FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Developing an automatic speech recognition system based on Czech spoken language
Student:	Bc. Richard Werner
Supervisor:	Mgr. Alexander Kovalenko, Ph.D.
Study Programme:	Informatics
Study Branch:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	Until the end of summer semester 2020/21

Instructions

Description:
The task is to create an efficient speech-to-text model trained on the Czech language dataset.

For task implementation:
- Review and analyze the latest state-of-the-art approaches for automatic speech recognition (ASR), mainly speech to text by using deep neural networks, including signal preprocessing, neural network type and architecture.
- Define general and specific obstacles, constraints and recommendations in the field of sound recognition.
- Based on this review, find an appropriate solution and train the model on any Czech open-source voice dataset.
- Design a voice dataset extending system for ASR which would use the trained neural network model, use unlicensed or open voice media like audiobooks, series, movies, etc.
- Optimize and train a model using above-mentioned dataset.

The result of the thesis will be the system and the open-source dataset of Czech annotated spoken language.

References

Will be provided by the supervisor.

Ing. Karel Klouda, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 5, 2020

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Developing an automatic speech recognition system based on Czech spoken language

Bc. Richard Werner

Department of Applied Mathematics
Supervisor: Mgr. Alexander Kovalenko, Ph.D.

July 30, 2020

Acknowledgements

I want to thank my supervisor Mgr. Alexander Kovalenko, Ph.D., for immediate consultations, whenever I was in need. I also thank Ing. Marek Sušický and Profinit/OpenDataLab for allowing me to work on this exciting project.

A big thanks to my friends and family for being not only mental support in dire times during my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on July 30, 2020 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Werner, Richard. *Developing an automatic speech recognition system based on Czech spoken language.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

Tato práce se zabývá automatickým rozpoznáváním řeči (ASR) za použití rekurentních neuronových sítí (RNN). Cílem je analyzovat state-of-the-art v těchto vědních odvětvích a najít vhodný český otevřený dataset společně s RNN modelem. Dalším krokem je natrénovat vybraný model na zvoleném datasetu a najít druhý zdroj hlasových dat, ke kterému bude možné následně vytvořit anotace. Výstupem práce bude natrénovaný model, open-source dataset a systém dovolující snadné předzpracování dat a další rozšiřování datasetů.

Zvoleným datasetem jsou připravená hlasová data z Poslanecké sněmovny a použitým modelem je DeepSpeech open-source projekt. Druhým zdrojem hlasových dat jsou zbylé nahrávky z PS, dostupné z jejich webových stránek. Součástí procesu přípravy těchto dat bylo použití detektoru hlasové aktivity (VAD), jehož výstup posloužil jako reference při segmentaci audio nahrávek.

Natrénovaný model dosáhl úspěšnosti 12.66 % WER (chybovost v rámci slov) a 4.63 % CER (chybovost v rámci znaků), což byly dostatečně nízké hodnoty k vytvoření anotací nových dat. Nový dataset po předzpracování obsahoval přes 580000 hlasových nahrávek s proměnnou délkou zhruba od 1 do 70 sekund. Projekt je navržen jako Docker image s předpřipravenými nástroji ke zpracování datasetů a jejich použití k učení RNN.

Výstupem je tedy natrénovaný model rekurentní neuronové sítě, otevřený český dataset s anotacemi a připravené Docker prostředí ke zpracování dat.

Klíčová slova automatické rozpoznávání řeči, rekurentní neurové sítě, long short-term memory sítě, DeepSpeech, Tensorflow, vlnková transformace, rozklad vlnkových paketů

Abstract

This thesis deals with automatic speech recognition (ASR) using recurrent neural networks (RNN). The goal is to analyze the state-of-the-art in those fields and propose a suitable Czech open-source voice dataset and an RNN model. Next, train the model on the dataset and use to trained model to transcribe another appropriate source of speech data. The output is a trained speech-to-text model, a new open-source dataset, and a system allowing accessible data preprocessing and further extension of datasets.

The dataset of choice is the Czech Parliament meetings (CPM) transcribed recordings, and the model used is the DeepSpeech open-source project. The secondary source of speech data is the rest of the recording gathered from the CPM website. Part of the preprocessing relied on the usage of a voice activity detection (VAD) model, which was used as a reference for the audio segmentation.

The trained model achieved 12.66 % WER (Word Error Rate) and 4.63 % CER (Character Error Rate), which were sufficient values for the final dataset transcription. After preprocessing, the final dataset consisted of over 580000 speech utterances of ranging length roughly from 1 up to 70 seconds. The project is designed as a Docker image with prepared custom tools and other means to preprocess datasets and feed them to an RNN.

Therefore, the output is a trained RNN model, an open-source dataset consisting of labeled recordings, and a ready-to-use Docker image with a toolkit for data preprocessing.

Keywords automatic speech recognition, recurrent neural networks, long short-term memory networks, DeepSpeech, Tensorflow, wavelet transform, wavelet packet decomposition

Contents

List of Figures

List of Tables

Introduction

Speech recognition, which is often referred to as automatic speech recognition (ASR), is the ability of a machine to transform natural spoken language to a machine-readable format. ASR algorithms work through two types of modeling: acoustic modeling and language modeling. Acoustic modeling deals with the relationship between linguistic units of speech (e.g., phonemes) and audio signals. Language modeling is looking for patterns in sequences of words and therefore helps to distinguish between different words with the same sound. There are many uses for such systems. They are varying from self-servicing call centers and self-ordering machines to mobile devices operated by voice commands.

This thesis deals with automatic speech recognition using artificial neural networks (ANN), or commonly neural networks (NN). Inspired by nature, neural networks indeed remotely resemble those found in the animal (and human) biology. They are computational systems that learn, or more precisely are trained, to perform a wide variety of classification and regression tasks. Neural networks are taught by considering examples without implicitly having any task-specific constraints or rules. In the image classification task, a NN would be introduced to a set of manually labeled images containing the labels "car" and "motorcycle," for example.

A vital part of the NN learning process is the feature extraction approach when it comes to speech recognition. There are several ways to extract information from a sound, and they all have different properties. Some of them are better suited noisy signal; others have better information compression. It often depends on the specific problem, when it gets to this choice.

The subsequent chapter will introduce neural networks and their early beginnings, followed by modern and more complex NN architectures used these days along with a mathematical background. This chapter puts a high emphasis on recurrent neural networks (RNN), namely long short-term memory (LSTM) and gated recurrent unit (GRU) networks.

The next chapter contains an introduction to automatic speech recognition

systems, their types, and typical uses. With ASR introduced, this work then explores various feature extraction methods for speech signal, focusing on the Mel-frequency spectral coefficients (MFCC) and several wavelet approaches based on the discrete wavelet transform (DWT). Lastly, an essential part of speech recognition is the classification and performance metric. That is managed by the so-called connectionist temporal classification (CTC) loss function.

The experiment section of this work is testing the performance of the wavelet packet decomposition (WPD) approach against the MFCC method.

The final chapter of this thesis describes the realization process of the tasks, showcases the results, and discusses obstacles, improvements, and the possibility of future work. The applicational nature of this work makes the contents of the Realization chapter its essential constituent.

Chapter 1

Neural Networks

Inspiration to create artificial neural networks (ANN), commonly referred to as *neural networks* (NN), was taken from the entirely different way in which human brain processes and computes information, compared to conventional digital computers. The human brain is a highly complex, nonlinear, and parallel information-processing system with the capability to organize it is structural components, known as *neurons*, to perform several different kinds of computation (e.g., perception, pattern recognition or motor functions). [1]

In its most general form, a neural network is a machine that tries to *model* how the human brain performs a particular task or function of interest. The network is usually implemented by using electronic components or is simulated by software on a digital computer. With this in mind, let us define neural networks as follows:

> *A neural network is a massively parallel distributed processor made up of simple processing units that has a natural propensity for storing experiential knowledge and making it available for use. It resembles the brain in two respects:*
>
> 1. *Knowledge is acquired by the network from its environment through a learning process.*
> 2. *Inter-neuron connection strengths, known as synaptic weights, are used to store the acquired knowledge. [1]*

The mentioned learning process is performed by a *learning algorithm*, which is a function that modifies the synaptic weights of the network to fit it to a specific problem or objective. Apart from modifying the internal weights, some networks are capable of even modifying their topology to suit themselves to a problem better.

In the following sections, I will go through the first concepts and history of neural networks, their learning algorithms, and advanced types of NN.

1.1 Basic Concept – Rosenblatt's Perceptron

Rosenblatt's perceptron was the first algorithmically described neural network. It was invented by psychologist Rosenblatt who inspired engineers, physicists, and mathematicians alike to devote their research effort to different aspects of neural networks in the 60s and 70s of the 19th century. Moreover, the concept of perceptron in its basic form is as valid today as it was in 1958 when Rosenblatt's paper on the perceptron was first published. [1]

An artificial neural network, in general, can be viewed as an oriented graph, where nodes are neurons, and edges are the inter-neuron connections. Each neuron consists of three basic components [2]:

1. **weight vector**, which stores the weights of the connections between neurons,
2. **summation function**, which computes weighted sum of all inputs, and
3. **activation function** that maps the result of sum function to a number based on the specific type of the activation function.

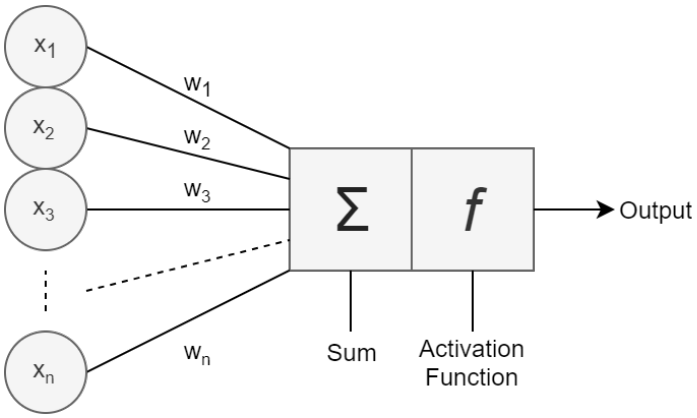


Figure 1.1: Perceptron schema

There is a general schema of a perceptron cell in figure 1.1. Each of the inputs x_i has a corresponding weight w_i. The weighted sum of the inputs a would then be computed as:

$$a = \sum_{i=0}^{n} w_i x_i + b, \tag{1.1.0.1}$$

where b is externally applied bias.

The activation function of the original perceptron was implemented as the signum function. That means the perceptron was able to classify an input vector $X = (x_1, ..., x_n)$ into one of two classes, because the range of this

function is a two-value set $\{-1, 1\}$. In the purest form of the perceptron, there are two decision regions separated by a *hyperplane*, which is defined by:

$$\sum_{i=0}^{n} w_i x_i + b = 0 \tag{1.1.0.2}$$

which would for example mean a simple line in a 2D space:

$$w_1 x_1 + x_2 x_2 + b = 0 \tag{1.1.0.3}$$

Since the basic perceptron is only able to generate a hyperplane as a decision boundary, a perceptron model is limited to linearly separable problems, thus is unable to classify any more complex tasks correctly. This leads us to a conclusion, that something more robust is needed. [1]

1.2 Multilayer Perceptron

The multilayer perceptron (MLP) consists of multiple layers of neurons that interact using weighted connections. After the first input layer, there are usually several more hidden layers, followed by the last output layer. There are no connections between neurons in one layer, while all neurons in one layer are fully connected to neurons in adjacent layers. While this statement tends to create an impression that all the information from one layer is copied to all the neurons of the adjacent layer, it is not necessarily so. Since the connection weights are usually real numbers, some of the connections may effectively be rendered insignificant. [3, 1] There is an example of a multilayer NN in figure 1.2.

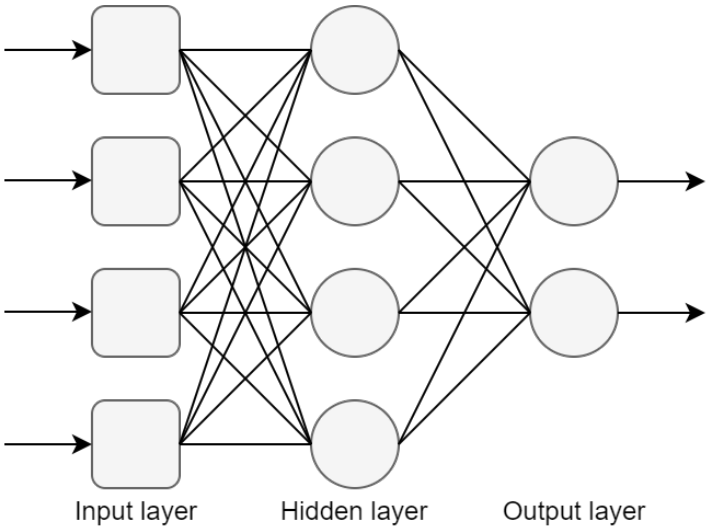


Figure 1.2: Multilayer NN schema

Another difference over the simple perceptron is the activation function. Each model of neuron includes a nonlinear activation function that is *differentiable*.

Those characteristics, however, are responsible for the deficiencies in our understanding of the behavior of the network. The distributed non-linearity and high connectivity of the network make the analysis of the network quite hard to undertake. Moreover, the use of hidden layers makes the learning process harder to visualize.

1.2.1 Backpropagation

Backpropagation, short for "backward propagation of errors", is an algorithm for supervised learning of ANNs, which is using the *gradient descend*. In this

section, especially in the formal definition, I've drawn from [4]. It proceeds in two phases:

1. In the **forward phase**, the connection weights of the network are fixed, and the input is propagated through the network, layer by layer, until it reaches the output. In other words, the network performs inference.

2. In the **backward phase**, the error is computed by comparing the network output with the desired output. The resulting error is then propagated through the network, but this time from the output layer to the input. In this phase, appropriate adjustments are made to the connection weights based on the amount of error introduced by the corresponding connection. [1]

Formal Definition

If we were to define the backpropagation formally, there would be three things required:

1. A **dataset** consisting of input-output pairs $(\vec{x_i}, \vec{y_i})$, where $\vec{x_i}$ is the input and $\vec{y_i}$ is the corresponding desired output. The dataset of size N is denoted $X = \{(\vec{x_1}, \vec{y_1}), ..., (\vec{x_N}, \vec{y_N})\}$.

2. A **feedforward NN**, e.g. our MLP, whose parameters are collectively denoted θ. Parameters of primary interest are w_{ij}^k, which is the connection weight between node j in layer l_k and node i in layer l_{k-1}, and b_i^k, the bias for node i in layer l_k.

3. An **error function** $E(X, \theta)$, which defines the error between the desired output $\vec{y_i}$ and the calculated (inferred) output $\hat{\vec{y_i}}$ of the NN on input $\vec{x_i}$ for a dataset X and parameters θ.

Training a NN with gradient descent requires the calculation of the gradient of the error function $E(X, \theta)$ with respect to the network parameters. Then, according to the learning rate α, each iteration of gradient descend updates the weights and biases θ as:

$$\theta^{t+1} = \theta^t - \alpha \frac{\partial E(X, \theta^t)}{\partial \theta}, \tag{1.2.1.1}$$

where θ^t denotes the parameters of the network at iteration t in gradient descend.

The following formulation is for a neural network with one output. The algorithm can be applied to a network with any number of outputs by consistent application of the chain rule and power rule. Thus, for all the examples below, input-output pair will be of the form $(\vec{x}, y)$, i.e. the target variable is a number, not a vector.

Preliminaries

Let us use the *mean squared error* as our error function:

$$E(X,\theta) = \frac{1}{2N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2, \tag{1.2.1.2}$$

where y_i is the target output value and $\hat{y}_i$ is the computed output value for the input $\vec{x_i}$. This function is used for numeric (not a vector) output. If we were to predict vectors, the use of a vector similarity function would be necessary.

Lets us lay down a few preliminaries:

- the sum of weighted inputs for the i-th node in the k-th layer a_i^k is:

$$a_i^k = b_i^k + \sum_{j=1}^{r_{k-1}} w_{ij}^k o_j^{k-1} = \sum_{j=0}^{r_{k-1}} w_{ij}^k o_j^{k-1} \tag{1.2.1.3}$$

  where b_i^k is bias for node i in layer k, r_k is number of nodes in layer l_k, w_{ij}^k is weight for node j in layer l_k from incoming node i and o_j^k is output for node i in layer l_k. After the standard version with the explicit bias follows the one with the bias incorporated into the weights:

$$w_{i0}^k = b_i^k. \tag{1.2.1.4}$$

- Backpropagation attempts to minimize the error function with respect to the NN weights by calculating the value of $\frac{\partial E}{\partial w_{ij}^k}$ for each weight w_{ij}^k. We can decompose the derivative with respect to each input-output pair as follows:

$$\frac{\partial E(X,\theta)}{\partial w_{ij}^k} = \frac{1}{N}\sum_{d=1}^{N}\frac{\partial}{\partial w_{ij}^k}\left(\frac{1}{2}(\hat{y}_d - y_d)^2\right) = \frac{1}{N}\sum_{d=1}^{N}\frac{\partial E_d}{\partial w_{ij}^k}. \tag{1.2.1.5}$$

Error Function Derivatives

The derivation of the backpropagation algorithm begins by applying the *chain rule* to the error function partial derivative:

$$\frac{\partial E}{\partial w_{ij}^k} = \frac{\partial E}{\partial a_j^k}\frac{\partial a_j^k}{\partial w_{ij}^k}, \tag{1.2.1.6}$$

where a_j^k is the weighted sum (activation) of node j in layer k before it is passed to the nonlinear activation function. This decomposition simply says the change in the error function due to a weight is a product of the change in

the error function E due to the sum a_j^k times the change in the sum a_j^k due to the weight w_{ij}^k.

Mostly, the whole equation is simplified by calling the first term the *error* and denoting it

$$\delta_j^k = \frac{\partial E}{\partial a_j^k}. \tag{1.2.1.7}$$

The second term is calculated from the equation (1.2.1.3) for a_i^k above:

$$\frac{\partial a_j^k}{\partial w_{ij}^k} = \frac{\partial}{\partial w_{ij}^k}\left(\sum_{l=0}^{r_{k-1}} w_{lj}^k o_l^{k-1}\right) = o_i^{k-1}. \tag{1.2.1.8}$$

And finally the partial derivative of the error function E with respect to a weight w_{ij}^k is

$$\frac{\partial E}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1}. \tag{1.2.1.9}$$

This is quite intuitive, since the weight w_{ij}^k connects the output of node i in layer $k-1$ to the input of node j in layer k in the computation graph.

The Output Layer

Backpropagation starts from the final layer, thus it attempts to define the value δ_1^m, where m is the index of the final layer and the subscript is constant 1, because we are concerned with only one-output NN. We are able to express the output value by a neuron from the weighted sum from the previous layer and the activation function of the current neuron. Therefore, the error function can be expressed as

$$E = \frac{1}{2}(\hat{y} - y)^2 = \frac{1}{2}(g_0(a_1^m - y)^2, \tag{1.2.1.10}$$

where $g_0(x)$ is the activation function for the single neuron in the output layer. Applying the partial derivative and using the chain rule gives

$$\delta_1^m = (g_0(a_1^m) - y)g_0'(a_1^m) = (\hat{y} - y)g_0'(a_1^m). \tag{1.2.1.11}$$

Putting this together with the partial derivative of the error function E with respect to a weight in the final layer w_{i1}^m gives

$$\frac{\partial E}{\partial w_{i1}^m} = \delta_1^m o_i^{m-1} = (\hat{y} - y)g_o'(a_1^m)o_i^{m-1}. \tag{1.2.1.12}$$

The Hidden Layers

Firstly, we need an equation for the error term δ_j^k for layers $1 \leq k < m$. Again, with the help from the chain rule, we get:

$$\delta_j^k = \frac{\partial E}{\partial a_j^k} = \sum_{l=1}^{r^{k+1}} \frac{\partial E}{\partial a_l^{k+1}} \frac{\partial a_l^{k+1}}{\partial a_j^k}, \tag{1.2.1.13}$$

where l ranges from 1 to r^{k+1}. l does not take the value of zero because the bias input o_0^k corresponding to w_{0j}^{k+1} is fixed, and its value is not dependent on the outputs of previous layers. Using the error term δ_l^{k+1} gives the following equation:

$$\delta_j^k = \sum_{l=1}^{r_{k+1}} \delta_l^{k+1} \frac{\partial a_l^{k+1}}{\partial a_j^k}. \tag{1.2.1.14}$$

With definition of a_l^{k+1} as

$$a_l^{k+1} = \sum_{j=1}^{r_k} w_{jl}^{k+1} g(a_j^k), \tag{1.2.1.15}$$

where $g(x)$ is the activation function for the hidden layers, we get:

$$\frac{\partial a_l^{k+1}}{\partial a_j^k} = w_{jl}^{k+1} g'(a_j^k). \tag{1.2.1.16}$$

Using this in the above equation yields a final equation for the error term δ_j^k for the hidden layers called the *backpropagation formula*:

$$\delta_j^k = \sum_{l=1}^{r_{k+1}} \delta_l^{k+1} w_{jl}^{k+1} g'(a_j^k) = g'(a_j^k) \sum_{l=1}^{r_{k+1}} w_{jl}^{k+1} \delta_l^{k+1}. \tag{1.2.1.17}$$

And finally, the partial derivative of the error function E with respect to a weight in the hidden layers w_{ij}^k for $1 \leq k < m$ is

$$\frac{\partial E}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1} = g'(a_j^k) o_i^{k-1} \sum_{l=1}^{r_{k+1}} w_{jl}^{k+1} \delta_l^{k+1}. \tag{1.2.1.18}$$

The Algorithm Itself

Assuming a suitable value for learning rate α and a random initialization of the parameters w_{ij}^k, the backpropagation algorithm proceeds in four steps:

1. **Calculate the forward phase** for each input-output pair $(\overrightarrow{x_d}, y_d)$ and store the results and store the NN output $\hat{y_d}$ and inner neuron outputs a_j^k, and o_j^k for each node j in layer k by proceeding from layer 0, the input layer, to layer m, the output layer.

2. **Calculate the backward phase** for each input-output pair and store the results $\frac{\partial E_d}{\partial w_{ij}^k}$ for each weight w_{ij}^k connecting node i in layer $k-1$ to node j in layer k by proceeding from layer m to layer 1. There are three steps to compute the partial derivatives:

   a) Evaluate the error term for the output layer δ_1^m.

b) Backpropagate the error terms to the hidden layers δ_j^k, working from the final hidden layer.

c) Evaluate the partial derivatives of the individual error E_d with respect to w_{ij}^k.

3. **Combine the individual gradients** $\frac{\partial E_d}{\partial w_{ij}^k}$ for each pair to get the total gradient $\frac{\partial E(X,\theta)}{\partial w_{ij}^k}$ for the entire set of pairs $X = (\overrightarrow{x_1}, y_1), ..., (\overrightarrow{x_N}, y_M)$ by using the above mentioned (section 1.2.1) simple average of the individual gradients.

4. **Update the weights** according to the learning rate α and the total gradient by using

$$\Delta w_{ij}^k = -\alpha \frac{\partial E(X,\theta)}{\partial w_{ij}^k}, \tag{1.2.1.19}$$

which is moving the weights in the direction of the negative gradient – gradient descend.

1.2.2 The Vanishing Gradient Problem

The problem lies in many layers using certain activation functions, a sigmoid function, for example, which squishes a large input space into a small input space between 0 and 1. That means a significant change in the input leads to a small change on the output; hence its derivative becomes small.

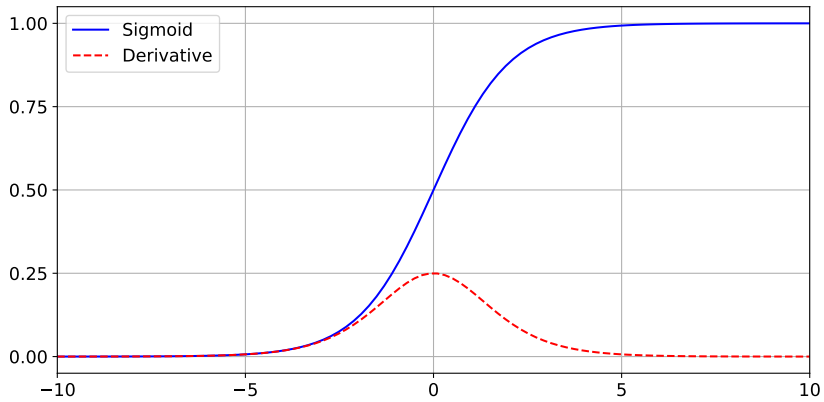


Figure 1.3: Sigmoid function and its derivative

There is an example in figure 1.3 with the sigmoid function and its derivative. When the input of the sigmoid becomes either too large or too small, its derivative comes very close to the zero. This problem does not really affect

shallow networks – networks with only a small number of layers. However, when more layers are added, it can cause the gradient to be too small to have any significant impact on the learning process.

This is caused by the chain rule used in the backpropagation, which is described for feedforward networks in section 1.2.1. It computes the gradients by moving layer by layer from the final one to the initial one. By the chain rule, each subsequent derivative is multiplied by the already computed value. Therefore, when there are n hidden layers using sigmoid-like activation function, n small derivatives are multiplied together. Thus, the gradient value decreases exponentially as the backpropagation algorithm advances to the initial layers. More on this topic and the problems mathematical background is explained in [5].

There are a few well-known solutions [6]:

- Probably the simplest solution is to use a different activation function, such as ReLU (Rectified Linear Unit), which does not cause a small derivative.
- Another solution is *residual networks* (ResNets). They provide residual connections (without weight parameters) straight to the next layers, effectively skipping the activation functions. That results in overall higher derivatives, and therefore the ability to train much deeper networks. [7]
- The last one is *batch normalization* layers. As mentioned before, the problem appears when a large input is mapped to small output, causing the derivatives to disappear. The batch normalization method normalizes the input on a predefined scale, where the sigmoid derivative is not that small.

1.3 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) is a group of neural networks quite similar to conventional NNs. They implement backward connections - cycles in the computational graph, which let the network keep some information from previous steps. [1] They can work with both variable and extensive input sequence lengths, and they use the so-called *parameter sharing*. Parameter sharing allows the RNN model to generalize across the inputs with different lengths (or generally forms) and use the same parameters (weights) across multiple time-steps. If we were to use a standard NN to read sentences and try to predict the next word or syllable, the model would need to learn all the word combinations possible for a sentence. An RNN model makes such a prediction much easier for itself.

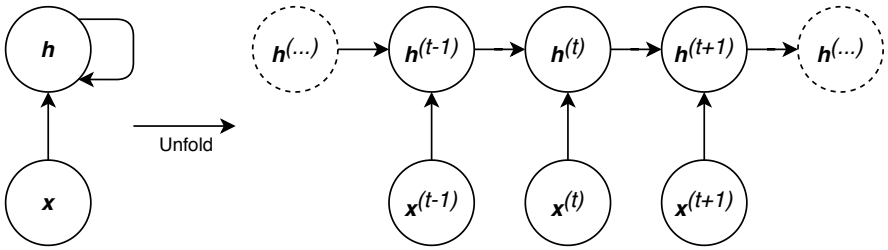


Figure 1.4: RNN unfolding

There are two ways to visualize the backward connections: [8]:

1. The more straightforward one is to illustrate the dependency as a circuit on one of the RNN component - neuron. This way, it operates in real-time, and the current state of the neuron influences its future states. The left side of figure 1.4 illustrates this approach.

2. The other way is to *unfold* the diagram into time-steps as on the right side of figure 1.4. It has separate variables for each time-step, yet each section (time-step) represents the same structure with different variables. The size of the unfolded graph depends on the size of the input sequence length.

From the figure and the description, it is evident that we need to feed the model with sequential data. That can be a next-word prediction based on an unfinished sentence, speech recognition with human speech wave signal as input, or weather prediction based on sequential environmental measurements—generally, any time-step based data.

Update equations for a standard RNN (illustrated on figure 1.4) can be defined as [8]:

$$\boldsymbol{a}^{(t)} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)}, \tag{1.3.0.1}$$

$$\boldsymbol{h}^{(t)} = h(\boldsymbol{a}^{(t)}), \tag{1.3.0.2}$$

$$\boldsymbol{o}^{(t)} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)}, \tag{1.3.0.3}$$

$$\hat{\boldsymbol{y}}^{(t)} = g(\boldsymbol{o}^{(t)}), \tag{1.3.0.4}$$

where $\boldsymbol{b}$ and $\boldsymbol{c}$ are the bias vectors, and $\boldsymbol{U}/\boldsymbol{V}/\boldsymbol{W}$ are the weight matrices for input-to-hidden/hidden-to-output/hidden-to-hidden connections, respectively. Activation function h is usually a hyperbolic tangent, g is a sigmoid and vectors $\boldsymbol{h}$ and $\hat{\boldsymbol{y}}$ are the output vectors.

RNNs, similarly as the standard deep neural networks, suffer from the vanishing and exploding gradient problems. The unfolding of neurons causes the same effect as the high amount of hidden layers in deep NNs. The solution to these problems is a more complex structure of recurrent neurons, namely, LSTM or GRU cells.

1.3.1 Long Short-Term Memory Networks

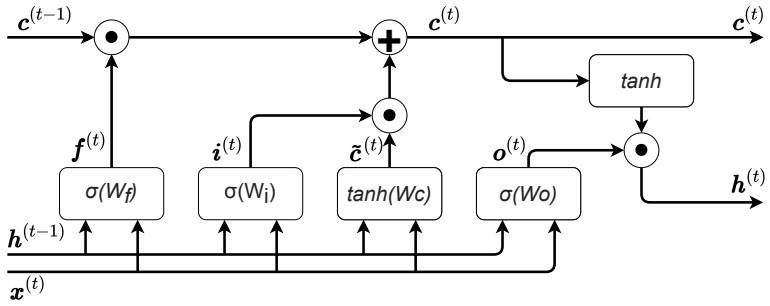


Figure 1.5: An LSTM cell. The dot in a circle sign means a component-wise product, while the plus sign is standard vector addition.

Long short-term memory networks (LSTMs) is an extension of standard RNNs where one LSTM neuron acts like several NN layers. These types of cells add the so-called gates. These gates manage access to the neural cell's internal state, and they help eliminate the vanishing/exploding gradient problem. [9]

Figure 1.5 illustrates an LSTM cell. There are two hidden states in the unit – a standard cellular state $\boldsymbol{h}^{(t)}$ and a cellular state $\boldsymbol{c}^{(t)}$, which helps to maintain long-term dependencies. The first of the gates is the forget gate $\boldsymbol{f}^t$. Its activation function, a sigmoid with values in $(0, 1)$, helps to decide which information from the cellular state $\boldsymbol{c}^{(t-1)}$ is to be kept as [10]:

$$\boldsymbol{f}^{(t)} = \sigma\left(W_f\left[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}\right] + \boldsymbol{b}_f\right), \tag{1.3.1.1}$$

where W is the weight matrix for the forget gate. The input gate $\boldsymbol{i}^{(t)}$ manages the addition of new information to the state $\boldsymbol{c}^{(t-1)}$:

$$\boldsymbol{i}^{(t)} = \sigma\left(W_i\left[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}\right] + \boldsymbol{b}_i\right). \tag{1.3.1.2}$$

The third gate, the output gate, takes care of which part of the state $\boldsymbol{c}^{(t)}$ leaves the cell as the hidden state output $\boldsymbol{h}^{(t)}$:

$$\boldsymbol{o}^{(t)} = \sigma\left(W_o\left[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}\right] + \boldsymbol{b}_o\right). \tag{1.3.1.3}$$

The current internal state $\boldsymbol{c}^{(t)}$ is computed by calculating the component-wise product of the result of the input gate and the cellular state candidate $\tilde{\boldsymbol{c}}^{(t)}$:

$$\boldsymbol{c}^{(t)} = \boldsymbol{f}^{(t)} \odot \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \odot \tilde{\boldsymbol{c}}^{(t)}, \tag{1.3.1.4}$$

where

$$\tilde{\boldsymbol{c}}^{(t)} = \tanh\left(W_c\left[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}\right] + \boldsymbol{b}_c\right). \tag{1.3.1.5}$$

The output $\boldsymbol{h}^{(t)}$ is then computed as

$$\boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)} \odot \tanh(\boldsymbol{c}^{(t)}). \tag{1.3.1.6}$$

1.3.2 Gated Recurrent Units

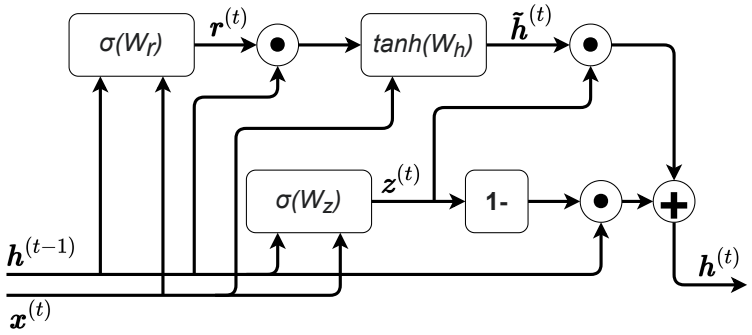


Figure 1.6: A GRU cell. The dot in a circle sign means a component-wise product, while the plus sign is standard vector addition.

Gated recurrent unit (GRU) is a simplification of LSTM-like units. It contains only two gates [10]:

- an **update gate** z_t, whose role is similar to the LSTM forget gate, and
- a **reset gate** r_t , whose role is somewhat similar to the input gate.

Other than removing the output gate from LSTM, GRU also removed the cellular state C_t. The whole cell structure is shown in figure 1.6. To represent the data flow through the GRU unit mathematically, we will start with the update gate:

$$\boldsymbol{z}^{(t)} = \sigma\left(W_z\left[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}\right] + b_z\right). \tag{1.3.2.1}$$

The reset gate is quite similar:

$$\boldsymbol{r}^{(t)} = \sigma\left(W_r\left[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}\right] + b_r\right). \tag{1.3.2.2}$$

As for the computation of the preliminary output $\tilde{h_t}$:

$$\tilde{\boldsymbol{h}}^{(t)} = tanh\left(W_h\left[\boldsymbol{r}^{(t)} \odot \boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}\right] + b_h\right). \tag{1.3.2.3}$$

Finally, the output:

$$\boldsymbol{h}^{(t)} = \left(1 - \boldsymbol{z}^{(t)}\right) \odot \boldsymbol{h}^{(t-1)} + \boldsymbol{z}^{(t)} \odot \tilde{\boldsymbol{h}}^{(t)}. \tag{1.3.2.4}$$

1.3.3 LSTM vs. GRU Performance

Experiment results of LSTM and GRU comparisons are usually inconclusive - the performances of both types of RNNs are mostly very similar, and the better choice is up to the specific task and its dataset.

For example, experiments in [11] were unable to conclusively declare the better recurrent unit when it comes to speech signal modeling.

Publication [12] tested several variations of LSTMs networks and a GRU network on several tasks (XML modeling, arithmetic modeling and filtering, word-level language modeling, and music) where GRU models slightly outperformed all other models on all tasks except for the language modeling. Nevertheless, the results were not strong enough to make a solid conclusion.

Paper [13] studies only various LSTM modifications, of which none of them has shown any significant performance improvements. On the other hand, some of the LSTM architecture simplifications did not necessarily worsen the prediction performance while (slightly) reducing the computational complexity.

There are other studies of the two network type comparisons [14, 15] which conclude either the LSTM or the GRU architectures being better the other (respectively to the citation order). However, the results are, again, quite similar as in the publications above. It always depends on the specific task, and the data the experiments are going to be conducted upon.

Chapter 2

Automatic Speech Recognition

Automatic speech recognition (ASR) has been undergoing active research for more than fifty years. It is an essential milestone in both human-human and human-machine communication. Due to the insufficient performance of technologies in the past, ASR has not become a desirable part of human-machine communication. It was because the lack of computing power did not allow passing the usability bar for real users, and other means of communication, such as keyboard and mouse, significantly outperformed speech in most aspects of communication efficiency with computers. [16]

This all changed in recent years. Speech technologies started to change how we live and work and, for some devices, became the primary means of interaction with them. By [16], there are several key areas of which progress allowed this trend:

- the first area is *Moore's law*. The law states that approximately every two years, the number of transistors in a dense integrated circuit doubles roughly every two years. [17] That leads to the computational capabilities of CPU/GPU clusters also being doubled every two years. That makes training of more complex and powerful models possible, and thus the error rates of ASR systems lower.

- The second area is the access to more data due to the continued advance of the Internet and cloud computing. By using and training models on big data collections, it is possible to build much more robust and assumption-less models than before.

- The third is that mobile, wearable, and intelligent living room devices and in-vehicle infotainment [18] became quite popular. Since the use of alternative interaction means, such as keyboard and mouse, is mostly not possible in these cases, speech communication, which is natural for humans, becomes more convenient.

There are several approaches to the ASR with different models, such as Gaussian mixture models or hidden Markov models. Since this thesis deals with neural networks, we are going to go through deep neural network (DNN) models in ASR.

Aside from diverse models, there are several ways to preprocess the raw audio sound. In the following sections, I am going to introduce some of the methods of preprocessing.

2.1 Introduction

Both in human and electronic communication, the speech information is encoded in the form of a continuously varying (analog) waveform that can be transmitted, recorded, manipulated, and ultimately decoded by a human listener. The primary analog form of the message is an acoustic waveform, which we call the *speech signal*. Those can be converted to an electrical waveform by a microphone, further manipulated by analog and digital signal processing, and then converted back to acoustic form by a loudspeaker or another electronic device.

Before we can apply any (digital) processing techniques, we must convert to the acoustic waveform, analog signal, to a sequence of numbers – digital signal. System or tool able to do such converting is called A-to-D converter, which creates digital representation by sampling the waveform in a very high rate, applies filters to preserve a prescribed bandwidth, and then reduces the sampling rate to the desired sampling rate. This discrete-time representation is the starting point for most digital signal processing applications.

Several areas that work with the digital representation of the speech signal follow. [19]

Speech coding

Perhaps the most widespread applications of digital speech are the A-to-D and D-to-A converting mentioned briefly above. It is commonly referred to as *speech coding* or *speech compression*, and its goal is to compress the digital waveform representation of speech into a lower bit-rate representation. The D-to-A decoder part of the system is often called the *synthesizer* because it must reconstruct the speech waveform from the discrete digital data.

Text-to-Speech

A Text-to-Speech analysis is another area that uses the digital signal. The goal of these systems is to start with text and automatically produce speech. The input is an ordinary text such as a newspaper article or an e-mail message. The system is depicted in figure 2.1. The first block named *Linguistic Rules*

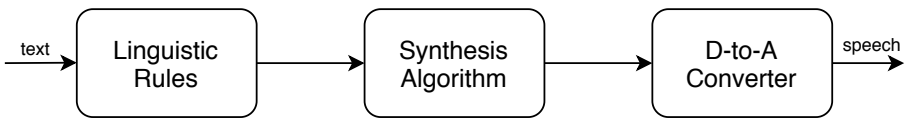


Figure 2.1: Text-to-Speech synthesis system inspired by [19]

has the job of converting the ordinary text into a set of sounds, which must be synthesized afterward. The system should include a set of linguistic rules that must determine appropriate sounds such as emphasis, pauses, or rates

of speaking. Moreover, pronounce acronyms and ambiguous words like *read, bass, or object*, how to pronounce abbreviations like *St.* (street or Saint?), *Dr.* (Doctor or drive?), and adequately pronounce specialized terms, and names. After the system builds the pronunciation set of sounds, it is time to synthesize the speech – to create the appropriate sound sequence to represent the text message in the form of speech.

Text-to-Speech synthesis systems are used to do things like to provide voice output from GPS systems, handle call center help desks, or providing information from handheld devices such as foreign language phrasebooks and dictionaries. They are also being used in announcement machines that provide information – stock quotes and airline schedules. Finally, perhaps the most important application is in reading machines for the blind, where an optical character recognition system provides the text input.

Speech Recognition

Quite the opposite of the Text-to-Speech problem described above is the *speech recognition* or *Speech-to-Text* problem. It is concerned with the extraction of information from the speech signal. Figure 2.2 shows a diagram of a generic ap-

Figure 2.2: Generic pattern matching system inspired by [19]

proach to pattern matching problems in ASR. There are several sub-problems in this class, such as

- **speech recognition**, where the goal is to extract the message from the speech signal,
- **speaker recognition**, where the goal is to identify the speaker,
- **speaker verification**, which verifies a speakers claimed identity from analysis of their speech signal,
- **word spotting**, which involves monitoring a speech signal for the occurrence of a specified word or phrase, and
- **automatic indexing** of speech recordings based on the recognition of spoken keywords.

The first block in the pattern matching system diagram 2.2 converts the analog speech waveform to the digital signal using an A-to-D converter. The *feature analysis* module, the second block, converts the sampled speech signal to a

set of feature vectors. The third and final block in the system called *pattern matching* block dynamically time aligns the set of feature vectors representing the speech signal with a concatenated set of stored patterns. Then it chooses the identity associated with the pattern, which is the closest match to the time-aligned set of feature vectors of the speech signal. The symbolic output consists of a set of recognized words in the case of speech recognition. In the case of speaker recognition, it would be the identity of the best matching talker.

Probably the most common use of speech recognition is in portable communication devices. Spoken name speech recognition in cell phones enables voice dialing, and today basically every smart phone has its own voice assistant that is capable of multitude functions including speech-to-text when writing messages, the device options management, or information retrieval from the internet. [20]

The long-time dream of speech researchers are *automatic language translation* systems. The goal of such systems is to convert spoken words in one language to spoken words in another language to facilitate natural language voice dialogues between people speaking different languages.

2.2 Feature Extraction

In this section, we are going to introduce two popular methods of speech signal feature extraction – namely, the Mel Frequency Cepstral Coefficients (MFCC) and the Wavelet approach.

The MFCC approach is computing the short-term *power spectrum*, which is based on an *inverse Fourier transform* of a log *power spectrum* on a nonlinear *mel scale* of frequency (first formulated in [21], well explained, for example, in [19]). There are several definitions of the cepstrum, and they, sometimes, deviate from the steps used in this thesis.

The wavelet approach uses the *wavelet transform* in the process of feature extraction. This transform does not expect a stationary signal and is well localized both in time and frequency using a *multiresolution* approach. The following section introduces both Fourier and wavelet transforms, their differences, and their advantages.

2.2.1 Wavelet vs. Fourier Transform

Let us start by stating that the Fourier transform can be viewed as a particular case of wavelet transforms. They are both expanding the input signal onto a set of basis functions and transforms that will give an efficient and informative description of the signal.

We are going to start by defining generic transforms and pinpointing the main differences between wavelet and Fourier transforms. After that, we will build an intuitive idea around the transforms to better understands what is hiding behind the mathematics. This section draws from [22, 23, 24].

A generic discrete linear decomposition of a signal $f(t)$ can be expressed as:

$$f(t) = \sum_{l} a_l \psi_l(t), \tag{2.2.1.1}$$

where a_l are the real-valued expansion coefficients, and $\psi_l(t)$ is a set of real-valued functions of t called the expansion set. If the expansion set is unique, it is called a *basis* for the class of functions that can be so expressed.

This is how a discrete Fourier transform looks like:

$$\hat{f}_n = \sum_{k=0}^{N-1} f_k e^{-\frac{2\pi i n k}{N}}, \tag{2.2.1.2}$$

with its inverse counterpart defined as:

$$f_k = \frac{1}{N} \sum_{n=0}^{N-1} \hat{f}_n e^{\frac{2\pi i n k}{N}}. \tag{2.2.1.3}$$

For a Fourier transform, the basis functions are sine and cosine functions – generally the complex exponential, which we can see in the definitions above.

The problem with the Fourier transform is that it expects a stationary signal in time. This presumption of a stationary signal comes from the basis this transform uses – sine and cosine. Those are periodic, infinite functions, and any linear combination of such functions creates, again, a periodic, infinite one. In other words, this transformation discards all the information about time but greatly captures the frequency information. We can say that the Fourier transform has great *frequency localization* and weak (none) *time localization*. That, however, does not go along with the properties of human speech.

Human speech is very far from being stationary. To be able to recognize words, syllables, and phonemes, we need to have both time and frequency localization. The first approach to tackle this problem is the *Gabor* transform named after Dennis Gabor.

The Gabor Transform

Gabor transform is a special case of short-time Fourier transform (STFT). STFT splits the signal into short segments of equal length and computes the Fourier transform for each segment separately. Gabor transform is additionally using a Gaussian window function to separate the desired part of the signal from the rest, squishing undesired parts of the signal (close) to zero.

MFCC approach to the feature extraction is using a very similar process with the signal splitting and windowing. The whole process is described in section 2.2.2.

As hinted above, the Gabor transform is still not ideal. Increasing the window size lowers the time resolution for better frequency information. Smaller window, on the other hand, increases the time resolution, but we lose the low-frequency content available in longer time intervals. How to avoid such trade-off and keep both frequency and time resolution as high as possible? Let us first introduce mother wavelets.

A Mother Wavelet

A *mother wavelet* is a fundamental parametric function defined as:

$$\psi_{a,b}(t) = \frac{1}{\sqrt{a}}\psi\left(\frac{t-b}{a}\right) \tag{2.2.1.4}$$

This is a generic description of a wavelet, without a specific wavelet function given. It has two real parameters:

- parameter a, which is a non-zero scaling parameter, and
- parameter b – a translation parameter.

Now, we have a function that is both scalable and translatable in time. In other words, we have got a sliding window that has a dynamic size. Now,

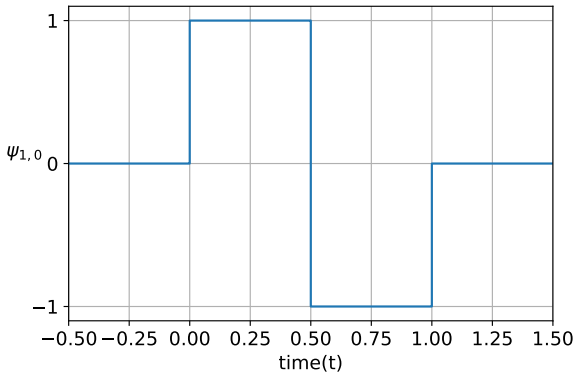


Figure 2.3: Haar wavelet

the only thing that remains is to define the wavelet function itself. There is a wide variety of wavelet functions that can be defined. Depending on the type of problem, we choose the one with the best properties for us. In the end, the wavelet is simply another expansion basis for the signal representation.

The Haar Wavelet

The first recognized wavelet function was the Haar wavelet in 1910, meaning the idea of wavelets is more than a century old. It can be described as:

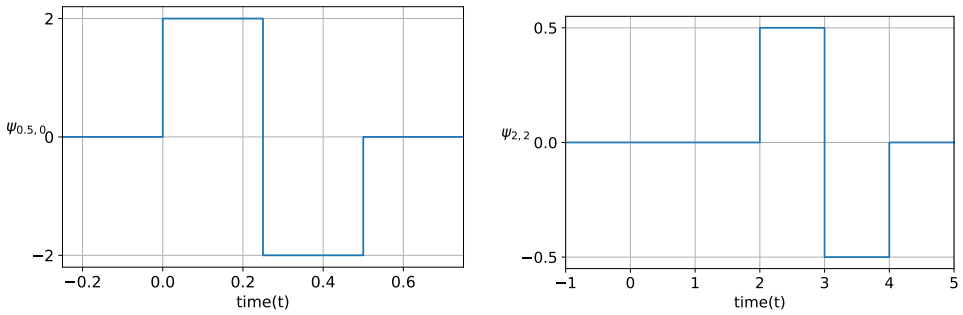
$$\psi(t) = \begin{cases} 1 & 0 \leq t < \frac{1}{2} \\ -1 & \frac{1}{2} \leq t < 1 \\ 0 & else \end{cases} \tag{2.2.1.5}$$

Figure 2.3 shows the Haar wavelet step function. It is an ideal wavelet for describing localized functions because is has *compact support*. Compact support means that a function returns values on a defined interval and returns zero outside the interval. That leads to Haar function having a strong time localization. However, its frequency localization is weak, because it decays like a sine function in powers of $\frac{1}{\omega}$ in the frequency domain (Fourier transform of the wavelet). This wavelet has two more properties, that are important for us. The sum of the "area under the curve" equals zero:

$$\int_{-\infty}^{\infty} \psi(t)dt = 0, \tag{2.2.1.6}$$

and the area of the squared absolute value of the curve equals one:

$$\int_{-\infty}^{\infty} |\psi(t)|^2 \, dt = 1. \tag{2.2.1.7}$$



(a) Compressed (b) Dilated and translated

Figure 2.4: Haar wavelet with different parameters

The Wavelet Transform

To be precise, figure 2.3, and the wavelet description (2.2.1.5) are both the wavelet $\psi_{1,0}(t)$. Meaning its translation is zero, and its scaling is unity. Figure 2.4 shows other parameter configurations for Haar wavelets. Plot 2.4a depicts wavelet $\psi_{\frac{1}{2},0}$, which is a compressed function without translation, and plot 2.4b depicts $\psi_{2,2}$, which is a dilated wavelet with positive translation.

That allows for large scale structures in time in the signal to be captured by wide time-domain Haar wavelets. On this broad scale, the time resolution is pretty bad. However, by rescaling the wavelet in time, we get a more delicate time resolution along with high-frequency information. Therefore all the information at both low and high scales is preserved. That allows a complete reconstruction of the original signal. In the end, the only limit in this process is the number of rescaling levels. That is also where the *multiresolution* term comes from. The wavelet transform captures the information on multiple time-frequency resolutions.

The wavelet basis is computed as

$$(Tf)(\omega) = \int_t K(t,\omega) f(t) dt, \tag{2.2.1.8}$$

where $K(t,\omega)$ is the kernel of the transform. The same applies to the Fourier transform, where the kernel is the oscillations given by $K(t,\omega) = exp(-i\omega t) = e^{-i\omega t}$. Now, we need to define a transform that incorporates the mother wavelet as the kernel. The following equation is the definition of the *continuous wavelet transform* (CWT):

$$\mathcal{W}_\psi[f](a,b) = (f, \psi_{a,b}) = \int_{-\infty}^{\infty} f(t) \bar{\psi}_{a,b}(t) dt. \tag{2.2.1.9}$$

CWT is a functions of the dilatation parameter a and the translation parameter b. A more simple yet equivalent formulation of the CWT by [22] is:

$$F(a,b) = \int_t f(t) \psi \left(\frac{t-a}{b} \right) dt. \tag{2.2.1.10}$$

Since the *discrete wavelet transform* is somewhat complicated to define, we will first define the inverse wavelet transform. It is even a more intuitive idea – it says how the signal is reconstructed from the set of wavelets. It goes as:

$$f(t) = \sum_k \sum_j c_{a,b}\psi_{a,b}(t), \tag{2.2.1.11}$$

where the set of expansion coefficients $a_{j,k}$ is called the *discrete wavelet transform.*

The result of a DWT are coefficients of a so-called two-band analysis tree. It is a binary tree, where only one of the branches is expanded into the next stages. Figure 2.5 depicts such a tree, which visualizes two stages of the DWT algorithm.

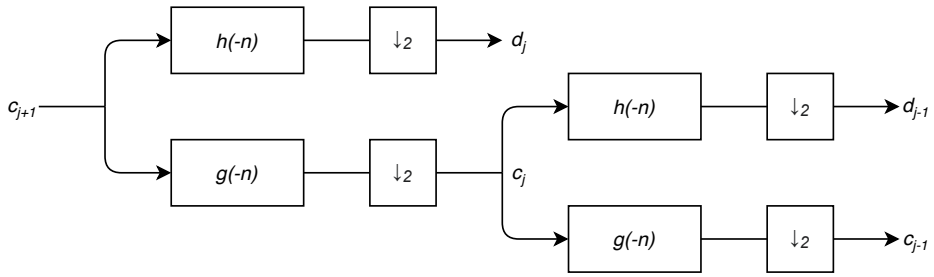


Figure 2.5: Two-stage two-band analysis tree

Each stage implements two digital filters, and two down-samplers (sometimes also called *decimators*). One of the filters, denoted as $h(...)$, is a high-pass filter, which is letting only higher frequencies of a signal through. The other one, denoted as $g(...)$, is a low-pass filter doing the opposite. The filters are where the wavelets play their role, and the filtering is achieved by convolving the signal sequence with the filter (wavelet) coefficients. The down-samplers discard half of the data that come through them; in this case, taking every other (even) value as $y(n) = x(2n)$. This decimation causes rougher resolution with every stage of the algorithm because we apply the same filter to fewer data. Also, it causes that the output from the tree has, on average, the same length as the input, meaning that there is a possibility of no information loss, and therefore complete signal recovery.

The outputs from high-pass filters d_i are called *detail coefficients*, and the outputs from low-pass filters c_i are called *approximation coefficients.*

One stage of the tree in figure 2.5 is implemented by two recursive equations. The equation for the high-pass filter coefficients:

$$d_a(b) = \sum_m h(m - 2b)c_{a+1}(m), \tag{2.2.1.12}$$

and the one for the low-pass filter coefficients:

$$c_a(b) = \sum_m g(m - 2b)c_{a+1}(m), \tag{2.2.1.13}$$

where $n = m - 2b$ to correspond with the figure 2.5. Also note, that the tree uses time-reversed recursion coefficients $h(-n)$, and $g(-n)$.

Summary

We have explained differences and (dis)advantages of Fourier, Gabor, and wavelet transforms. Figure 2.6 is quite a famous picture to visualize these differences. Understanding this cartoon is critical for understanding the wavelets and the wavelet transform.

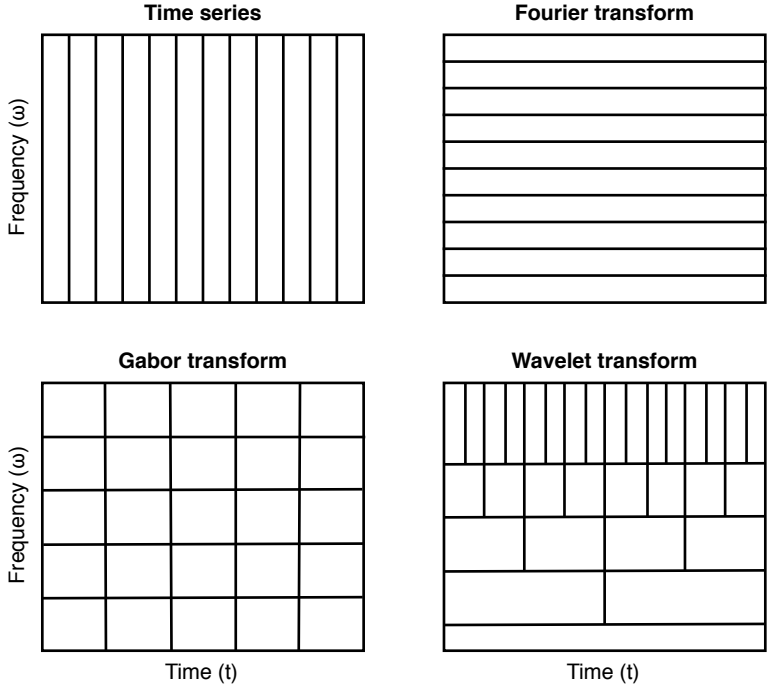


Figure 2.6: Transform comparisons

It shows us that a time series, the top left box, has all the time information we can get. We know what exactly happens in every point of time in the signal, but it tells us nothing about the magnitude of the signal at different frequencies.

The second box, the top right one, visualizes the Fourier transform of the time series. We cast the signal from the time domain into the frequency domain and therefore lose all the time information. So, as described above, we have perfect frequency resolution, but no way to say or reconstruct when any event in the signal happened. That is why Fourier transform expects a stationary signal in time.

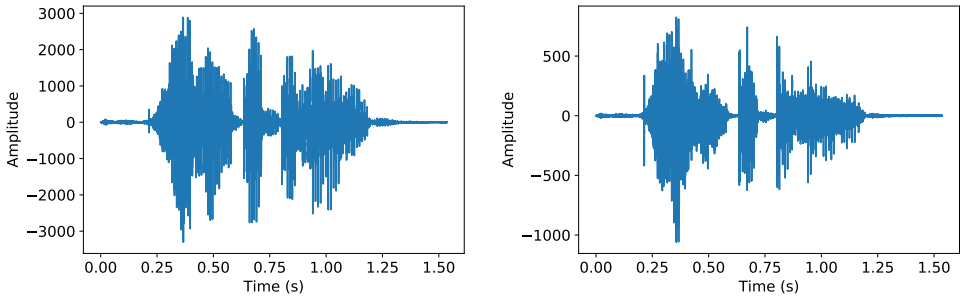
The third box, the bottom left one, is the Gabor transform. We are splitting the signal into windows and applying Fourier onto those windows.

We trade some time resolution for frequency resolution. It is a trade-off, a compromise where the window length parameter says, how much of each we want to trade.

The last box, the bottom right one, is the wavelet transform. It starts at the bottom of the box. The first thing the algorithm does is to take the whole signal and (to some extent) perform a transform on it (with the mother wavelet scaled to the entire signal). Now, we take the mother wavelet, scale it to half its size and slide it across the time series. Repeat this until we have enough granular time resolution for the analysis we need to perform. After this, we have high frequency (bottom of the box) resolution and also the time (top of the box) resolution.

2.2.2 MFCC Features

The goal of this section is to describe the transformation process of the digital signal into a sequence of acoustic feature vectors. Each of the vectors represents the information in a small time window of the signal. This section draws from [25]. Most of the figures in this section are my own, created with help from [26]. The original wave file, which is the whole process of feature extraction demonstrated on, is my recording saying "yellow dandelion."



(a) Before (b) After

Figure 2.7: Digital signal before and after the preemphasis

Preemphasis

The first step in this process is to boost the amount of energy in the high frequencies. If we look at the frequency spectrum of voiced segments like vowels, there is more energy at the lower frequencies than the higher frequencies. This drop of energy is called the *spectral tilt*.

This preemphasis is done by applying a filter which goes as

$$y_n = x_n - \alpha x_{n-1}, \tag{2.2.2.1}$$

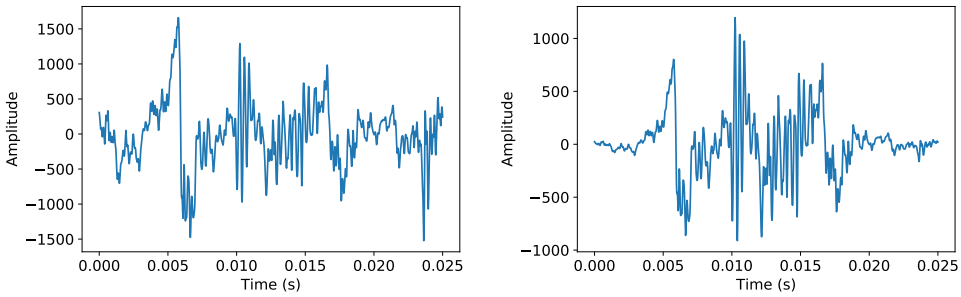
where signal x and parameter $0.9 \leq \alpha \leq 1.0$ are the inputs and y is the signal after preemphasis. Figure 2.7 shows a digital signal before and after the preemphasis is applied, where the value of α is set to 0.97.

Windowing

Since the goal of this feature extraction is to get spectral features that help us build a phone or sub-phone classifier, we do not want to extract our features from an entire recording of a sentence or whole conversation. In such long pieces, the spectrum changes very quickly. Technically, speech in a *non-stationary* signal, meaning that its statistical properties are not constant across time. Therefore, we need to extract spectral features from a small *window* of speech that characterizes a single sub-phone. For this window, we can make a rough assumption that the signal is stationary (i.e., its statistical properties are constant within the window).

The windowing process takes three parameters:

- the **width** of the window (in milliseconds),
- the **offset** between successive windows, and
- the **shape** of the window.

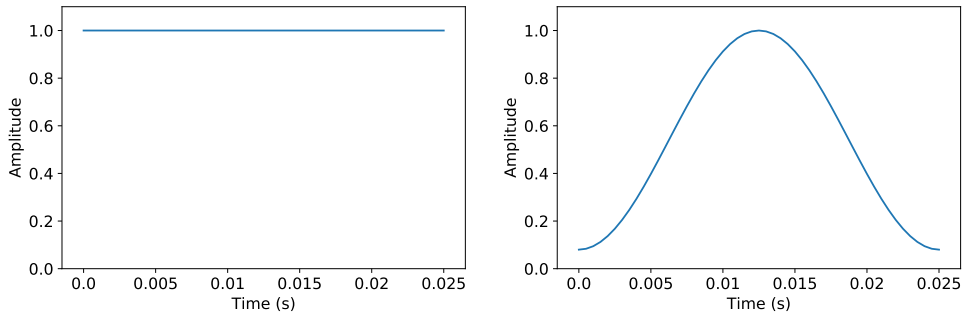


(a) Before (b) After

Figure 2.8: A frame before and after the Hamming window function application

Each piece of speech extracted by the window function is called a *frame*, and its length in milliseconds the *frame size*. A number of milliseconds between the left edges of two consecutive frames (their overlap) is *frame shift*. Figure 2.8 shows the 40th frame of the original signal before and after the application of the Hamming function. We can see that the length of each frame is 25 ms.

The third parameter, the shape of the window, can provide additional transformation to the frame. The most basic type of window is the *rectangular* window, which keeps the signal as is. That can cause problems, however,



(a) Rectangular window (b) Hamming window

Figure 2.9: Window functions

because it abruptly cuts the signal at the window edges. Such discontinuities create issues when we use the Fourier transform. That is the reason a more typical window used in MFCC extraction is the *Hamming* window. It shrinks the values of the signal near the boundaries toward zero, avoiding discontinuities. Figure 2.9 shows both the rectangular window function and the Hamming window function. We can see how are the left and right edges being squished close to the zero when the function is applied in the case of the Hamming window.

Discrete Fourier Transform

The next step is extracting the spectral information from our windowed signal – we need to know the magnitude of energy the signal contains at different frequency bands. The tool used to extract spectral information for discrete frequency bands for a discrete-time (sampled) signal is the *Discrete Fourier Transform* (DFT).

The input for the DFT is signal $x_m ... x_n$, and the output is a complex number X_k which represents the magnitude of that frequency in the original signal (frame). Plotting the magnitude against the frequency visualizes the *spectrum*. Figure 2.10 shows such spectrum computed for the same frame, on which was the hamming window function demonstrated.

The definition and the intuitive idea of the discrete Fourier transform is introduced in section 2.2.1. Popular algorithm for the computation of the DFT is the Fast Fourier Transform (FFT). This implementation is very efficient with on constraint – N must equal to values which are powers of two.

Mel Filter Banks

This part of the extraction process tries to simulate the human hearing, which is not equally sensitive at all frequencies. It is, in fact, less sensitive at higher frequencies. Modeling this property of human hearing in the feature extraction

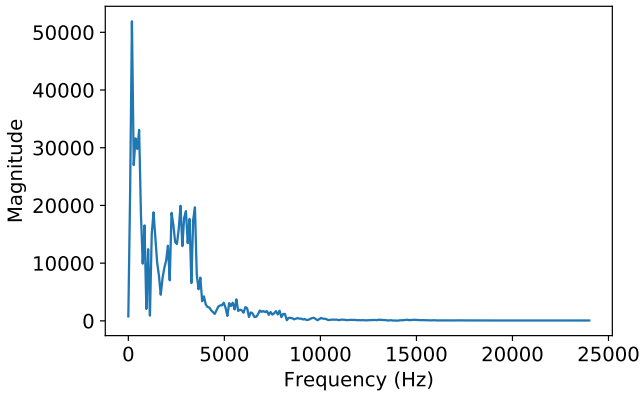


Figure 2.10: A frame spectrum computed with DFT

process turned out to improve the speech recognition performance. It is done by warping the output from DFT onto the *mel* scale. A *mel* is a unit of pitch defined so that an equal number of mels separates a pair of sounds that are perceptually equidistant in tone. In essence, two pairs of adjacent notes in different octaves would have different distances in hertz, yet the same distance in mels. Mels can be computed from raw frequency as follows:

$$mel(f) = 1127 \ln\left(1 + \frac{f}{700}\right), \tag{2.2.2.2}$$

and therefore computing frequency from mels:

$$f(m) = 700\left(e^{\frac{m}{1127}} - 1\right). \tag{2.2.2.3}$$

There are ten filters spaced linearly below 1000 Hz, and the remaining filters logarithmically above 1000 Hz – usually 16 more, giving 26 filters in total.

In figure 2.11, there is visualized a spectrogram of the original signal. It is 152 feature vectors (one vector for each frame) of length 26 (number of triangular filters in the bank). The spectrogram is normalized by mean normalization to make the image clearer.

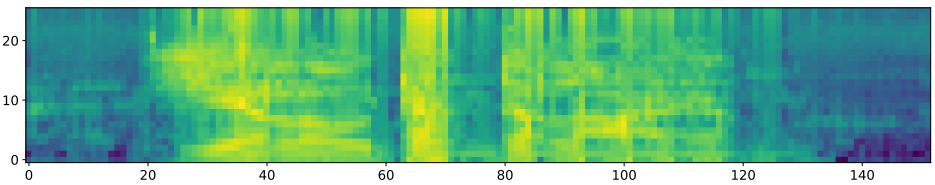


Figure 2.11: Spectrogram of the original signal

Cepstrum

It would be possible to use the mel spectrum by itself as a feature representation for phone detection. Yet, there is still a lot of unimportant information in the so-far extracted features. One way to further refine the features is the computation of the so-called *cepstrum*.

The cepstrum can be thought of as a *spectrum of the log spectrum*. We take the computed spectrum and apply a logarithmic function to it. Now, we visualize the logarithmic spectrum as if it were an ordinary waveform – its domain is still frequency, but we pretend the domain is time. And the last step is computing the spectrum of this pseudo-waveform we have just created. Generally, we only take the first 12 cepstral features, that solely represent the information of the human vocal tract without unnecessary information. Figure 2.12 visualizes the extracted cepstral features from the original signal.

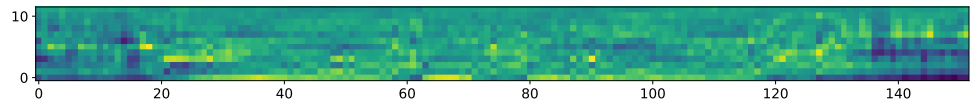


Figure 2.12: Cepstrum features of the original signal

The formal definition for the cepstrum is *inverse DFT of the log magnitude of the DFT of a signal*. Mathematically:

$$c_n = \sum_{n=0}^{N-1} \log \left(\left| \sum_{n=0}^{N-1} x_n e^{-\frac{i2\pi}{N} kn} \right| \right) e^{\frac{i2\pi}{N} kn}. \tag{2.2.2.4}$$

It turns out that cepstral coefficients have the handy property that the variance of the different coefficients at different frequency tends to be uncorrelated. That is not true for the spectrum. Spectral coefficients at various frequency bands are correlated. That significantly eases the job for the acoustic model.

Deltas and Energy

We have got 12 features extracted from each frame. Now, we next add the thirteenth feature: the energy from the frame. The energy correlates with phone identity and therefore is useful for phone detection (vowels and sibilants have more energy than stops, etc.). It is the power sum over time in the frame. Thus, for a signal x in a window from time t_1 to time t_2:

$$E = \sum_{t=t_1}^{t_2} x_t^2. \tag{2.2.2.5}$$

In the next step, since the speech signal is not constant from frame to frame, we are adding features related to the change in cepstral features over

time. We do this by adding for each of the current 13 features a delta or velocity feature, and a double delta or acceleration feature.

Each of the 13 delta features represents the change between frames in the corresponding cepstral/energy feature. Each of the 13 double delta features represents the change between frames in the corresponding delta features. A simple way to compute the deltas is by calculating the difference between frames. Thus, the delta value d_t for a particular cepstral value c_t at time t is:

$$d_t = \frac{c_{t+1} - c_{t-1}}{2}. \tag{2.2.2.6}$$

Usually, a much more sophisticated estimate of the slope, which uses a broader context of frames, is used.

Summary

Now, when we have all the features computed, we are at 39 MFCC features for each frame. In table 2.1, we can see the summary of the features with counts for each category.

Table 2.1: MFCC features summary

12	cepstral coefficients
12	delta cepstral coefficients
12	double delta cepstral coefficients
1	energy coefficient
1	delta energy coefficient
1	double delta energy coefficient
39	MFCC features

Again, the most useful fact about MFCC features is that the features are uncorrelated, which turns out to make the acoustic model much simpler.

2.2.3 Wavelets in Feature Extraction

Since the usage of wavelets in feature extraction is still considered relatively new, and most of the models still use MFCC vectors in practice, we are going to introduce a few of those approaches in this section superficially. Some research papers propose to substitute the inverse Fourier transform (or the discrete cosine transform) in MFCC vectors with the discrete wavelet transform (DWT) [27]. Those features are called MFDWC vectors. Another research suggests first applying the DWT to the signal to extract the approximation coefficients and then computing MFCCs from those coefficients [28]. The third extraction method is the *wavelet packet decomposition* (WPD), or sometimes called the *wavelet packet transform* (WPT), which is one of the implementations for the DWT [29, 30].

MFDWC Features

MFDWC stands for *mel-frequency discrete wavelet coefficients*, and it was proposed by [27]. Those features are obtained by applying the DWT to the mel-scaled log filterbank energies of a speech frame, thus not using the discrete cosine transform (DCT) as in MFCCs. Using the DWT has the benefit of its localization properties, which are described in section 2.2.1. This approach is trying to deal with two drawbacks of the DCT. Again, this has already been described above, and we will only recap the main points. These are the drawbacks:

1. the DCT basis covers all frequency bands. That means that a corruption of the speech signal on one band affects all the MFCCs. However, if we use a localized transform instead of DCT, corruption could be reduced to only a few coefficients.

2. One frame of speech may (and very probably will) contain information about two adjacent phonemes. If one of them is voiced and the other is not, then the voiced phoneme might dominate the low-frequency spectrum and vice versa. Since MFCCs assumes a stationary signal in one frame, it also assumes there can only be one phoneme present in one frame.

MFDWCs consist of fifteen static feature coefficients for each frame. There are eight coefficients at scale four, four coefficients at scale eight, two at scale sixteen, and one at scale thirty-two. The feature vector also includes delta coefficients and delta energy to include dynamic features. The delta extraction is described in section 2.2.2 for the MFCCs.

These proposed features performed better than other speech features tested in the paper. Those included SUB (subband-based), MULT (multi-resolution), and MFCC features. MFDWCs had relatively better performance on a noisy signal, and the difference from other approaches on a clean signal was lower. The experiments were conducted on the TIMIT database on a phoneme recognition task.

DWT-Based MFCCs

This approach, proposed in [28], computes the MFCC feature vectors from the approximation coefficients of the DWT of a speech signal. Again, the reason to use the wavelet transform is to reduce the impact of noise in a signal.

The extraction consists of three phases:

1. Use a voice activity detection (VAD) model to separate speech and non-speech (utterances and silence between words) segments,

2. compute the discrete wavelet transform of the speech signal, and get the approximation coefficients, and finally

3. compute the MFCCs from the approximation coefficients.

This model was tested on the PIEAS speech database. It contains 2000 short samples of both telephonic (noisy) and non-telephonic (clean) speech of both males and females. Its goal was speaker recognition, and these DWT based MFCCs outperformed LPC (linear predictive coding), especially in noisy speech. Comparison with standard MFCCs was unfortunately not present.

Wavelet Packet Decomposition

Wavelet packet decomposition (WPD) is an implementation of the DWT, which decomposes both sides of approximation and detail coefficients. Therefore, it is creating a whole binary tree as opposed to the classic DWT, which decomposes only one branch in each stage. Such a tree is depicted in figure 2.13.

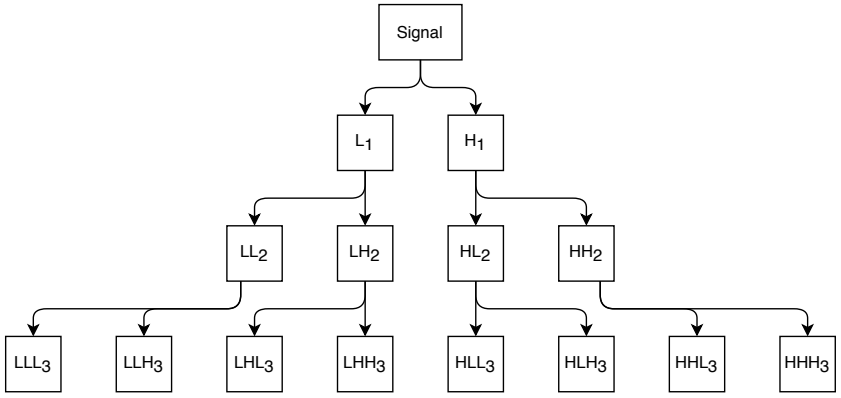


Figure 2.13: Wavelet packet decomposition tree, inspired by [31]

The WPD produces 2^n sets of coefficients (for each leaf node) as opposed to the standard DWT, where n is the level of decomposition. The standard DWT provides only $n + 1$ coefficients – one for each level from the high-pass filters, and +1 from the last level as the output from the low-pass filter. That lets us choose from a wider variety of coefficients and experiment with different combinations.

Research [30] experiments with the recognition rate of Indonesian syllables, and uses three datasets: a training syllable dataset DLSK, a testing syllable dataset DUSK, and a testing word dataset DUK. It compares MFCC feature vectors and WPD (with DB3 and DB7 wavelets) as a feature extraction method on a *hidden Markov model* (HMM) with Euclidean distance as a classifier. The best results for the MFCC method are 75/38/47 % for the DLSK/DUSK/DUK datasets, respectively. As for the WPD approach, the best results are 70.8/37.5/53.1 %. Therefore, it depends on a specific problem, which one of the strategies to use.

Research [29] deals with speaker recognition over two datasets. The first one is a custom dataset created in laboratory conditions, and it consists of 783 recorded samples of both male and female speech. These samples are several minutes long, and there are ten speakers. The second dataset consists of human dialogues of fifty speakers created for Czech Railways.

There are four conventional methods compared with the WPD method: MFCCs, *perceptual linear prediction coefficients* (PLPC), *linear prediction reflection coefficients* (LPREFC), and *linear prediction cepstral coefficients* (LPCEPSTRA). All of these were tested on two models: a *Gaussian mixture model* (GMM) and a *multilayer perceptron* (MLP).

The WPD, in this case, uses three discrete wavelets: *Daubechies*, *Symlets*, and *Coiflets*. It computes the *fifth* level of decomposition, which therefore outputs 2^5 sub-signals. Those sub-signals are then used to calculate the final feature vectors as follows:

$$f_i = \log_{10}\left(\frac{1}{N_i}\sum_{k=1}^{N_i-1}\left|(w_i(k))^2 - w_i(k-1)\cdot w_i(k+1)\right|\right), \qquad (2.2.3.1)$$

where f_i are values of the feature vector (one value for each leaf node, thus $2^5 = 32$ values), $w_i(k)$ is the k-th coefficient in leaf node i, and N_i is the number of coefficients in leaf node i.

The results are:

- MLP requires less training data per speaker than GMM to classify speakers correctly. In this case, it needed two times fewer data.
- The minimal utterance duration for correct classification is two seconds for both models and all feature extraction methods.
- The accuracy results are dependent on the model:
  - GMM: all three wavelet families had very similar results (with the Symlets in the lead), and they were all slightly outperformed by other conventional methods.
  - MLP: all methods were clustered together with LPREFC and LPCEPSTRA in the lead. The only exception were MFCCs, which were far behind every other approach.

The choice of feature extraction method again seems to be dependent on the task we want to perform, and on the dataset the model needs to work with. All of the presented approaches had very similar results, except for the significant negative deviation of MFCC features in the MLP model.

2.3 Connectionist Temporal Classification

When we think about training a neural network on a speech dataset, we expect it to output a sentence after we feed it with a speech utterance. The first problem that pops up is that our dataset consists only of several seconds long utterances and labels. There is no information about where each of the letters or phonemes exactly is "located" in the sound wave. We could slice the audio into tiny time-frames, annotate each slice with a letter from an alphabet (plus a space symbol), and teach the neural network on this new dataset. However, that would be very time-consuming and boring. Apart from the time inefficiency, the other problem is that some people speak more slowly than others and some phonemes take longer to pronounce. Thus, some letters would take several time-frames in the dataset for themselves. The word "dog" could be therefore labeled as "dooogg."

Moreover, there are many words with duplicate characters, and some of them differ only in that property (*to* vs. *too*). The network would then be either outputting non-existent words or needing a sophisticated processing algorithm to decode the text. That is where the CTC plays its role. [32, 33]

The CTC loss function only needs to know the text that occurs in the audio (the ground truth) without the position context or phoneme width and the predicted network output. The algorithm does not know where the characters are located; instead, it tries all possible alignments of the reference label to the predicted text and sums the scores.

2.3.1 Blank Symbol Importance

The introduction of a *blank* character aims to solve the duplicate character problem. It is denoted as "-" has nothing to do with the standard space character. The algorithm can insert arbitrary many blank symbols at any position as well as repeat any characters as it likes. There is only one rule: a blank symbol must be inserted between duplicate characters (e.g., in the word "fu**ll**y"). The decoding then goes as:

1. remove repeated characters,
2. remove blank symbols.

If $\mathcal{F}$ were such a decoding function, the decoding process would look like this: $\mathcal{F}(-aa--abb) = \mathcal{F}(-a--ab) = aab$. That gives the CTC power to differentiate between duplicated characters and long pronunciation of phonemes.

2.3.2 Loss Calculation

As mentioned above, the loss is calculated by summing up scores of all possible alignments (or *paths* [32]). The neural network outputs a character probability matrix. Figure 2.14 depicts a tiny example of such a matrix. [33]

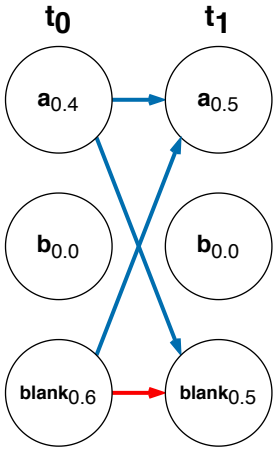


Figure 2.14: NN output matrix with character probabilities

The score for one path is computed by multiplying the corresponding character scores together. If we were to compute the score for the path "aa", we would get $0.4 \cdot 0.5 = 0.2$. In the same way, the score for the "-a" path is $0.6 \cdot 0.5 = 0.3$, and $0.4 \cdot 0.5 = 0.2$ for the "a-". To get the score for a prediction given a ground truth, we would compute scores for all paths corresponding to the ground truth after decoding. With this in mind, if the ground truth were "a", the corresponding paths would, therefore, be "aa", "-a", and "a-" (only paths of length two because the matrix has two time-steps). Those paths are highlighted with blue color in figure 2.14. A red color highlights a path corresponding to an empty string. There are no paths containing character "b" since the character has zero probability in both time steps. We have already computed the three correct path scores, so their sum gives $0.2+0.3+0.2 = 0.7$. The CTC loss is then a negative logarithm from the probability. Mathematically:

$$\mathcal{L}(x, z) = -\ln p(z|x), \tag{2.3.2.1}$$

where $\mathcal{L}$ is the CTC loss function, x is a speech utterance, z is the ground truth label, and $p(z|x)$ is a probability of predicting z given x. The loss for the whole dataset S is then:

$$\mathcal{L}(S) = -\sum_{(x,z)\in S} \ln p(z|x). \tag{2.3.2.2}$$

2.3.3 Decoding

Once the network is trained, we want to label an unknown input x. Ideally, label the input with the most probable labeling l^*:

$$l^* = \arg\max_l p(l|x). \tag{2.3.3.1}$$

However, there is no general method of choosing the most probable path, in other words *decoding*, for CTC. We are going to go through two popular methods that work in practice.

Best Path Decoding

The first, and the most straightforward method, is called the *best path decoding*. It assumes that the most probable path is the most probable labeling:

$$l^* \approx \mathcal{F}(\pi^*), \tag{2.3.3.2}$$

where π is a path, and $\pi^* = \arg\max_\pi p(\pi|x)$. Therefore, the best path is easy to compute, since π^* is just a concatenation of the most probable characters from each time-step. However, this method can lead to error, especially if a label is predicted poorly for several consecutive steps.

Prefix Search Decoding

The second method, *prefix search decoding*, utilizes the fact that we can efficiently calculate the probabilities of successive extensions of labeling prefixes.

Meaning we have a tree, where nodes are labels. Children nodes are labels that have their parent as a prefix (e.g., "aaab" and "aaac" would be children nodes of node "aaa"). In such a tree, we can efficiently calculate the cumulative probabilities of all labels with a particular prefix. It is solved with a dynamic-programming algorithm – see [32] for more details.

The search itself is then a *best-first search*, which in each step expands the node with the highest score. The score is, in this case, the total probability of all labels beginning with that label as a prefix. This search process ends when a single label is more probable than any other remaining prefix.

With enough time, prefix search decoding always finds the most probable label. However, the search space grows exponentially with increasing sequence length. Thus, for many tasks, a heuristic is required to make the search applicable.

Decoding Constraints

When it comes to speech, we often want to constraint the output labeling to a pre-defined grammar; we want the transcriptions to form sequences of dictionary words. Also, it is common practice to use a *language model* to take probabilities of specific sequences of words into account. Incorporation of the constraints in the search for the most probable labeling, as in equation 2.3.3.1, would look like:

$$l^* = \arg\max_l p(l|x, G). \tag{2.3.3.3}$$

A *language model* is usually a tree-like structure holding probability distribution over sequences of words. In other words, it gives us a probability

$P(w_1, w_2, ..., w_m)$ of a sequence of words w_i. Each node in this tree structure represents a sequence (actually a word but its parent nodes are the rest of the sequence) and also gives as probabilities for the following words. Order of such a tree is the depth of the tree, i.e., the number of words taken as a context for the next prediction. [34]

One could say that these constraints and additional probabilities contradict the assumption of input sequence independence of the CTC. Nevertheless, since the model takes the whole sequence labeling, and only then the external grammar or constraints are applied, it has enough space to learn the inter-label transitions from the data. So as long as the constraints focus on long-term dependencies, it does not interfere with the CTC's dependencies modeled internally.

Chapter 3

Experiments

Apart from the ASR system realization, this thesis compares the performance of two feature extraction methods for a speech audio signal. It examines the Mel-frequency cepstral coefficient (MFCC), and the wavelet packet decomposition (WPD) approaches.

The MFCCs use a standard definition with 26 DCT coefficients. The feature vectors lack the delta coefficients, which were described in section 2.2.2. It is the default approach used by DeepSpeech (described in section 3.1), and it is implemented using the Tensorflow [35] package. The latter approach implementation is a part of the practical section of this thesis and is implemented in Python 3 using the Tensorflow package. It uses a wavelet proposed by [36], referred to as the *Haar classic wavelet function.*

The goal is to substitute the original feature extraction method used in DeepSpeech with the newly proposed one and test its performance.

There are two datasets used for this experiment: the initial CPM dataset, and the Free Spoken Digit Dataset (FSDD). Initially, the experiments were meant to be conducted only over the CPM dataset. A different dataset had to be used after a few attempts to train the network with WPD on this dataset. The CPM dataset is extensive and contains recordings that are over ten seconds long. Moreover, it consists of a whole vocabulary of words, so the classification is not an easy task, whatsoever. Therefore, the FSDD dataset (section 3.2.2) was introduced. Both of them are described in the following sections alongside the model used for the testing.

3.1 Model – The DeepSpeech Project

DeepSpeech [37] is an open-source Speech-To-Text project by Mozilla. It comes with a pre-trained English model and the possibility to train a model in a different language on custom data. It provides a large variety of tweakable hyperparameters alongside means of signal augmentation (frequency, pitch, and speed). The model training can be executed in parallel over multiple GPUs, which leads to much faster learning. The model uses machine learning (ML) techniques based on Baidu's Deep Speech research paper [38] and is implemented using the TensorFlow [35] libraries/framework by Google.

Currently, the engine differs in many ways from the original design. Its core is a recurrent neural network, specifically LSTM, which takes speech spectrograms as input and generates text transcriptions.

The process of converting speech signal into text transcriptions is described for the English language as [39]:

Let x be an utterance and y its label from a training set

$$S = \left\{ (x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots \right\}. \tag{3.1.0.1}$$

Each utterance $x^{(i)}$ is divided into $T^{(i)}$ time-frames, while each frame is represented by audio features $x_t^{(i)}$, where $t = 1, \dots, T^{(i)}$. Since DeepSpeech uses MFCC vectors by default, the features are denoted $x_{t,p}^{(i)}$, which is the p-th MFCC feature of the t-th audio frame in utterance $x^{(i)}$. The goal is to convert a sequence x into a sequence of character probabilities for the transcription y, with $\hat{y}_t = P(c_t|x)$ – the conditional probability of c_t given x. In English, $c_t \in \{a, b, c, \dots, z, space, apostrophe, blank\}$.

The RNN model consists of 5 hidden layers. Layers are denoted $h^{(l)}$, where $h^{(0)}$ is the input layer. The whole schema of the network is depicted in figure 3.1. The recurrent units (LSTMs) are dependent on $C = 9$ audio frames on each side as a context for the currently computed result.

The backpropagation algorithm for MLP (non-recurrent) network is described in section 1.2.1. The DeepSpeech model uses a clipped rectified linear unit (ReLU) defined as $g(z) = min\{max\{0, z\}, 20\}$. The LSTMs are described in section 1.3.1. For the notation purposes, let us show, how are the outputs for the first three layers computed:

$$h_t^{(l)} = g\left(W^{(l)} h_t^{(l-1)} + b^{(l)}\right), \tag{3.1.0.2}$$

where $W^{(l)}$ and $b^{(l)}$ are the weight matrix and bias parameters.

The output layer uses *logits* that correspond to the predicted character probabilities for each time frame t and character k in the alphabet:

$$h_{t,k}^{(6)} = \hat{y}_{t,k} = \left(W^{(6)} h_t^{(5)}\right)_k + b_k^{(6)}. \tag{3.1.0.3}$$

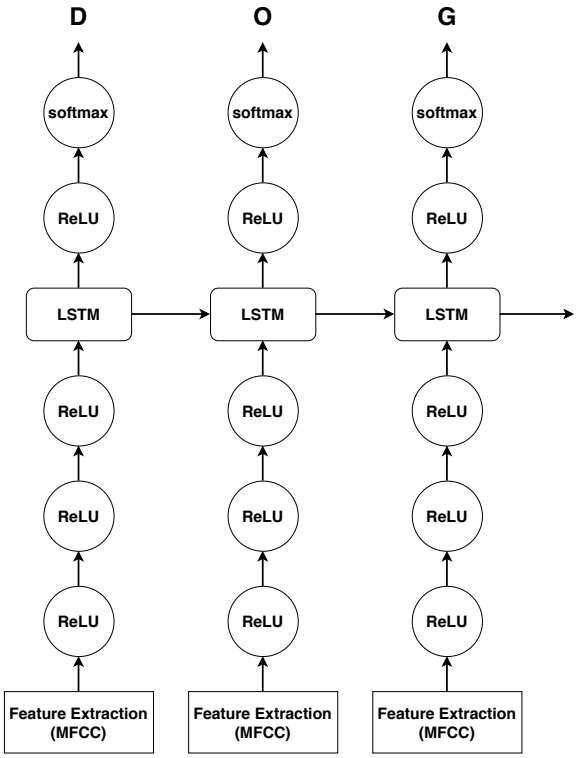


Figure 3.1: DeepSpeech RNN schema, based on [39]

In Math, *Logit* is a function that maps probabilities $[0, 1]$ to real values $(-\infty, \infty)$, where the probability of 0.5 corresponds to a logit of zero. In ML, it can be a vector of non-normalized predictions generated by a model. [40]

Once we have a prediction $\hat{y}_{t,k}$, we compute the *connectionist temporal classification* (CTC) loss $\mathcal{L}(\hat{y}, y)$. CTC loss function is also the reason we need a *blank* "symbol" in the character sequence S, as described in section 2.3.

3.2 Datasets

This section describes both datasets used in this thesis. Apart from the Czech Parliament Meeting (CPM) dataset, the Free Spoken Digit Dataset (FSDD) is used only for the experiments.

3.2.1 Czech Parliament Meetings Dataset

The Czech Parliament meetings (CPM) dataset [41] are recordings from the Chamber of Deputies of the Parliament of the Czech Republic. It consists of 88 hours of annotated speech data. The dataset contains 18 audio files

in 16-bit, 44.1 kHz raw WAV format, and a single-channel sound. For each of the recordings, there is a corresponding transcription in the XML-based format. In figure 3.2, there is an example of the XML transcription used in the dataset.

```
<Sync time="3528.820"/>
<Event desc="speaker" type="noise" extent="instantaneous"/>

<Event desc="hesitation" type="noise" extent="instantaneous"/>

<Event desc="rozhodl" type="pronounce" extent="begin"/>
rozhodnul
<Event desc="rozhodl" type="pronounce" extent="end"/>
 o uveřejnění těchto smluv a to v takovém rozsahu že jsem je
<Event desc="speaker" type="noise" extent="instantaneous"/>
<Sync time="3533.345"/>
úplně všechny
<Event desc="speaker" type="noise" extent="instantaneous"/>
 nechal zveřejnit na internetových stránkách aby byly volně přístupné,
<Event desc="speaker" type="noise" extent="instantaneous"/>
 skutečně to není mým účelem,
<Event desc="speaker" type="noise" extent="instantaneous"/>
<Sync time="3541.060"/>
```

Figure 3.2: XML annotation example

The transcriptions were created for both acoustic model training for ASR, and models for speaker recognition or verification. Therefore, they contain a wide variety of tags and information:

- Every transcription contains a header with the name of the corresponding audio file, version of the transcription, and date.
- After the header, there is a list of speakers present in the transcription with information like name, ID, dialect, or accent.
- The rest of the file is divided into:
  - **Episode** tag that encapsulates the whole transcription and does not play any role in this case.
  - **Section** tag that is also only once in each of the files. This time it contains the type property (e.g., report), start time, and end time.
  - **Turn** tags, which divide the file into speech sections of one speaker. They contain information about start time, end time, mode (e.g., spontaneous), and speaker ID.
  - There are also **Event** tags, which have either a single-line occurrence or may occur in pairs. They are characterized by their type (e.g., noise, or pronounce), description, and extent. If the *type*

property is "noise," the extent is instantaneous. If the *type* is "pronounce," the tag is paired with and ending one. They then enclose a word or a phrase, which was pronounced differently than it is written in the transcription. In the latter case, the description property of this tag contains the real transcription of the word, which is captured in the audio file.

– The most crucial elements for this thesis are the **Sync** tags, which contain a synchronization timestamp. Together with the "Turn" tag time properties, it is possible to cut the audio files and assign corresponding transcriptions to the segmented pieces.

3.2.2 Free Spoken Digit Dataset

The Free spoken digit dataset (FSDD) [42] is an open speech dataset which consists of English spoken digits. There are 2500 recordings by five speakers where each numeral from zero to nine is pronounced fifty times by each speaker.

The dataset contains raw WAV mono-channel audio recordings with an 8 kHz sample rate. Recordings are appropriately trimmed, so there remains only a minimal leading and trailing silence.

The individual files are named as *[digitLabel]_[speakerName]_[index].wav*, meaning the label for each of the recordings is encoded in its name. The repository also contains tools to either extend the dataset with one's voice or preprocess it. Neither of those was necessary for the purposes of this thesis. The steps to preprocess this and the CPM datasets are described in the following section.

3.3 Data Preprocessing

This section describes the data preprocessing for the two datasets before they can be fed to a neural network. The more complex one, the CPM dataset, is introduced first, followed by the FSDD dataset. This section also begins the practical part of this thesis. Meaning, the knowledge presented from now on is the result of my implementations, unless explicitly stated otherwise.

3.3.1 CPM Dataset

Since the data needs to be fed to a neural network, we have to create relatively short audio segments with text labels. So the first step of the data preprocessing was to locate the time-synchronization tags. Each subsequent pair of tags meant one audio segment, and for that segment, a corresponding part of the transcription had to be isolated.

Each audio segment was then downsampled to 16kHz. That is a high-enough sample rate for human speech recognition, and yet small file size for

relatively quick manipulation. Each segment was exported as a new wave file alongside its corresponding transcription text file. With each export, an entry to a CSV file was generated to be later able to easily locate the files and their labels for the neural network.

To ease the job for the model, and for it to "focus" on the phoneme side of the problem, there are a few filters applied:

- all the labels are lower-cased,
- the punctuation is removed,
- labels are stripped of leading and trailing white-spaces,
- segments with labels containing non-alphabetic characters are excluded, and
- wrongly pronounced words or phrases, which are correctly written in the transcription, are substituted with their wrongly written equivalents.

Additionally, data with empty labels had to be later excluded because the chosen model could not work with such data.

DatasetManipulator Class

The "DatasetManipulator" class manages the process above. The *main.py* file, located in the root directory, manages the class. It has two options:

- **-i/–inputdir**, where a user specifies the dataset directory, and
- **-n/–njobs**, which is the number of parallel threads to execute the task. Since each of the threads will read one of the files, and load it into RAM, it is crucial to choose a reasonable value.

The program output is a folder named *cpm_cut* right next to the dataset directory, which contains the following:

- audio wave files for each segment,
- corresponding text files with transcriptions,
- a CSV file called *data.csv* with an entry for each transcription/audio pair with the name of the audio file and a string label, and
- three more CSV files named train.csv, test.csv, and dev.csv. Those contain audio file paths, audio size, and transcription. Data is randomly distributed between those three CSV files as 70/20/10 according to train/test/dev.

The CSV file format DeepSpeech expects has three columns: absolute path to the file, file size in bytes, and the transcription. It also contains a header line: "wav_filename,wav_filesize,transcript".

3.3.2 FSDD

This dataset was preprocessed using the *spoken_digit_preprocess.ipynb* script. The process consists of four steps:

1. Translating numeric labels into strings using a dictionary,
2. upsampling the 8 kHz audio files to 16 kHz as required by the DeepSpeech model,
3. converting the files into the right WAV format type, and
4. generating the CSV files required by the DeepSpeech model. This step and the CSV file format are described in section 3.3 for the CPM dataset.

The other steps were implemented using the *pydub* Python library.

3.4 Implementation

The WPD is implemented as a static class called *HaarClassicWPD*. It follows the implementation proposed by [36], which defines the low-pass and high-pass filters as

$$c_{j+1,i} = \frac{x_{j,2i} + x_{j,2i+1}}{2} \tag{3.4.0.1}$$

and

$$d_{j+1,i} = \frac{x_{j,2i} - x_{j,2i+1}}{2} \tag{3.4.0.2}$$

accordingly, where $x_{j,i}$ is the j-th value of a (sub)signal x of the j-th level of decomposition. The indexing and naming on right sides were changed, so they would comply with the figure the author included in the source. Names on the left sides were changed so they would match the coefficient naming used in section 2.2.1.

The class contains two essential methods:

- *get_level(signal, level)*, which computes the WPD using the filters 3.4.0.1 and 3.4.0.2 defined above, and thus returning 2^n sub-signals, where n is the specified level.
- *get_features_level(signal, level)* uses the above method and converts its output into a feature vector of 2^n values as $f_i = \ln\left(\sum_{j=0}^{N} e^{x_{i,j}}\right)$. There, N is the length of a sub-signal x_i, and $x_{i,j}$ is the j-th value of the i-th sub-signal. [29] inspired this process, yet it is not entirely identical.

When integrated into the DeepSpeech project, the input signal is framed into 512 samples-long segments. The class then processes those segments, and the output feature vectors are sent further into the computational graph. The WPD feature vectors have to be shortened to 26 values (the same length

as the MFCCs) to integrate the class seamlessly; therefore, the *fifth* level of decomposition ($2^5 = 32$ sub-signals) is computed for each frame.

A copy of the DeepSpeech file using the custom class is in the *speech2text* repository [43], which is a part of this thesis (more on this later). The file is called *feeding_custom.py*, and its original counterpart is in "DeepSpeech/util/feeding.py" in the DeepSpeech repository. The function to look for is called *samples_to_mfccs*, and, along with another explicitly added function, is highlighted by hash (comment) characters.

3.5 Results

The experiments were performed on a GPU server. Its specifications were:

- **CPU:** 2x Intel® Xeon® Silver 4114
- **GPU:** 2x Nvidia RTX 2080 Ti GPU
- **RAM:** 64 GB DDR4
- **OS:** Ubuntu 18.04

The computations were performed in the Docker container environment described in section 4.1.

Table 3.1: Experiment results

dataset	features	width	dropout	CER	WER	loss
CPM	**MFCC**	**512**	**0.25**	**0.0778**	**0.1886**	**50.3236**
CPM	WPD	512	0.25	0.9277	0.9509	272.0035
FSDD	**MFCC**	**512**	**0.1**	**0.4663**	**0.8958**	**4.756**
FSDD	MFCC	512	0.25	0.7476	0.9263	148.6702
FSDD	WPD	1024	0.1	0.8262	1	8.7274
FSDD	WPD	512	0.1	0.85	1	9.8756
FSDD	WPD	512	0.25	0.95	1	11.4334

Table 3.1 contains the results of the experiment. This table does not include the best-achieved model for the CPM dataset, as it is introduced later in section 4.3.1. The *width* column specifies the width of dense NN layers. The column called *dropout* is the dropout rate in the dense layers. The metrics are defined as:

- $WER = \frac{(i_w+s_w+d_w)}{n_w}$, where n_w is the number of words in the reference text, s_w is the number of words substituted, d_w is the number of words deleted, and i_w is the number of words inserted to transform the output text into the reference text. The minimal possible transformation is chosen. Number of correct words c_w would be $c_w = n_w - (d_w + s_w)$.

- $CER = \frac{(i+s+d)}{n}$, where the variables are numbers of characters instead of words.
- *Loss* is an internal metric for measuring the performance of the neural network. DeepSpeech uses the CTC loss described in section 2.3.

3.6 Evaluation

The first thing that could be noticed during the experiments is that the WPD implementation is very slow and memory inefficient when compared to the Tensorflow MFCCs. The main difference is that the MFCC operations are very efficiently implemented (for both memory and time) in C++, compiled, and then executed from Python. This difference leads to high dense layers reductions in width that had to be made to train the network. Also, the batch sizes had to be reduced (removed for the CPM dataset), which caused substantial computation times.

When it comes to performance results, this WPD implementation performed very poorly in representing phoneme information. It tended to underfit or end up in local minimums without learning the internal dependencies of a feature vector and phoneme. MFCC features massively outperformed WPD in both datasets, no matter the configurations.

The first step that should be taken before further experiments with WPD and speech recognition are GPU optimized parallel implementations of those decompositions. There is a non-Tensorflow Python package called *PyWavelets* [44], a Tensorflow implementation of CWT called *tf-wavelets* [45], and a CWT Tensorflow implementation called *cwt-tensorflow* [46]. However, none of them fitted for this problem, nor did it seem like a significant improvement when it comes to optimization. When there is an efficient implementation, it would be interesting to try different types of wavelets, or to combine wavelets with another conventional approach.

Chapter 4

Realization

The applicational nature of this thesis makes this chapter the essential component. It aims to train an appropriate neural network model on any Czech spoken language dataset and use this trained model to transcribe different open speech data. The chosen model used in this work is the DeepSpeech project, described in section 3.1.

This thesis's practical side is implemented as a set of tools, scripts, and custom classes in a docker image environment tailored for the said task. The source code, scripts, and other outputs are being stored and versioned in a GitHub repository called *speech2text* [43]. It will often be referred to as a "repository."

The following sections will introduce the custom Docker environment used for this thesis with its Dockerfile. The reasons to use a Docker image are explained, followed by the description of the most critical installed packages and used tools with a few usage or data examples. The last few sections will describe the process, which leads to generating a new dataset for the Czech spoken language.

4.1 Docker Image

This project is designed to be used as an Nvidia Docker container (Docker [47], nvidia-docker [48]). Therefore a custom-made Dockerfile defining the image is located in the repository. Moreover, the repository also contains a step-by-step tutorial on how to build and run the Docker container successfully.

It is based on an Nvidia CUDA image with Ubuntu 18.04 as its operating system (OS). The reason for using a Docker image is to avoid complicated, and generally not recommended, installation of a specific combination of GPU drivers, a CUDA library, and a CuDNN library. These three components are dependent on each other and also on the libraries used in a project; Tensorflow in this case.

There are several packages and programs installed. From essential tools (e.g., Python 3, Git, nano, ...) to additional libraries to ensure all dependencies and requirements are met. Besides those system-wise packages, the image is built with a prepared Python virtual environment, where are other necessary Python packages, most of which are DeepSpeech related. These include a Jupyter server package.

Since this project is meant to be run in a remote Docker container, the Jupyter notebooks are one of the few means of having a user-friendly IDE (integrated development environment). The *speech2text* repository contains a script called *start_jupyter.sh*, which starts the Jupyter server with the necessary options. An essential part of getting the server to work on a local machine while running in a remote container is explained in section 4.1.1.

One thing, that the Dockerfile counts on, is to have the DeepSpeech repository cloned next to the *speech2text* repository. That is because the DeepSpeech repository is quite large (around 1.5 GB) and cloning the entire repository each time a user wants to re-build the image would be very time-consuming. With this said, we can go to the Makefile.

4.1.1 Makefile

The Makefile is a parametric script using the *make* program to execute Docker related commands. There are four main parameters, which are environment-dependent and are up to the user to configure before executing the commands. It includes:

- *GPU* where a user specifies which graphical units are supposed to be accessible in the container,
- *RAM_LIMIT* to be able to limit the RAM usage by the container,
- *I_NAME* which is the name of the image, and
- *C_NAME* which is the name of the container.

Two additional parameters specify mounted (shared) directories from the host machine to the container. That is, again, very user-specific, and no folder has to be necessarily mounted.

The Makefile supports several commands:

- *build*, which builds the image defined in the Dockerfile, potentially using cache to skip a few layers if their definition has not changed from the last time the image was built.
- *build-nocache* does the same as *build* except it does not use cache.
- *run* can be executed after a successful build. It uses the parameters mentioned above and runs the container using the *nvidia-docker* toolkit. It also exposes specific internal container ports to the host machine so that the user can access both the Jupyter server and Tensorboard from the local machine. Additionally, it mounts both host directories so they are accessible from within the container.
- *run-cpu* does the same as the *run* command except it does not expose the GPU units to the container; it uses regular *docker* command, and not the *nvidia-docker* one.
- *exec* runs a new bash in the running container, and the
- *attach* option attaches local inputs and outputs to the running container (useful on connection loss, for example).
- *default_arguments* prints out the values of the parameters.

There is a work-around for the Docker to work with the DeepSpeech repository during a build. Each time a build is executed, the Dockerfile is copied outside the repository to the parent directory. Therefore the build context includes the repository, and after each build (successful or unsuccessful), the file is removed.

Now, we should be able to build a Docker image and run a container with all necessary means to train a speech-to-text model.

4.2 Training Prerequisites

To start a model training, we need to meet a few prerequisites. The first one is to have preprocessed data with three CSV files which refer to training, evaluation, and test subsets of the data. The process to achieve that is described in section 3.3.

The second is to have a file containing an alphabet (plus a space character) for the language the model is going to be trained on. Such a file is located in the root of the repository and is called *alphabet_cz.txt*.

The third is to create a language model. This step is explained in the following section.

4.2.1 Creating a Language Model

A brief introduction to language models (LM) is in section 2.3.3. The first step to generate a language model is to gather data. The data should ideally be from the same domain as the speech audio we want to transcribe. Therefore, there is a Jupyter notebook in the repository (*cpm_lm_cralwer.ipynb* in the *notebooks* directory) containing an implementation of a web crawler for the Czech Parliament meetings (CPM) website, which downloads all available transcriptions. The process to gather the CPM data goes as:

1. Download a website with a list of all the Parliament meetings and extract links to the meetings,
2. from the individual meetings, gather links to all the "repositories" with compressed archives containing meetings transcriptions,
3. gather links to all the archive files, since each "repository" contains several archives, and
4. download and extract the files.

After gathering the data, it is passed to an algorithm that processes the data. First, it sentence-tokenizes the text a puts one sentence per line. Then it repeats the process done for the dataset labels, which lower-cases the text, removes punctuation, and excludes sentences with numeric characters.

With the text file in the described format, we are ready to generate the language model. We are going to use a program called *kenlm* to generate necessary files. First, we have to generate an ARPA file - a text file with a specific format. It contains so-called *n-grams*, which are probabilities of word sequences of length *n*. Specifying an order of 5 (which is the value used in this thesis) generates all *n*-grams up to 5-grams. It can be executed from "/opt/kenlm/build/bin/" directory as:

```
./lmplz --text [SENTENCES_FILE] \
```

```
        --arpa [OUTPUT_PATH] \
        --o [ORDER]
```

The next step is to build an LM binary from the generated ARPA file. That is done by running:

```
./build_binary [ARPA_FILE] \
               [OUTPUT_PATH]
```

from the same directory. The last thing required by DeepSpeech is to generate a *trie*. Trie is the tree structure described in section 2.3.3, and DeepSpeech has its own tool to create the trie from the LM binary. It is executed from "/opt/DeepSpeech/native_client_prebuilt" directory as:

```
./generate_trie [ALPHABET] \
                [LM_BINARY] \
                [TRIE]
```

With the trie generated, we have got everything required to start the model training.

4.3 Training a Model

For the model training, there is a script which helps with:

- a convenient way of setting training parameters,
- creation of directories for training file outputs – checkpoints, exports, summary, and logs,
- redirecting terminal output to a file,
- converting a trained model to a mmap-able model for inference, and
- saving training parameters next to the exported model in a text file.

The script *train_custom.sh* is located in the root of the repository. For the list of available DeepSpeech parameters and their meaning, refer to the DeepSpeech repository [37] or launch *DeepSpeech.py* with the *helpfull* flag as:

```
./DeepSpeech --helpfull
```

from the "/opt/DeepSpeech" directory.

To start a training, run the script in the background as:

```
./train_custom &
```

and watch the progress of the training by executing:

```
tail -f [LOG_FILENAME]
```

with the log filename reported by the script when executed.

Models are exported to a single directory and are named based on their performance. The name consists of three numbers: word error rate (WER), character error rate (CER), and the loss value. Those models can later be sorted, and it can be quickly determined, which model is the best by those mentioned metrics.

4.3.1 The Best Model for the CPM

So far, the best model trained on the CPM dataset achieved 12.66 % WER and 4.63 % CER. That can be boldly interpreted as one wrongly transcribed word in every eight words or one character mistake for every 21 characters. Those are pretty reasonable values to be used on the transcription of the final dataset.

The model was trained for 13 epochs, after which it triggered an early stop – that means the test error reached a point of convergence, and the model would tend to overfit after more epochs. These were the hyperparameters used:

- *n_hidden:* 2048
- *learning_rate:* 0.0001
- *dropout_rate:* 0.25
- *epochs:* 25
- *early_stop:* true
- *lm_alpha:* 0.75
- *lm_beta:* 1.85

Moreover, frequency, pitch, and speed augmentations were taken advantage of during the training. The training took approximately three and a half hours with batch sizes 24/48/48 for train/dev/test accordingly.

4.4 Transcribing a New Dataset

Now, we have a successfully trained and exported model, therefore, the means to create transcriptions for new speech audio data.

The Docker environment offers three options to run inference on an audio segment. The first is to run the pre-installed *bash* program *deepspeech* as:

```
deepspeech --model [MODEL] \
           --lm [LM_BINARY] \
           --trie [TRIE] \
           --audio [AUDIO_FILE]
```

Another option used for testing purposes is to use a custom script called *inference.sh*, which is used for the initial CPM dataset. A user has two options: to specify a recording ID or run inference on n random files from the CPM dataset. The script automatically picks the best model available in the model export directory and runs the inference. It outputs the original transcription alongside the inferred one for each of the files to compare.

The best way to transcribe multiple files is to use the *deepspeech* package programmatically. The package offers a reasonably straightforward API capable of loading a trained model and performing inference over it. Since the package has Python bindings, loading multiple directories containing recordings, transcribing them, and saving the transcriptions into a file is very comfortable.

Data Gathering

After consulting this section with my supervisor, the rest of the CPM recordings were decided to form the final dataset. That is around 200 GB of speech data in over 16 thousand mp3 files. Each one of the files is around 13 minutes long.

There is a web crawler called *audio_crawler.ipynb* for the task of data collection. The script goes through all years, months, and days with meetings on the CPM website and downloads all the individual recordings. Since the CPM site is limiting download speeds for individual requests, it is implemented as a parallel algorithm. Other than that, the process is quite similar to the language model web crawler from section 4.2.1.

Speech Audio Segmentation

The next step is to cut the long recordings into several-seconds-long segments. Preferably, the segments should be cut so that they do not begin nor end with speech fragment; i.e., the cuts have to be made when silence occurs in the recordings. That is where a voice activity detection (VAD) plays its role.

We are going to use *inaSpeechSegmenter* [49, 50] to differentiate speech from silence (and music). It is a pre-trained model that outputs CSV-like files in a pre-defined format, containing labeled time intervals. Labels are *speech*, *noEnergy* (silence), and *music*. The segmenter is executed from the command line as follows:

```
ina_speech_segmenter.py      -i INPUT [INPUT ...] \
                             -o OUTPUT_DIRECTORY \
                             -g false
```

This way, the segmenter uses (by default) the better VAD engine and does not bother with gender recognition. The input flag accepts either a list of files or a regular expression. Unfortunately, the expanded regular expression cannot be too long, or the program ends with an error—that led to writing a short script that repeatedly executed the program on subsets of the recordings. Another obstacle was that the program crashed when encountering some of the recordings for unknown reasons. It was always the same files, but there was, from a user side, nothing wrong with them. Nevertheless, the VAD was successful in the end.

The last remaining step in the data pre-processing is to cut the speech file according to the VAD model output. This step is implemented in the *cut_audio_mp3.ipynb* notebook. Its output is a directory with short speech audio segments ready to be transcribed. After this step, the dataset contained over 580000 speech segments of ranging length roughly from 1 to 70 seconds.

Final Transcriptions

For the transcription task, there is a notebook called *transcriber.ipynb*. The functions there programmatically use the *deepspeech* package to load the best model from the specified directory and run inference on a file. The output is a CSV file in the DeepSpeech format that contains filenames, file sizes, and the transcriptions. A sub-sample of the final generated dataset is in the *dataset_sample* directory in the repository. Figure 4.1 shows a few entries from the dataset CSV file. Apart from the mentioned functions, the notebook also

```
/opt/shared_data/cpm_wav_cut/2014091018081822_0020.wav,217004,sankcích jako takových
/opt/shared_data/cpm_wav_cut/2013112514181432_0054.wav,280364,pan poslanec david kasal
/opt/shared_data/cpm_wav_cut/2014091018081822_0003.wav,205484,pane poslanče máte slovo
/opt/shared_data/cpm_wav_cut/2014051416081622_0037.wav,230444,na veřejné finance živící
/opt/shared_data/cpm_wav_cut/2013121211281142_0021.wav,197804,kdo je proti tomuto návrhu
/opt/shared_data/cpm_wav_cut/2014012212081222_0016.wav,178604,tolik přidat se pana prezidenta
/opt/shared_data/cpm_wav_cut/2014092512281242_0048.wav,186284,prosím pane ministře máte slovo
/opt/shared_data/cpm_wav_cut/2014051320482102_0017.wav,282284,byla nezbytná k tomu aby zazněla
/opt/shared_data/cpm_wav_cut/2014090216581712_0063.wav,307244,který třeba kteří mohli být obeťou
/opt/shared_data/cpm_wav_cut/2014071712281242_0017.wav,387884,to že se vláda ani politici neodvážili
/opt/shared_data/cpm_wav_cut/2014051512281242_0014.wav,439724,ano pan místopředseda bartošek na slovo
/opt/shared_data/cpm_wav_cut/2014062009380952_0064.wav,332204,že nechce tento problém radikálně řešit
/opt/shared_data/cpm_wav_cut/2014021812481302_0018.wav,341708,ale dostávají stále horší veřejné služby
/opt/shared_data/cpm_wav_cut/2014071711081122_0056.wav,332204,být ve svých daňových úvahách odvážnější
```

Figure 4.1: Final dataset sample

contains a handy little script that loads the best model from a directory, picks a random recording from the new dataset, transcribes it, and allows a user to play the recording. At the same time, he or she can read the transcribed string and compare the results.

Conclusion

This thesis dealt with automatic speech recognition using recurrent neural networks. It analyzed the current state-of-the-art in the field both for RNN and ASR and backed up the theory with a mathematical background.

Based on the analysis, an experiment was conducted to compare speech recognition performance of two distinct feature extraction methods: Mel-frequency cepstral coefficients (MFCC) and wavelet packet decomposition (WPD). The results were unequivocally on the side of MFCCs, while the model with the WPD approach was unable to learn properly.

The goal of this thesis was to train an adequate NN model on a Czech spoken language dataset. The dataset of choice was the Czech Parliament meetings (CPM) dataset. DeepSpeech, an open-source speech recognition project, served as the neural network model.

After preprocessing and transforming the dataset into a suitable form, and creating a language model, the training and optimization could begin. The best model achieved 12.66 % WER and 4.63 % CER. Those values were more than enough to gather different data and perform a transcription over them. The second dataset is very similar to the initial one because it comes from the same domain - more Czech Parliament recording. This time, they were scraped straight from the Parliament web site. After the data was collected, it was preprocessed using a voice activity detection (VAD) model as a reference. It consists of over 580000 speech utterances of ranging length roughly from 1 up to 70 seconds. The last step was to transcribe the speech segments, and the dataset was complete.

As for future work and improvements, the model could always be more fine-tuned and better generalized by using more sophisticated augmentation methods. With every small bit of increased performance, the dataset would get of a much higher quality. When it comes to gathering suitable data for this or similar tasks, there are a few ideas, such as audiobooks, where the text version could serve as a reference. Unlicenced movies or series with subtitles could also be useful.

One fascinating project that is rising as of writing these lines is called the Mozilla Common Voice [51, 52]. It is an open-source project where people contribute their voice to a freely available dataset. Each recording is validated and, in the end, added to the speech pool. Right now, the English dataset consists of over 1400 hours of validated recordings from more than 62000 speakers. Unfortunately, the Czech language got just recently from the preparation and translation stage and, at the moment, has only 28 hours of recordings from 272 speakers. Once this project fully launches for us, the ASR will never be the same for Czech speakers.

Bibliography

[1] Haykin, S. *Neural Networks and Learning Machines (3rd Edition)*. Pearson Education India, 2010, ISBN 0131471392.

[2] Patel, N. Data Mining – Artificial Neural Networks. Online; accessed 25-February-2020. Available from: https://ocw.mit.edu/courses/sloan-school-of-management/15-062-data-mining-spring-2003/lecture-notes/NeuralNet2002.pdf

[3] Pal, S. K.; Mitra, S. Multilayer perceptron, fuzzy sets, classifiaction. 1992, iEEE TRANSACTIONS ON NEURAL NETWORKS, VOL. 3, NO. 5.

[4] Brilliant.org. Backpropagation. Online; accessed 10-March-2020. Available from: https://brilliant.org/wiki/backpropagation/

[5] Pascanu, R.; Mikolov, T.; et al. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, 2013, pp. 1310–1318.

[6] Wang, C.-F. The Vanishing Gradient Problem. Online; accessed 17-March-2020. Available from: https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484

[7] Shorten, C. Introduction to ResNets. Online; accessed 17-March-2020. Available from: https://towardsdatascience.com/introduction-to-resnets-c0a830a288a4

[8] Goodfellow, I.; Bengio, Y.; et al. *Deep Learning, Part II, Chapter 10*. MIT Press, 2016, http://www.deeplearningbook.org.

[9] Gers, F. A.; Schmidhuber, J.; et al. Learning to forget: Continual prediction with LSTM. 1999.

[10] Zhou, G.-B.; Wu, J.; et al. Minimal gated unit for recurrent neural networks. *International Journal of Automation and Computing*, volume 13, no. 3, 2016: pp. 226–234.

[11] Chung, J.; Gulcehre, C.; et al. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[12] Jozefowicz, R.; Zaremba, W.; et al. An empirical exploration of recurrent network architectures. In *International conference on machine learning*, 2015, pp. 2342–2350.

[13] Greff, K.; Srivastava, R. K.; et al. LSTM: A search space odyssey. *IEEE transactions on neural networks and learning systems*, volume 28, no. 10, 2016: pp. 2222–2232.

[14] Shewalkar, A.; Nyavanandi, D.; et al. Performance Evaluation of Deep Neural Networks Applied to Speech Recognition: RNN, LSTM and GRU. *Journal of Artificial Intelligence and Soft Computing Research*, volume 9, 10 2019: pp. 235–245, doi:10.2478/jaiscr-2019-0006.

[15] Khandelwal, S.; Lecouteux, B.; et al. Comparing GRU and LSTM for automatic speech recognition. 2016, available from: `https://hal.archives-ouvertes.fr/hal-01633254/document`.

[16] Yu, D.; Deng, L. *AUTOMATIC SPEECH RECOGNITION, A Deep Learning Approach*. Springer, 2016, ISBN 978-1-4471-5779-3.

[17] Schaller, R. R. Moore's law: past, present and future. *IEEE spectrum*, volume 34, no. 6, 1997: pp. 52–59.

[18] Dictionary.com. Definition of Infotainment. Online; accessed 25-March-2020. Available from: `https://www.dictionary.com/browse/infotainment`

[19] Rabiner, L. R.; Schafer, R. W.; et al. Introduction to digital speech processing. *Foundations and Trends® in Signal Processing*, volume 1, no. 1–2, 2007: pp. 1–194.

[20] Hoy, M. B. Alexa, Siri, Cortana, and more: an introduction to voice assistants. *Medical reference services quarterly*, volume 37, no. 1, 2018: pp. 81–88.

[21] Davis, S.; Mermelstein, P. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *IEEE transactions on acoustics, speech, and signal processing*, volume 28, no. 4, 1980: pp. 357–366.

[22] Burrus, C.; Gopinath, R.; et al. Introduction to Wavelets and Wavelet Transform—A Primer. *Recherche*, volume 67, 01 1998.

[23] Kutz, J. N. *Data-driven modeling & scientific computation: methods for complex systems & big data, PART III, 13.: Time-Frequency Analysis: Fourier Transform and Wavelets*. Oxford University Press, 2013, ISBN 978-0199660346.

[24] Kutz, N. Time Frequency Analysis & Wavelets. Available from: `https://www.youtube.com/watch?v=ViZYXxuxUKA`

[25] Pearson Education, Inc. *Chapter 9, Automatic Speech Recognition*. Pearson Prentice Hall, 2008, available from: `http://www.cs.columbia.edu/~julia/courses/CS6998-2019/[09]AutomaticSpeechRecognition.pdf`.

[26] Fayek, H. Speech Processing for Machine Learning: Filter banks, Mel-Frequency Cepstral Coefficients (MFCCs) and What's In-Between. 2016, online; accessed 6-April-2020. Available from: `https://haythamfayek.com/2016/04/21/speech-processing-for-machine-learning.html`

[27] Gowdy, J. N.; Tufekci, Z. Mel-scaled discrete wavelet coefficients for speech recognition. In *2000 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No. 00CH37100)*, volume 3, IEEE, 2000, pp. 1351–1354.

[28] Malik, S.; Afsar, F. A. Wavelet transform based automatic speaker recognition. In *2009 IEEE 13th International Multitopic Conference*, IEEE, 2009, pp. 1–4.

[29] Král, P. Discrete Wavelet Transform for automatic speaker recognition. In *2010 3rd International Congress on Image and Signal Processing*, volume 7, IEEE, 2010, pp. 3514–3518.

[30] Hidayat, S. Speech recognition of KV-patterned Indonesian syllable using MFCC, wavelet and HMM. *Kursor*, volume 8, no. 2, 2016: pp. 67–78.

[31] Muo, U.; Madamedon, M.; et al. Wavelet packet analysis and empirical mode decomposition for the fault diagnosis of reciprocating compressors. 09 2017, pp. 1–6, doi:10.23919/IConAC.2017.8082065.

[32] Graves, A. Supervised Sequence Labelling with Recurrent Neural Networks [Ph. D. dissertation]. *Technical University of Munich, Germany*, 2008.

[33] Scheidl, H. An Intuitive Explanation of Connectionist Temporal Classification. Online; accessed 18-May-2020. Available from: https://towardsdatascience.com/intuitively-understanding-connectionist-temporal-classification-3797e43a86c

[34] Bahl, L. R.; Brown, P. F.; et al. A tree-based statistical language model for natural language speech recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, volume 37, no. 7, 1989: pp. 1001–1008.

[35] Abadi, M.; Agarwal, A.; et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. 2015. Available from: https://www.tensorflow.org/

[36] Kaplan, I. The Wavelet Packet Transform. Online; accessed 19-May-2020. Available from: http://bearcave.com/misl/misl_tech/wavelets/packet/index.html

[37] Mozilla Corporation. DeepSpeech. https://github.com/mozilla/DeepSpeech, 2019.

[38] Hannun, A.; Case, C.; et al. Deep Speech: Scaling up end-to-end speech recognition. 2014, 1412.5567.

[39] Mozilla Corporation. DeepSpeech – Introduction. Available from: https://deepspeech.readthedocs.io/en/v0.6.0/DeepSpeech.html

[40] https://stackoverflow.com/users/1090562/salvador-dali, S. D. What is the meaning of the word logits in TensorFlow? Stack Overflow, url: https://stackoverflow.com/a/43577384/8281969 (version 2018-10-02).

[41] Pražák, A.; Šmídl, L. Czech Parliament Meetings. 2012, LINDAT/CLARIAH-CZ digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University. Available from: http://hdl.handle.net/11858/00-097C-0000-0005-CF9C-4

[42] Zohar Jackson (Jakobovski). free-spoken-digit-dataset. https://github.com/Jakobovski/free-spoken-digit-dataset, 2019.

[43] OpenDataLab/Richard Werner. speech2text. https://github.com/opendatalabcz/speech2text, 2020.

[44] Lee, G.; Gommers, R.; et al. PyWavelets: A Python package for wavelet analysis. *Journal of Open Source Software*, volume 4, no. 36, 2019: p. 1237, doi:10.21105/joss.01237. Available from: https://doi.org/10.21105/joss.01237

[45] UiO-CS. tf-wavelets. https://uio-cs.github.io/tf-wavelets/, 2018.

[46] Nick Geoca (nickgeoca). cwt-tensorflow. https://github.com/nickgeoca/cwt-tensorflow, 2018.

[47] Docker Inc. Docker Documentation. Online. Available from: https://docs.docker.com/

[48] NVIDIA Corporation. nvidia-docker. https://github.com/NVIDIA/nvidia-docker, 2020.

[49] Doukhan, D.; Carrive, J.; et al. An Open-Source Speaker Gender Detection Framework for Monitoring Gender Equality. In *Acoustics Speech and Signal Processing (ICASSP), 2018 IEEE International Conference on*, IEEE, 2018.

[50] Doukhan, D.; Lechapt, E.; et al. INA'S MIREX 2018 MUSIC AND SPEECH DETECTION SYSTEM. In *Music Information Retrieval Evaluation eXchange (MIREX 2018)*, 2018.

[51] Ardila, R.; Branson, M.; et al. Common Voice: A Massively-Multilingual Speech Corpus. 2019, 1912.06670.

[52] Mozilla. Common Voice. Online. Available from: https://voice.mozilla.org/en

Appendix A

Acronyms

ANN Artificial neural network

ASR Automatic speech recognition

CER Character error rate

CPM Czech Parliament meetings

CPU Central processing unit

CSV Comma-separated values

CTC Connectionist temporal classification

CUDA Compute unified device architecture

CuDNN The NVIDIA CUDA® Deep Neural Network library

CWT Continuous wavelet transform

DCT Discrete cosine transform

DFT Discrete Fourier transform

DNN Deep neural network

DWT Discrete wavelet transform

FFT Fast Fourier transform

FSDD Free spoken digit dataset

GMM Gaussian mixture model

GPS Global positioning system

GPU Graphics processing unit

GRU Gated recurrent unit

IDE Integrated development environment

LM Language model

LPC Linear predictive coding

LPCEPSTRA Linear prediction cepstral coefficients

LPREFC Linear prediction reflection coefficients

LSTM Long-short-term memory (network)

ML Machine learning

MLP Multilayer perceptron

MFCC Mel-frequency cepstral coefficients

MFDWC Mel-frequency discrete wavelet coefficients

MULT Multi-resolution (features)

NN Neural network

OS Operating system

PLPC Perceptual linear prediction coefficients

ReLU Rectified linear unit

ResNet Residual network

RNN Recurrent neural network

STFT Short-time Fourier transform

SUB Subband-based (features)

VAD Voice activity detection

WER Word error rate

WPT Wavelet packet transform

XML Extensible markup language

Appendix B

Contents of the enclosed medium

- `dataset_sample` ..... a directory containing the sample of the dataset
- `notebooks` ..... a directory containing data preprocessing tools
- `thesis` ..... the thesis directory
  - `DiplomaThesis.pdf` ..... the exported thesis in a pdf format
  - `DiplomaThesis.zip` ..... the thesis source code
- `tools` ..... a directory containing scripts and tools used in the thesis
  - `speech2text` ..... a directory containing source codes
  - `model` ..... a directory containing the NN model