of browser API's

The goal in choosing the browser API's is to find out which API's allow to receive an audio stream from the user and send it to the web server in a format like base64[10] or as a BLOB.[11]

Media Capture and Streams API

The Media Capture and Streams API - MediaStream API for short - is a browser API for accessing the user's video/audio hardware. With the user's consent, one has continuous access to a video/audio stream of the user. The format of the audio stream depends on the hardware but is often 44.100KHz 16 bit PCM and the samples themselves are stored in a Float32Array, one of the TypedArrays[12] of javascript which is very important for later processing. A few browsers in the latest versions even allow specifying

[9] Reactive Extensions Library for JavaScript
[10] base64 encoded data represens binary data in an ASCII string format
[11] A Blob object represents a file-like object of immutable, raw data
[12] A TypedArray is an array-like object in javascript containing only specific types of elements like 16 bit integer values for an Int16Array.

the sample rate at which the audio stream should be recorded. The output of the media device, e.g. the microphone, can be configured to for example use its “NoiseSupression” or “EchoCancellation” function, so that the output stream contains less noise or echoes. The API is available in all common browsers except Internet Explorer.

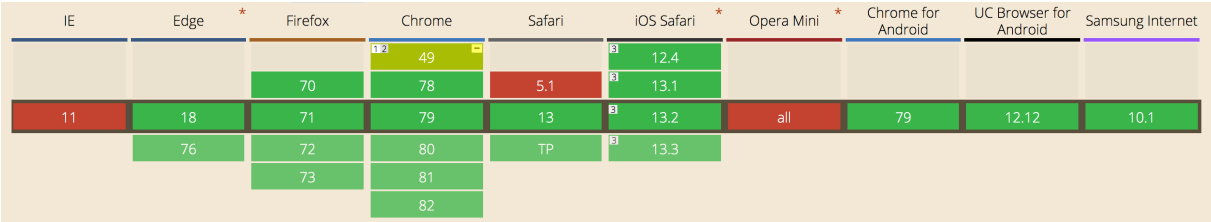

Figure 9: Browser compatibility of MediaStream API

Source: https://caniuse.com/#search=getusermedia

The MediaStream API is not to be confused with the outdated implementation of getUserMedia from the Navigator API.

MediaRecording API

The MediaRecording API allows the processing of the stream received from the MediaStream APi. It offers the possibility to collect the data and make it available as a BLOB on demand[24]. This simplifies the handling of audio data, because the output generated by the MediaRecorder can be used directly. The stream of the MediaStream API consists only of buffers in the format of Float32 arrays which contain the data of the recorded speech. Since new data is added continuously and these tiny data snippets are not suitable for further processing, the MediaRecording API allows creating a BLOB from the buffer in defined intervals of e.g. 1000ms, which results in a valid audio recording. This is convenient because this way one can wait as long as needed and send the data collectively. The MediaRecorder object sends an event containing the BLOB at the end of an interval. If no interval is specified, the event is sent as soon as the audio stream ends. This is useful for speech recognition if only one voice command is to be executed, which the user can record by holding down a button.

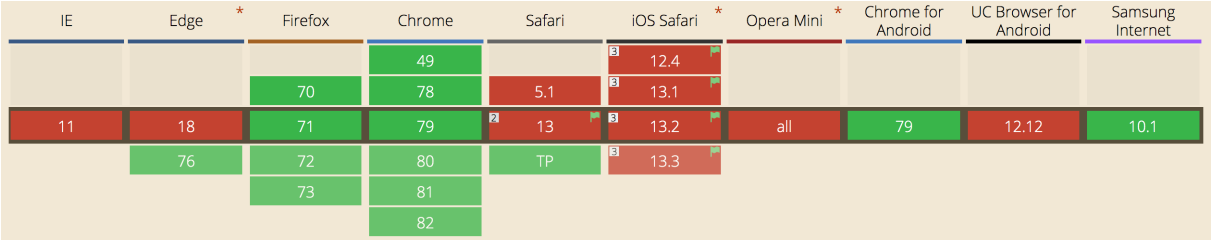

Figure 10: Browser compatibility of MediaRecorder API

Source: https://caniuse.com/#search=MediaRecorder

The availability MediaRecording API is currently still limited, but the Javascript Library “Recorder.js” provides a suitable alternative, which also works in older browser versions. Even if the use of third party libraries leads to unwanted dependencies, Recorder.js is still a good cross-browser compatible solution. A special feature of Recorder.js is that the library allows to extract data directly in `wav` format. Using both the MediaRecording API and the library Recorder.js requires further audiotranscoding in the backend, since DeepSpeech needs the audio data with a sample rate of 16KHz [25].

Web Audio API - Client-side resampling

Another approach eliminates the need to use the MediaRecording API and resamples the audio stream already in the frontend, which has decisive advantages. Resampling is the process of rewriting the audio stream from for example 44.100 Hz to 16.000 Hz, which is the format required by DeepSpeech. Since one second only contains 16.000 samples instead of 44.100, the amount of data to be transferred to the web server is reduced. It also reduces the load on the web server enormously when many users send unformatted data to the server simultaneously. If DeepSpeech will be available in later versions in the browser on the client side, the Web Audio API is also the only way to encode the audio data correctly, as the server side components for encoding the data are no longer present. Note that the audio source cannot be recorded with less than 16KHz, since resampling is only possible downwards.

For resampling the data must be processed in single sequences by a `ScriptProcessorNode` or `AudioWorkletNode`. They process the data chunk by chunk and output the resampled data into a new stream. This approach requires a custom implementation of a component, which maps the binary data chunks to a sample rate of 16KHz. The `ScriptProcessorNode` and `AudioWorkletNode` are

two interfaces of the Web Audio API, the latter replacing the former.
Although ScriptProcessorNode has been marked as obsolete, the replacement of `AudioWorkletNode` in the W3C standard is still in the “Editor’s Draft”, a design phase of W3C and therefore not yet recommended over the interface `ScriptProcessorNode`.

In order for this solution to output the resampled data collected in intervals, equivalent to the MediaRecorder API, further steps are necessary, which are explained in the practical part of this work in the chapter 4.2.3.

3.4 Transmission of audio from client to server

From the previous chapters, the following requirements for the transfer of audio data from client to server can be concluded:

1. Transfer of a single recording
2. Continuous streaming of audio data

This chapter will work out which communication protocols and methods are best suited for this purpose.

3.4.1 WebRTC

WebRTC (Web Real-time Communication) provides communication protocols and programming interfaces for P2P (Peer-to-Peer) communication [26]. This allows two devices to communicate with each other without a web server/third party after the connection has been established. A web server is required once to establish the connection. A classic use case for WebRTC are telephone calls or video chats between two or more users. Even though WebRTC provides interfaces for audio and video transmission, it is not suitable for the use of DeepSpeech, because a communication between web server and client is required. There are also server-side implementations where the server itself behaves like a client, but a traditional client-server architecture makes more sense here, because different clients want to communicate independently with the web server, not with each other, and want to use the function of the web server - the transcription of audio.

3.4.2 XHR

XMLHttpRequest (XHR) is a browser interface for transferring data via HTTP[13]. If a connection to the Web server is open, data can be transferred or requested. Virtually all modern HTTP wrappers such as axios, the FetchAPI or the HTTPClientModule from Angular use XHR internally and try to simplify handling of http requests. Each of these wrappers implements the interface somewhat differently, which is why the Angular HttpClientModule, for example, can only transfer raw data as an ArrayBuffer and not directly as an Int16Array or BLOB, although the underlying browser-native XHR interface allows this.

If the raw data is already available in its entirety, i.e. when the recording is fully captured, it makes sense to transfer it via HTTP. It is irrelevant whether it is actually a BLOB or the same data in raw format inside a Int16Array. Depending on the length of this recording, the individual parts of the BLOB/Int16Array are received asynchronously as buffers on the web server and can be further processed there.

3.4.3 WebSocket

WebSockets provide a simple TCP-based transmission method that establishes a bi-directional connection between the web browser and the WebSocket server. The origin and type of this data is not important. A Buffer or BLOB can also be transmitted without any problems. This makes it perfect for continuous streaming of audio data to the web server in one direction, and the web server's response with the transcribed text in the other direction.

Libraries like Socket.IO simplify the development with the underlying technology of WebSockets and build on it with additional functionality. For the simple use case of one client and one server the standard implementation of WebSockets are sufficient. When streaming continuously from the browser, the MediaRecording API delivers a BLOB with the data collected up to that point every interval. This BLOB can then be transmitted to the WebSocket server, which can write the data to the server-side stream for further processing

[13] HTTP functions as a request–response protocol for a client–server architecture

3.5 Server side stream handling

Once the audio data is received by the web server - whether as a BLOB or stream - it must be converted in several steps to a format that is readable by DeepSpeech.

To be able to process the audio data further, it must be modified via a stream pipeline. This is also the case when the audio is a finished recording, as large amounts of data should always be processed through a stream of individual buffers.

As described in previous chapters, there are two different use cases for this:

1. pipeline for continuous streaming
2. pipeline for converting an audio recording

Streams in Node.js have one input and one output. The input or the source of the stream pushes the data as a buffer into the stream and the modified output can be fed into the input of the next stream. The source for audio streaming would be the WebSocket-Server which pushes the binary data from the audio stream of the frontend into the stream of the backend in intervals. In Node.js there are different stream types like Readable, Writable or Duplex streams. For the WebSocket-server a readable stream is suitable, because buffers arriving via the Websocket-connection have to be pushed manually into the stream and the stream has to be readable by another stream. The naming “readable” means that the stream has to be able to be read out via a pipe from another (writable) stream. A pipe is structured as follows: `source.pipe(destination)`, whereas source must be readable and destination writable. An example of a duplex stream would be `source.pipe(duplex).pipe(destination)`, since this is both source and destination.
It is not to be misunderstood that a readable stream must still be manually pushed to, since it also needs a buffer that the destination can read from. It is just not able to be the “destination” of a pipeline.

This chapter will describe how such pipelines have to be built to be able to deliver the right format for DeepSpeech.

3.5.1 FFMPEG

After the server-side stream was created via the WebSocket server and data is continuously loaded into the stream via WebSockets, the same stream that was created in the browser now exists on the web

server.

Since the format of the stream differs depending on the user's browser and hardware/operating system, the first step in the pipeline is to rewrite the stream to a uniform format readable by DeepSpeech. The only format readable by DeepSpeech is `16le PCM` in raw format or in a `wav` file container, which is a signal with one channel (mono), 16bit bit depth and 16KHz sample rate.

A suitable software component for this is FFMPEG, a program library which can convert audio and video formats. This can be used in Node.js by starting FFMPEG as a child process on which various commands can be executed. For this purpose FFMPEG must be installed on the operating system of the web server. As described in chapter 3.3.2, other approaches exist where the stream is already converted to the appropriate format in the frontend, but the server-side approach is also a practical solution depending on the use case. FFMPEG has an interface for converting a stream as well as one for static/completed recordings.

3.5.2 VAD

One of the big challenges in continuously transcribing the audio stream is to recognize the difference between silence and speech, so that individual sentences/inputs can be recognized and the stream can be processed step by step only when speech is spoken. This is where Voice activity detection (VAD) comes in.

A robustly trained acoustic model should be able to handle silence correctly, but it makes sense to load the neural network only when speech is actually being spoken. If VAD is used in the stream pipeline, it will only pass on data containing words in its output stream. For Node-js there is the NPM package `node-vad`. Depending on the configuration of the VAD stream, the activity detection reacts more sensitive. A very important note is that `node-vad` in the current version has problems with the sensitivity, which causes a few milliseconds to be missing at the beginning and end of the speech. This causes the first word to not being recognized by DeepSpeech or being recognized incorrectly when it is short. In this pipe VAD receives the formatted stream from FFMPEG and passes the buffers (a chunk of the stream) containing speech to DeepSpeech.

VAD itself only accepts streams with a sample rate of 8, 16, 32 or 48 KHz. If this was not the case, and VAD would support any format, it would make more sense to use VAD in the pipeline before FFMPEG,

because then resampling does not have to be done for the whole continuous stream, but only for those parts where speech is spoken, which would save resources.

3.5.3 DeepSpeech streaming

In DeepSpeech there are basically two ways to obtain a transcription.

1. start inference after all of the data is available
2. stream data and already transcribe while speaking.

The former would result in waiting too long to get a response if the recording is very long. Therefore the streaming interface offers the possibility to start the transcription while the recording is still ongoing. Transcriptions can be requested while the stream is active as well as when it is finished. Even if the client recording is already available in its entirety and is sent via XHR, the data would still arrive on the web server in chunks, depending on the length of the recording, as is the case with the WebSocket connection. Therefore, it would be a possibility to use the streaming interface of DeepSpeech instead of awaiting the complete data.

3.5.4 SOX

Chapter 3.5.1 showed that FFMPEG is suitable for resampling an audio stream. However, if the frontend already is delivering the audio binary data in wav format as a BLOB to the backend, i.e. when an audio recording is sent from Recorder.js, only the sampling rate and bit depth needs to be adjusted. It would be possible to use FFMPEG for this, but it is easier to do this with the library SOX if the data is already in `wav` format and not in `audio/webm` format which would be the case if the blob was created using the MediaRecording API. Similar to FFMPEG, SOX is a program library for converting audio. But unlike FFMPEG, it does not require a child process to be started in the Node.js runtime environment. However it cannot handle data from the `audio/webm` file container.

Sox provides several Node.js packages, including `sox-stream` which can be used to convert streams. Even if the audio recording is finite and already exists as a finished recording, the audio file must be streamed again on the server side, as this simplifies the handling of the audio data and otherwise converting longer files would synchronously block the event loop of Node.js.

3.6 General summary

In previous chapters, individual components were worked out. How these work together in general is to be shown in this chapter.

The following figure shows the roundtrip for both cases - transcribing a continuous stream and an audio recording - in case the audio data is sent unformatted to the web server.

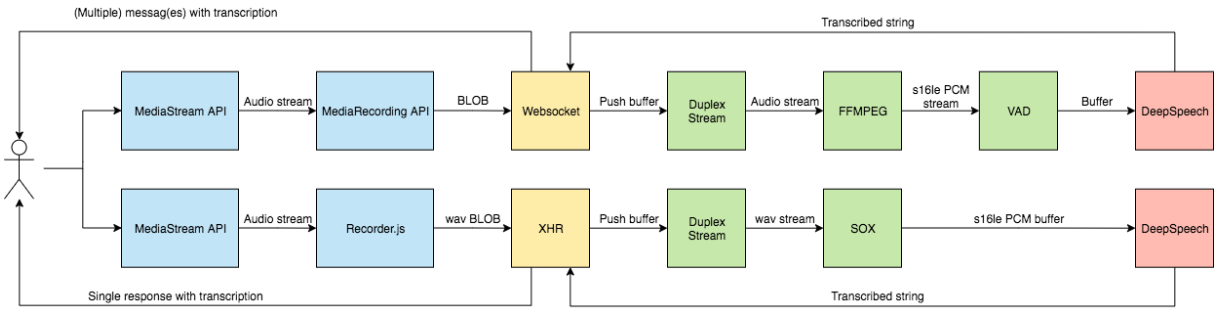

Figure 11: Web architecture roundtrip with server side resampling

To send the completed audio recording, it would also be useful to use WebSockets, but since this is actually only one “request”, XHR makes more sense, since no ongoing connection is needed after sending the recording and you would have to manually closed. If the audio data is already resampled in the frontend, the server-side components responsible for resampling are not needed.

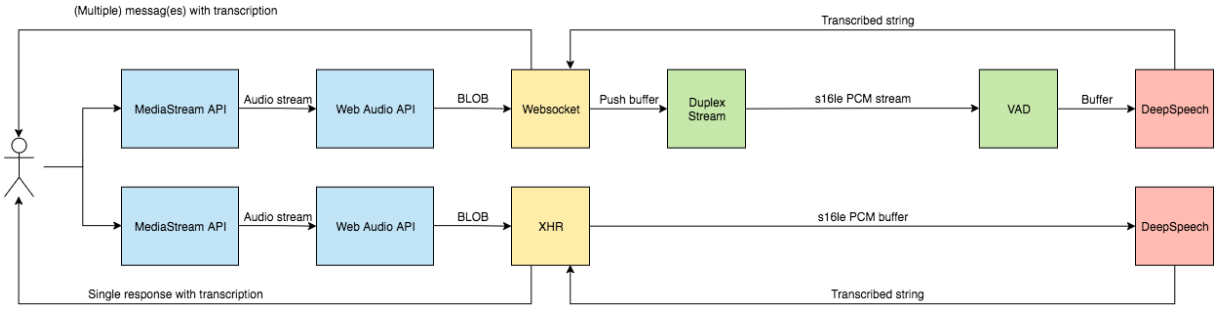

Figure 12: Web architecture roundtrip with client side resampling

If it is already possible to specify in the MediaStreams API at which sample rate a recording shall be made, neither the Web Audio API nor any server-side transcoding is needed. However, this is not a very common feature and only available in a few modern browser versions.

The decision whether to resample client- or server side depends on the circumstances. If an increased network traffic is not a problem, it is sufficient to resample the data on the web server first to reduce the processing power of the user's device.

4 Implementation

4.1 Introduction demo application

The application in question is a cocktail machine called Mocktail Mixer which was developed by Google in cooperation with the innovation studio Deeplocal as a DIY (do-it-yourself) open source project. All steps to reproduce the hard- and software are being explained in their Github Repository.

Figure 13: Design of the cocktail machine

The replica of the cocktail machine is mainly based on the hardware of the “Mocktail machine”, but for the replica NestJS[14] is used for the backend instead of Python. Since the machine is to be used at trade fairs, the use of the Google Assistant SDK is not ideal, since it requires an existing connection to the Internet to access the Google Speech API. At this point it makes sense to use DeepSpeech, because then, everything that is technically required for the functionality of the machine is provided within the machine, without any need to communicate with anthing other than it’s own locally running web server.

[14] A node.js framework which builds up on express

To start the order by voice there is a button on the side to start and end the recording by holding it. In addition to the voice control an app is being developed to create the cocktail via a user interface.

At the start of the recording a Rasberry Pi communicates with the NextJS application which, after processing the voice command, gives the necessary commands to an Arduino microcontroller to pour the right amount of the different drinks into a glass.

4.1.1 Implementation of the elaborated concepts in a demo application

Since the cocktail machine is still under construction, the concepts worked out will be implemented in a separate application. This application will allow testing the functionality of DeepSpeech beyond the use case of the cocktail machine.

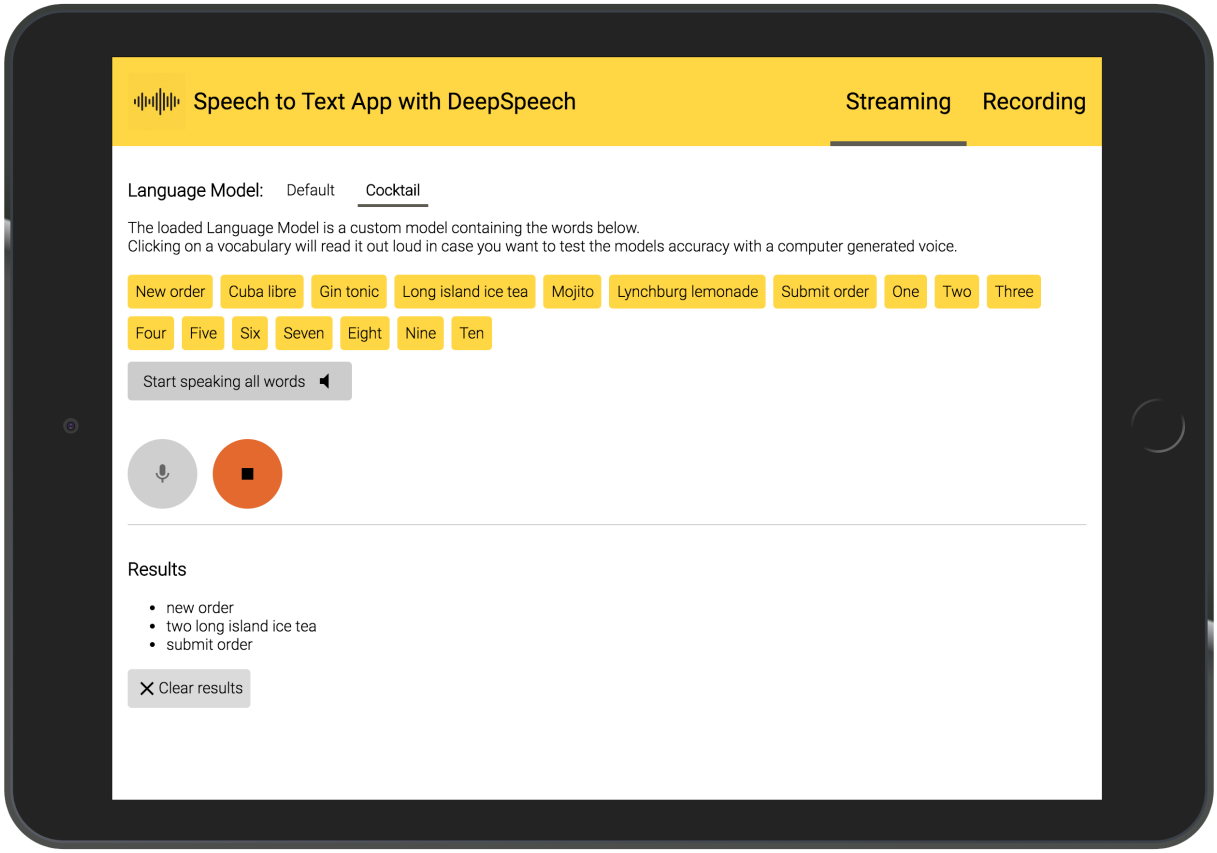

Figure 14: User interface of the demo application

The demo application supports streaming audio as well as recording and sending a recording. The

language model should also be capable of being switched between a custom model that understands various commands specifically for ordering cocktails, and the original language model of DeepSpeech that contains the entire English vocabulary. Furthermore it should be possible to output the vocabulary via speech synthesis[15] in the browser to use the output of the speaker as input for speech recognition, which simplifies manual testing with a computer generated voice.

4.2 Implementation Frontend

4.2.1 Access to the user's microphone

Via the MediaDevices interface of the MediaStream API the user's microphone can be accessed via `getUserMedia`. Since the stream is returned as promise, this can be resolved with "await". The `constraints` object can be configured arbitrarily, like in this example with `noiseSupression`, if the device supports this function.

```
let constraints = {
    audio: {
        "noiseSuppression": true
    }
};

let mediaStream = await navigator.mediaDevices.getUserMedia(constraints
    );
```

If the device doesn't support one of the constraints it's simply not being used.

The stream should be assigned a variable for later use. In this case it contains an audio track, but could also contain a video track if this was configured in the `constraints` object. The format of the audio stream is 44,100Hz, stereo and 16bit depth in most browsers.

Depending on the browser and version the stream can be output in 16KHz which is possible with the following constraint:

```
let constraints = {
    audio: {
        "sampleRate": 16000
    }
};
```

[15] Speech synthesis is the artificial production of human speech[27]

4.2.2 Obtaining binary data from the audio stream

Via the MediaRecording API the stream from the MediaStream API can now be further processed with a `MediaRecorder` object. When the MediaRecorder is started, it internally collects the audio data as long as it is running. For each `ondataavailale` event the data collected since the last event can be retrieved. If no interval in milliseconds is specified as a parameter when the MediaRecorder is started, the event is only triggered when the stream ends. If a number is defined, the MediaRecorder triggers an event every 1000ms with the data collected in the last 1000ms, as in the example below.

The data from the event is output as a BLOB.

```
let mediaRecorder = new MediaRecorder(mediaStream);
mediaRecorder.start(1000);

mediaRecorder.ondataavailable = (e) => {
    let blob = e.data;
    //Send BLOB to Webserver
}
```

For using the MediaStream API in Angular, it is important to note that there is not yet a MonkeyPatch for zone.js for the API. Zone.js assists Angulars change detection, which is responsible for rendering in the browser when properties change, by monkey-patching the browser's asynchronous API's. Monkey-patching means that a call to the API does not go directly to the API, but first to the wrapper implemented by zone.js, in whose context Angular now detects changes and can re-render if necessary. Since the MediaStream API provides an asynchronous interface with `oondataavailable` which is not patched, the event handler of the API would have to be executed manually in the zone.js context for use in Angular.

```
// this.zone ist eine injizierte Dependency des
mediaRecorder.ondataavailable = (e) => {
    this.zone.run(() => {
        // Change something that should effect the View
    }
}
```

The chapter 4.3 describes how the data is transferred to the web server at this point. If the recording is to be played back in the browser, an `ObjectURL` can be created from the entire `BLOB`, which can be included natively in HTML through the `src` attribute of the <`audio`> tag.

```
window.URL.createObjectURL(blob);
```

```
<audio [src]="url">
```

If the data should arise in intervals, these chunks of data must be collected cumulatively in a new BLOB, which is possible via the following lines of code.

```
let collectedBlob = new Blob(
    [collectedBLOB, newBlob],
    { type: "audio/webm" }
)
```

4.2.3 Client side resampling

Resampling audio frames

`AudioContext` is an interface that can be used to process audio data. It can be created via `window.AudioContext` or `window.webkitAudioContext` for Webkit-based browsers. A data source can be connected with `createMediaStreamSource` as in the following code example. `createScriptProcessor` creates a `ScriptProcessorNode`, which splits the audio data into 4096 byte buffers and provides them one by one for further processing. This 4096 byte buffer is referred to as an audio frame. The audio frames are made available in chronological order via the event `onaudioprocess` after the AudioNode is connected to the AudioStream.

```
function resampleInternal(stream){
    const audioContext = new window.AudioContext();
    const audioSource = audioContext.createMediaStreamSource(stream);
    const node = audioContext.createScriptProcessor(4096, 1, 1);
    node.onaudioprocess = (e) => {
        processAudioFrame(e.inputBuffer.getChannelData(0));
    };
    audioSource.connect(node);
    node.connect(audioContext.destination);
    const inputSampleRate = audioSource.context.sampleRate;
}
```

The function `processAudioFrame` accepts an audio frame in any format and converts it to 16KHz.

```
let inputBuffer = [];

function processAudioFrame(inputFrame) {
    for (let i = 0; i < inputFrame.length; i++) {
        inputBuffer.push((inputFrame[i]) * 32767);
    }
    const PV_SAMPLE_RATE = 16000;
    const PV_FRAME_LENGTH = 512;

    while ((inputBuffer.length * PV_SAMPLE_RATE / inputSampleRate) >
        PV_FRAME_LENGTH) {
        outputFrame = new Int16Array(PV_FRAME_LENGTH);
        let sum = 0;
        let num = 0;
        let outputIndex = 0;
        let inputIndex = 0;

        while (outputIndex < PV_FRAME_LENGTH) {
            sum = 0;
            num = 0;
            while (inputIndex < Math.min(inputBuffer.length, (
                outputIndex + 1) * inputSampleRate / PV_SAMPLE_RATE)) {
                sum += inputBuffer[inputIndex];
                num++;
                inputIndex++;
            }
            outputFrame[outputIndex] = sum / num;
            outputIndex++;
        }

        resampledFrames$.next(outputFrame);
        inputBuffer = inputBuffer.slice(inputIndex);
    }
}
```

The result is output by `resampledFrames$.next(outputFrame)`.

`resampledFrames$` is an RxJS subject[16] that outputs each resampled frame in turn.

Buffering of the resampled audio frames

Since there is now an output of AudioFrames, an implementation is needed which buffers these frames and outputs an audio buffer or BLOB either in intervals or at the end of the audio stream analog to the fuctionality to the MediaRecorder.

[16] Observables are part of RxJs and offer a way to subscribe to new values which are emitted. In contrast to a Promise an Observable can emit multiple values as long as it's not finished. A subject is special kind of Observable that can be called itself to emit a new vaue directly and can multicast to many observers

A function is needed that takes the original stream and an interval in which the function should output the resampled data as a parameter. To create a more general solution it is best to return the data in an Int16Array, from which a BLOB can be created later on if desired. The RxJs subject `resampledFrames$` which is declared globally accessible is instantiated when the `resample` function is called. `resampleInternal` is the function described in the previous chapter which outputs the resampled frames to the subject `resampledFrames$`.

Line 8 is the most important part, which can be described in text as follows:
For each resampled frame that comes in, buffer it as long as specified in `bufferTime` and return the data reduced in an Int16Array. The process is repeated until `streamEndNotifier$` gives a signal. Every interval and at the end `bufferTime$` passes all of the emitted data in an array. If no interval is specified, the value falls back to the given default. The behavior with an unspecified interval is that the data is collected endlessly until `streamEndNotifier$` ends the observable and `bufferTime` finally returns all data since the start of the recording.

```
1  let resampledFrames$: Subject<Int16Array> = new Subject<Int16Array>();
2
3  function resample(stream: AudioStream, interval = Number.
       MAX_SAFE_INTEGER ): Observable<Int16Array>{
4      resampledFrames$ = new Subject<Int16Array>();
5      streamEndNotifier$  = new Subject<void>();
6      resampleInternal(stream);
7
8      let resampledStream$ = resampledFrames$.pipe(
9          takeUntil(streamEndNotifier$),
10         bufferTime(interval),
11         map(mergeInt16Arrays),
12     );
13     return resampledStream$;
14 }
```

The utility function `mergeInt16Arrays` reduces several Int16Arrays to one. It is needed because the operator function `bufferTime` of RxJs passes the data from the source in an array. Since `resampledFrames$` emits Int16Arrays, this creates an array of Int16Arrays.

```
1 function mergeInt16Arrays(arrays: Int16Array[]): Int16Array {
2     return arrays.reduce((previous, current) => {
3         return new Int16Array([...previous, ...current]);
4     }, new Int16Array())
5 }
```

Finally, the functionality presented above can be used as follows:

```
resample(mediaStream,1000).subscribe((data: Int16Array) => {
    //send 'data' each interval to backend
});
```

Overview resampling

The following figure gives an overview of the processing chain of the audio stream.

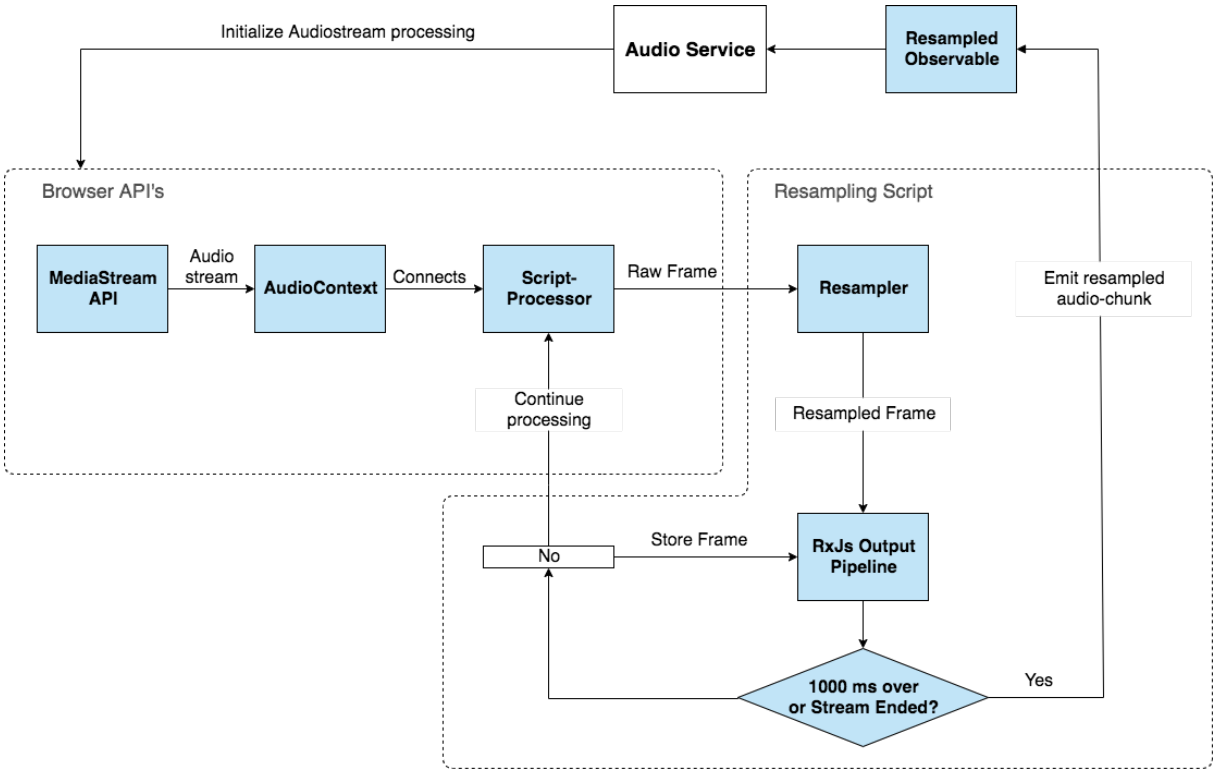

Figure 15: Overview resampling process

The `AudioService` starts the recording by requesting the audio stream from the MediaStream API. This stream is bound to a script processor via the AudioContext Interface of the Web Audio API. The script processor provides the individual frames of the audio stream one after the other for further processing by a resampling script. This resampling script converts the incoming frames to 16KHz and accumulates them internally until the time of the interval specified in the previous example (for example 1000ms) has elapsed or the stream is terminated. These resampled frames are returned as a data chunk in the Int16Array which is emitted in the Observable. The service caller receives the

Observable once at initialization and can subscribe to the emitted values.

Speech synthesis

Another feature of the application should be to output words via speech synthesis on the speaker. To make this possible the SpeechSynthesis interface of the Web SpeechApi can be used as follows.

```
speakOneWord(word:string){
    let utterance = new SpeechSynthesisUtterance(word);
    window.speechSynthesis.speak(utterance);
}
```

4.3 Client-Server communication

4.3.1 XHR

This chapter will explain the steps necessary to send audio recordings as Int16Array or BLOB to the web server and respond with their transcription.

Client

`getTextFromAudio` is a function which is passed the data and returns a promise which resolves when the transcribed text is available.

The function caller can receive the transcription as follows:

```
let transcription = await this.audioService.getTextFromAudio(this.data)
    ;
```

Internally, the function generates the HTTP request with the HttpClientModule from Angular. It is to be noted that the request body must contain the data as an `ArrayBuffer` and cannot be sent directly as an `Int16Arrray`. `data.buffer` retrieves the corresponding ArrayBuffer from an Int16Array.

```
function getTextFromAudio(data: Int16Array): Promise<string>{
    return this.http.post<string>("/api/audio",  data.buffer, {
       responseType: 'text' }).toPromise();
}
```

In comparison, this is how to create the request without a http-wrapper. Angular does the same thing, but hides the implementation with a simpler version.

```
function getTextFromAudio(data: Int16Array): Promise<string>{

    let transcriptions$ = new Subject<string>();

    let xhr = new XMLHttpRequest();
    xhr.open('POST', 'http://localhost:3000/audio', true);

    xhr.onreadystatechange = ()=> {
      if (xhr.readyState == XMLHttpRequest.DONE) {
        transcriptions$.next(xhr.responseText);
        transcriptions$.complete();
      }
    };

    xhr.send(data);
    return transcriptions$.toPromise();
  }
```

It makes sense to put the functionality into an Angular Service like in this example the `audioService`, because it is an interface to the web server and could be used in different places.

Server

The previous chapter showed how the client-side connection to the web server looks like when a POST request is sent to `/audio` with the data as a BLOB or Int16Array.

In the backend Node.js is used in conjunction with express.js. The variable `app` is already available in the following code as an HTTP server created by express.

```
app.post('/audio', (req, res) => {
    let chunks = [];
    req.on('data', (chunk) => {
        chunks.push(chunk);
    });
    req.on('end', () => {
        let buffer = Buffer.concat(chunks);
        let transcription = // get transcription for buffer from
           DeepSpeech
        res.send(transcription)
    });
});
```

When the connection is established, an array is created which the asynchronously arriving chunks are temporarily pushed to via HTTP. When the data transfer is complete, the collected chunks are loaded into a buffer, which can be sent to DeepSpeech for further processing and transcription.

One is used to receive data from REST interfaces in the request body. But this is only possible with the use of the `body-parser` middleware. This middleware accumulates the asynchronously arriving chunks into the body equivalent to the code example above. However, the body parser can only handle smaller, less complex data types such as JSON or text. For parsing files like audio data additional middleware like “multer” would be necessary. Since the above implementation works very directly and with little overhead, the use of more complex middleware is not necessary.

4.3.2 WebSockets

This chapter will present the steps necessary to stream audio recordings in real time to the server and receive the transcriptions.

Client

For the client-side implementation, only the `WebSocket` constructor needs to be used to connect to the URL of the WebSocket server. The `onmessage` event is used to receive messages from the web server. The WebSocket connection should only be held up as long as the audio recording is streamed. To avoid sending data to the web server while the connection is not established, it makes sense to put the establishment of the connection into a function which returns a promise which is resolved as soon as the connection is established.

```
let ws: WebSocket;
function websocketConnection(): Promise<void> {
    let isOpen = new Subject<void>();

    ws = new WebSocket("ws://localhost:3000");
    ws.onopen = () => {
        console.info("Connection is open");
        isOpen.complete();
    };

    ws.onmessage = (e) =>{
        console.info('Received message: ${e.data}');
        //e.data contains message
    }

    ws.onclose = () => console.info("Websocket connection is closed");

    return isOpen.toPromise();
}
```

Now, to send data to the WebSocket server the establishment of the connection can be awaited before sending it.

```
1 await websocketConnection();
2 this.ws.send(data);
```

It is therefore necessary to make the WebSocket connection accessible via a global variable. In the case of the demo application , it is waited for the connection to be established when the user starts recording before registering the media recorder's `ondataavailable` event. Once the connection is established and the `ondataavailable` event is registered, the data chunks received from the MediaRecorder can be transferred in the event handler using `ws.send()` easily.

WebSocket Server

After creating the web server, a WebSocket server is required for the WebSocket connection. It can be easily created and provides the two implementable event handlers `message` and `close`.

```
1  let app = express();
2  const server = http.createServer(app);
3
4  const websocketServer = new WebSocket.Server({ server });
5  websocketServer.on('connection', (ws) => {
6      ws.on('message', (data)=>{
7          //handle data
8      });
9
10     ws.on('close', () => {
11         //handle closed connection
12     })
13 });
```

If a data chunk sent by the frontend is received, it is directly available in the `message` event as `data` as a buffer as in this example.

4.3.3 Connection establishment

The following figure shows the timing of the connection establishment via WebSockets which is established when audio streaming is started by the user.

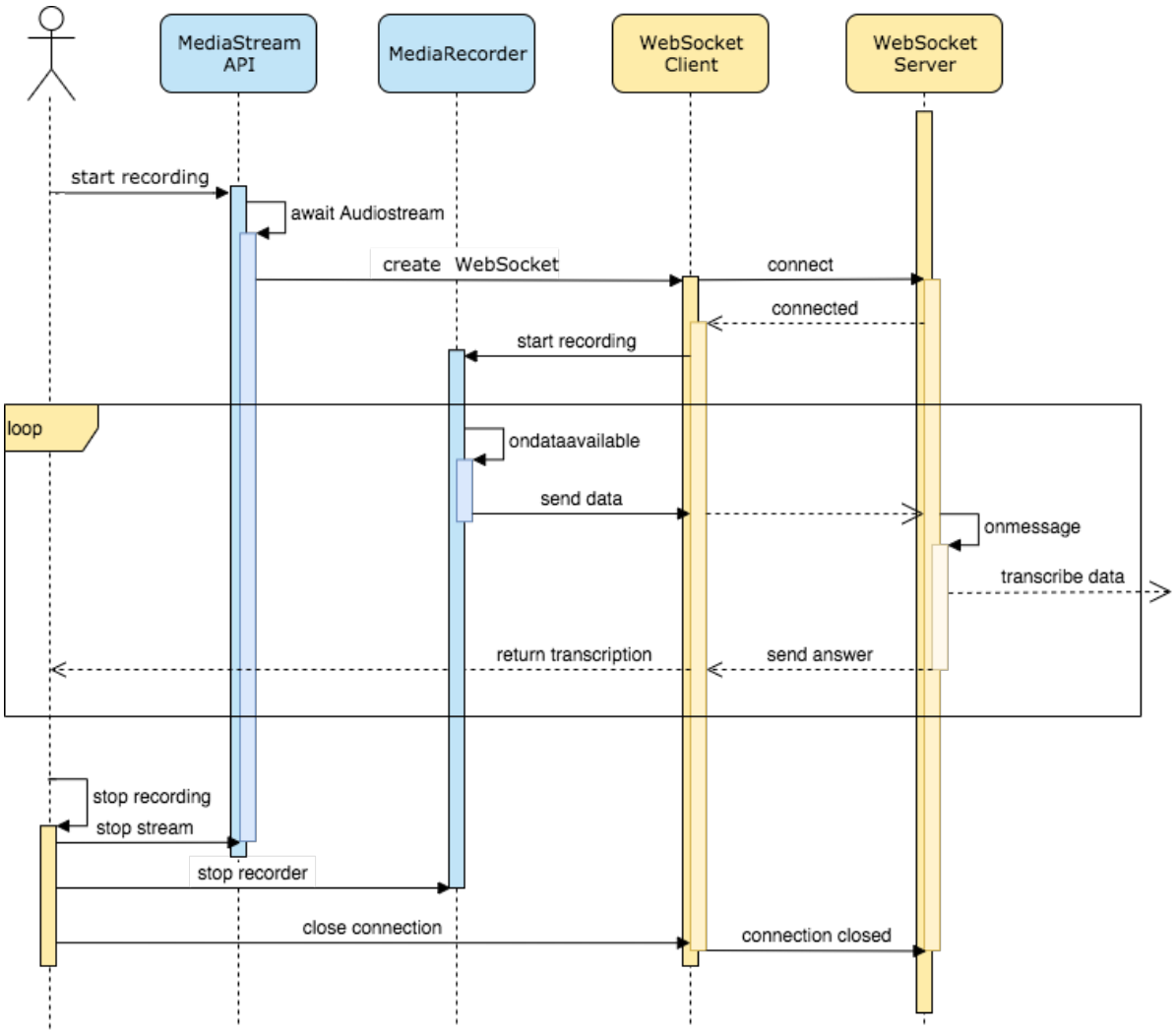

Figure 16: Timing of connection establishment for streaming with WebSockets

This approach transcribes the data only after the WebSocket connection is established. If the connection takes longer to establish, words previously recorded are ignored. The problem can be solved in the implementation by starting the media recorder in parallel to the connection. In this case it may happen that an `ondataavailable` event occurs when the connection has not yet been established. Therefore

it is necessary to check in each of these events whether the connection is established and if not, to buffer the data in the background until the connection is established. In the `onopen` event of the WebSocket the accumulated buffer can then be sent initially.

In addition, when the connection is terminated, the recording is terminated without taking into account the last interval still running. In the worst case with an interval of 1000ms a time span of 999ms is lost which is not transcribed.

The fundamental problem is that the connection may only be terminated when the last transcription has been sent. To make this possible the following steps are necessary:

- In each `ondataavailable` event, before sending the data, it must be determined if it is the last event to come, which is possible by setting a boolean flag when the recording is stopped.
- If this is the case, in addition to the audio data, the WebSocket server must be told that this is the last chunk to come.
- The server must remember this and also signal in the response with the last transcription that this is the last transcription, which is another challenge, as it can happen that within the last data chunk two separate transcriptions are created by the `VAD` component and sent one after the other.
- Only after the client has received the last transcription it can terminate the connection from its end.

For reasons of complexity, however, both optimizations will be omitted at this point.
The problem during connection establishment can be made clear to the user in the user interface by displaying the connection status.
Due to its use cases, audio-streaming is more likely to be used in such a way that it is started once, then runs for a longer period of time and is only terminated when the application is closed, for example.
For these reasons the loss is within reasonable limits.
For the simplified case of transcribing a completed recording, these problems are either not relevant as described in the chapter 4.2.3, or have been taken into account in the implementation.

For larger intervals the effect would be correspondingly worse, but it is not advisable to select a larger interval than 1000ms if the transcribed text is to be output in real time as in the specific case of the demo application. If the audio stream is only to be transcribed in the background - for example for

transcribing a speech or a telephone conference - it would also be sufficient to select an interval of 30 seconds. In this case the optimizations described above would be advisable.

4.4 Implementing the web servers

4.4.1 General setup

For the fundamental setup of the backend, a simple HTTP server with exress and a WebsocketServer is created.

With `app.use`, the necessary middleware can be used. `express.json` and `express.urlencoded` are both configurations for the middleware `body-parser` which writes the data arriving asynchronously via HTTP into the request-body.

```
const express = require('express');
const http= require('http');

let app = express();
app.use(express.json());
app.use(express.urlencoded({extended: true}));

const server = http.createServer(app);
const websocketServer = new WebSocket.Server({ server });

server.listen(3000, function () {
    console.log('Webserver listening on port 3000!');
});
```

4.4.2 Pipelining and transcoding the audio stream

Both with XHR and when sending data via WebSockets, the data is available as a buffer.

```
req.on('end', () => {
    let buffer = Buffer.concat(chunks);
});

ws.on('message', (data)=>{
    //handle data
});
```

With XHR the buffer is already given in the required format as s16le PCM, so it can be transcribed directly. For streaming over WebSockets, however, the pipeline evaluated in the chapter 3.5 must be implemented.

Creating a ReadStreams

If the buffers are coming in in 1000ms intervals from the frontend via the WebSocket connection, they can easily be pushed into a `Readable` stream. This stream is needed as the source of the pipeline.

```
1  audioStream = new Readable({
2      // The read logic is omitted since the data is pushed to the socket
3      // outside of the script's control. However, the read() function
4      // must be defined.
5      read(){}
6  });
7
8  ws.on('message', (data)=>{
9      audioStream.push(data);
10 });
```

If **`null`** is pushed into the readable stream the EOF flag (End of File) is set which causes the stream to end.

FFMPEG

After the audio stream is available on the server side, it must be resampled. An FFMPEG command can be easily executed in node in a child process with parameters.

The following parameters are the most relevant ones:

- `-i` indicates which file should be transcoded. If - is given, ffmpeg expects the data to be loaded as a stream via the interface `ffmpeg.stdin`.
- `-acodec` specifies the audio codec for the output.
- `-ar` specifies the sample rate for the output.
- `pipe:1` specifies that the transcoded data should be output as a stream on `ffmpeg.stdout`.

```
    ffmpeg = spawn('ffmpeg', [
        '-hide_banner',
        '-nostats',
        '-i', '-',
        '-vn',
        '-acodec', 'pcm_s16le',
        '-ac', 1,
        '-ar', AUDIO_SAMPLE_RATE,
        'pipe:1'
    ]);
    audioStream.pipe(ffmpeg.stdin);
```

The resampled stream is available on FFMPEG's streaming output `fmpeg.stdout`.

VAD

To use Voice Activity Detection (VAD), only the NPM package must be integrated. When generating the stream it is necessary to specify the format of the incoming data. The `mode` can be used to specify how sensitive or aggressive VAD should be to trigger activation. Different configurations are recommended for different applications:

- NORMAL: Detection mode with lowest miss-rate. Works well for most inputs.
- LOW_BITRATE: Optimized for low bitrate recordings.
- AGRESSIVE: Recommended for environments with some background noise or low quality recordings.
- VERY_AGRESSIVE: Suitable for high bitrate, low-noise data. May classify noise as voice, too.

Depending on the choice of this configuration, even quiet background noise may activate the stream.

The `debounceTime` specifies how many ms of silence must elapse before the voice input is recognized as finished.

```
const VAD = require("node-vad");

const VAD_STREAM = VAD.createStream({
    mode: VAD.Mode.Normal,
    audioFrequency: 16000,
    debounceTime: 20
});

ffmpeg.stdout.pipe(VAD_STREAM);
```

The generated `VAD_STREAM` is an extension of the Transform Stream - a special duplex stream from Node.js. Each output of the VAD stream contains a buffer with the audio data that VAD recognized as speech input.

This data can be received with `VAD_STREAM.on('data', handleData)`, whereas `handleData` is the function that receives the buffer and sends it to DeepSpeech for transcription.

4.4.3 Integrating DeepSpeech in Node.js

For using DeepSpeech in Node.js, four things are generally required:

- NPM package ‘DeepSpeech
- Acoustic model
- “trie” and “lm.binary” file for the language model

All of the required files are already provided by DeepSpeech. The chapter **??** and 4.4.5 will explain how to create and modify them independently.

```
1 const Ds = require('DeepSpeech');
2
3 const LM_WEIGHT = 0.75;
4 const BEAM_WIDTH = 500;
5 const LM_BETA = 1.85;
6
7 const model = new Ds.Model('/models/output_graph.pbmm', BEAM_WIDTH);
8 model.enableDecoderWithLM('/models/lm.binary','/models/trie', LM_WEIGHT
    , LM_BETA);
```

In the last two lines of code the model is created and linked to the Language Model. Some very technical parameters must be specified. The only parameter of interest to the developer is `LM_WEIGHT` which specifies how much the language model may affect the transcription. The value is to be understood as a percentage (0.75 = 75%). The two parameters `BEAM_WIDTH` and `LM-BETA` are necessary for the configuration of CTC and will not be explained further, as they go very deep into the implementation of the decoder and will be hidden by the API in future versions of DeepSpeech [28].

4.4.4 Transcribing the buffer with DeepSpeech

The chapter 4.4.2 explained how to prepare the audio stream for processing for DeepSpeech. This chapter shows how to transfer the resulting resampled buffer to DeepSpeech.

Basic transcription of a buffer from a finished recording

By calling `model.stt(buffer)` the transcribed string of the buffer is returned. Currently, a small workaround is necessary due to the underlying implementation of DeepSpeech, so only half of the `buffer` must be provided. The reason for this is that the Node.js Buffer API does not allow to specify the size of each element contained in the buffer and sets it to 8 bit. However, since one sample of the audio signal is 16 bit in size, one sample must be stored in 2 elements of the buffer which takes up 16 bit. When reading the buffer, DeepSpeech first determines how many elements it contains and assumes that one element corresponds to one sample. Then the number of elements in the buffer is read in 16 bit steps, starting from the memory address of the buffer's start point.

This would result in reading beyond the memory of the last element from the buffer, because when 64.000 elements which are 8 bit each are being read in 16 bit chunks (64.000 * 16 bit) there won't be any data after 32.000 * 16 bit in memory. To avoid this problem it is currently only possible to transfer half of the buffer. Thus DeepSpeech reads in only 32.000 * 16 bit, which is the same length as the original 64.000 * 8 bit buffer. This workaround only works because `Array.slice` returns an object that has a different start- and end pointer, but shares the underlying memory with the original buffer. The cause of this problem is the mapping from a Node.js buffer to a python buffer, which internally has to rely on reading in the data from the memory in order to convert it.

```
1 let result = model.stt(buffer.slice(0, buffer.length / 2));
```

Besides simply obtaining the string with the transcription, it is possible to get additional information about the transcription with `model.sttWithMetadata()`. This includes the `confidence` of the transcription which indicates how sure the acoustic model is about the result and a list of letters contained in the transcription together with their timestamp from the recording. The value of the confidence is the natural logarithm of the probability. In order to determine the actual probability value, it must be converted, whereas `l` corresponds to the logarithmic value "logit":

$$P = \frac{e^l}{1 + e^l}$$

It should be noted, however, that the transcribed text must be put together by hand, as only the letters are given individually without including the entire transcription. Also currently, the confidence for correct transcriptions can often be around just 10% because the amount of alternatives are too high.

If it would be an array of letters it could be easily merged with `Array.join()`. But since every `MetadataItem` is an object containing the letter and the corresponding timing information, the array must be reduced as follows:

```
const metadataResult = model.stt(buffer.slice(0, buffer.length / 2));

const transcription = metadataResult.items.reduce((acc, curr)=>{
    return acc + curr.character
}, "");

const probability = Math.exp(metadataResult.confidence) / (1+ Math.exp(
    metadataResult.confidence));
```

Streaming in DeepSpeech

To stream data to DeepSpeech the stream is created as follows.

```
let stream = model.createStream();
```

Then data chunks can be loaded into the stream.

```
model.feedAudioContent(stream, chunk.slice(0, chunk.length / 2));
```

At this point, DeepSpeech does not respond automatically with the transcriptions, but processes them internally. While the stream is active, a temporary transcription can be received as follows:

```
let result = model.intermediateDecode(sctx);
```

At the end of the stream the final transcription can be obtained with `let result = model.finishStream(stream)`. Even if `intermediateDecode` is not called while the stream is active, `finishStream` will return the result without delay at the end.

In addition, there is also the variant `model.finishStreamWithMetadata()` which is equivalent to `model.sttWithMetadata()`, but this does not work with `intermediateDecode`.

4.4.5 Creating a custom language model

To create a custom Language Model a program called “kenlm” is needed first. This can either be cloned from Github and built independently to get executable scripts, or be downloaded as executable binaries for the appropriate platform. Kenlm creates the file `lm.binary` which DeepSpeech needs directly from a text file `vocabulary.txt`. To get the second file `trie` DeepSpeech provides a program with the “Native_Client” which creates it from `lm.binary` and an `alphabet.txt`. The vocabulary in `vocabulary.txt` contains all words that should be recognized. It may not only contain single vocabularies but phrases as well. Each entry, whether word or phrase, goes into a separate line. Words may therefore appear several times in combination with other phrases. None of the words may contain a capital letter, as kenlm cannot handle this. Special characters like `'` are allowed.

The following directory structure is recommended for the creation the language model:

```
utils
|   generateLM.sh
|
|---training_material
|   |   alphabet.txt
|   |   vocabulary.txt
|
|---output_files
|   |   ...
|
|---kenlm
|   |   ...
|
|---native_client
|   |   ...
|
```

If all required programs and files are provided, `generateLM.sh` can be executed with the following commands

```
./kenlm/build/bin/lmplz --text training_material/vocabulary.txt --arpa
    output_files/words.arpa --order 5 --discount_fallback --temp_prefix
    /tmp/
./kenlm/build/bin/build_binary -T -s trie output_files/words.arpa
    output_files/lm.binary
./native_client.amd64.cpu.osx/generate_trie training_material/alphabet.
    txt output_files/lm.binary output_files/trie
```

After executing this script the files `words.arpa`, `lm.binary` and `trie` are located in the directory `output_files`, of which only the last two are needed by DeepSpeech.

4.5 DeepSpeech client-only

In previous chapters, comments have been made regarding client side usage. This chapter is intended to explain what would be necessary for this in the future.

On the client side, DeepSpeech can only be used as a native client on the various operating systems, for example for a Windows App. For the Web, there is currently no way to run DeepSpeech purely in the user's browser. Although the necessary models have been scaled down considerably in version 0.6.0, there is still no possibility to run DeepSpeech only in the browser. Since version 0.6.0, DeepSpeech also provides the pre-trained model compressed in .tflite format. `tflite` or "TensorFlow Lite" optimizes the models created with TensorFlow especially for mobile devices by using a special memory strategy and by excluding special and rather rarely used operations. For example, a model optimized with TensorFlow Lite has only about 40 MB instead of 200 MB in the case of DeepSpeech. A complete and comprehensive language model still is 1000 to 2000 MB in size, whereas a minimal language model with just 20 words is only about 1 KB.

Since all preceding steps such as the resampling of the stream are possible to be performed on the client-side, only the integration of DeepSpeech on the client side is missing, which is why a web server is currently still needed.

5 Conclusion

The DeepSpeech architecture presented by Baidu is a state-of-the-art e2e speech recognition system. Years of consecutive research have made it possible to solve this complex problem with a very simple architecture with few components. For a developer this means that he can integrate ASR into a web application in the simplest possible way. Only a few steps are necessary because Mozilla provides all necessary components.

Unfortunately, Mozilla's implementation DeepSpeech is still rather unsuitable for professional use or only recommended for some special cases. Due to the problems mentioned in chapter 2.3.3, DeepSpeech scores very low on background noise and accents.

However, the accuracy is greatly improved with a custom language model consisting of only a couple of words. When creating the language model, the accuracy is improved even further by adding common combinations of words. For example, for ordering any number of "gin tonic", adding the combinations "one gin tonic, two gin tonic, three gin tonic…" to the language model would be helpful because the language model then knows what other words can occur in combination with the word "gin tonic".

If one were to compare DeepSpeech with voice assistants like Alexa or Google Assistant, one would not think that there is a competitive architecture behind it. The reason for the success of Alexa or Google Assistant being the amounts of data these companies own. If the same amount of data were available open-source, DeepSpeech would not be far from the accuracy of these assistants. The key for the success of DeepSpeech is primarily CommonVoice and a continuous development and improvement of the underlying architecture.

The advantage of the solutions developed in this thesis is that only the DeepSpeech acoustic model needs to be replaced to improve the overall result. Since the API should not change too much, but only the underlying implementation, the processes presented in this thesis are reusable for future versions of DeepSpeech. To improve the accuracy, all that is needed is a `npm update deepspeech` and the replacement of the acoustic model.

It always depends on the individual case, but one of the most elegant solutions would be the combination of client-side resampling and utilizing the streaming interface of DeepSpeech. This combination minimizes the data transfer to the server and the response time of DeepSpeech.

At this point in time, there is still a lot of additional knowledge needed about DeepSpeech, as the implementation does not yet abstract some very technical details with its API. For future releases Mozilla has announced an improvement regarding this issue.

Overall, this very bleeding edge technology will be a great asset to the open-source community and for developers who need a simple, yet effective STT solution. There is yet a lot to improve but with the right time given, we could be close to having this powerful tool at hand for our own projects.

References

[1] D. Smilkov *et al.*, “TensorFlow.js: Machine learning for the web and beyond,” *CoRR*, vol. abs/1901.05350, 2019.

[2] “TeachableMachine - a fast, easy way to create machine learning models.” https://teachablemachine.withgoogle.com/.

[3] “Implementations of deepspeech.” https://github.com/search?q=deepspeech.

[4] “Mycroft open source voice assistent.” https://mycroft.ai/.

[5] D. Amodei *et al.*, “Deep speech 2: End-to-end speech recognition in english and mandarin,” *CoRR*, vol. abs/1512.02595, 2015.

[6] “DeepSpeech documentation.” https://github.com/mozilla/DeepSpeech/releases/.

[7] “DeepSpeech for maori.” http://techdeeps.com/mozilla-updates-deepspeech-with-an-english-language-model-that-runs-faster-than-real-time/.

[8] “Kaldi - an introduction.” https://medium.com/@jonathan_hui/speech-recognition-kaldi-35fec0320496.

[9] “Kaldi - official documentation.” https://kaldi-asr.org/.

[10] “Companies using kaldi.” https://discovery.hgdata.com/product/kaldi.

[11] M. Ravanelli, T. Parcollet, and Y. Bengio, “The pytorch-kaldi speech recognition toolkit.” 2018.

[12] C. Munteanu and et al., “Measuring the acceptable word error rate of machine-generated webcast transcripts.” 2006.

[13] M. Shannon, “Optimizing expected word error rate via sampling for speech recognition,” *CoRR*, vol. abs/1706.02776, 2017.

[14] D. Graff and S. Bird, “Many uses, many annotations for large speech corpora: Switchboard and TDT as case studies,” *CoRR*, vol. cs.CL/0007024, 2000.

[15] “SentenceCollector - collecting sentences for commonvoice.” https://common-voice.github.io/sentence-collector/.

[16] A. Masumori *et al.*, “Artificial neural networks on the base of biological neurons.” 2020.

[17] S. Fujita and H. Nishimura, “An evolutionary approach to associative memory in recurrent neural networks.” 1994.

[18] W.-L. Chao, H.-J. Ye, D.-C. Zhan, M. Campbell, and K. Q. Weinberger, “Revisiting meta-learning as supervised learning.” 2020.

[19] Y. He *et al.*, “Streaming end-to-end speech recognition for mobile devices,” *CoRR*, vol. abs/1811.06621, 2018.

[20] “DeepSpeech model explained - an architecture by baidu.” https://en.wikipedia.org/wiki/Pulse-code_modulation.

[21] “Pulse-code modulation.” https://deepspeech.readthedocs.io/en/v0.6.1/DeepSpeech.html.

[22] A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber, “Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks,” in *Proceedings of the 23rd international conference on machine learning*, 2006, pp. 369–376.

[23] “Reactive extensions library for javascript.” https://rxjs-dev.firebaseapp.com/.

[24] “MDN: The mediastream recording api.” https://developer.mozilla.org/en-US/docs/Web/API/MediaStream_Recording_API.

[25] A. Y. Hannun *et al.*, “Deep speech: Scaling up end-to-end speech recognition,” *CoRR*, vol. abs/1412.5567, 2014.

[26] “WebRTC - p2p communication for the web.” https://webrtc.org/.

[27] O. Karaali, G. Corrigan, and I. A. Gerson, “Speech synthesis with neural networks,” *CoRR*, vol. cs.NE/9811031, 1998.

[28] “Deepspeech decoder api change.” https://github.com/mozilla/DeepSpeech/pull/2681.