Feature Denoising using CNNs for Noisy Speech Recognition

Jens Fröschel

A thesis submitted for the degree of
Master of Science

Primary Supervisor:	Prof. Dr. Stefan Conrad
Secondary Supervisor:	Prof. Dr. Michael Schöttner
Advisor:	Michael Stadtschnitzer

September 2017

Erklärung zur Eigenständigkeit

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 15.09.2017

Jens Fröschel

Contents

Abstract

In modern days automatic speech recognition (ASR) systems rise in popularity especially in smartphones and smart home devices. If those ASR systems were to reach the level of human hearing, they could for example be used to remotely control intelligent devices or create live subtitles on television. Those systems would, among other things, vastly increase the living standards of deaf people and revolutionize the human-computer interaction.

One of the biggest problems of ASR are background noises like car sounds, conversations or wind. With the exclusion of these noises state-of-the-art ASR systems are already nearing the proficiency of human hearing. A common approach to handle these noises are speech enhancement (SE) systems.

In this thesis we examined a speech denoising system based on Convolutional Neural Networks (CNN). CNNs were already successfully used for speech recognition, SE and image denoising in previous studies. Based on these results using CNNs as a speech denoising system stands to reason.

The goal of this thesis was to create a CNN based speech enhancer to be used in the Fraunhofer speech recognition pipeline. The network presented consisted solely of convolutional layers and mapped noisy filter bank features onto clean filter bank features. As foundation the speech recognizer from the Eesen toolkit of the Wall Street Journal (WSJ) example was used.

In our experiments we found out, that the CNN denoising network can decrease the word error rate (WER) of a speech recognition system up to more than an absolute of 20% in an environment with a moderate signal-to-noise ratio (SNR). At the same time the WER of speech recorded on high SNRs only increased by one percent. Additionally it was shown that the denoising system generalizes onto multiple noise types and onto real world data.

The results of our studies showed, that denoising audio using a CNN on the feature level is possible and can improve state-of-the-art speech recognition systems significantly for noisy environments while at the same time only slightly decreasing the performance for clean speech.

1 Introduction

1.1 Motivation

Automatic Speech Recognition (ASR) is one of the most sophisticated tasks in artificial intelligence and computer science. The process of simulating human hearing and transforming spoken sentences into written text is far from perfect up till today. Yet those systems are already influencing our everyday life. They are for example used by smart assistants in our smartphones, computers or smart home devices. Other important tasks for ASR is the remote control of navigation systems or intelligent cars and the creation of video subtitles for online videos or live television. The last example is especially important for deaf people.

Even though ASR systems are so important and omnipresent, they still got many flaws. One of their major problems is noise. Audio recorded outside a quiet room contains background noises like car sounds, conversations, machines working, wind and many more. Most ASR systems were not setup to deal with those background noises. It is a big problem if a navigation system fails to understand a location because of car noises, a broadcast stenocaptioning creates faulty results because of wind noise or a smartphone does not understand the user because he is sitting in a crowded cafeteria.

Recently SE systems based on Deep Neural Networks (DNN) became the state-of-the-art. For example Xu et al. [1] proposed a DNN based speech enhancer in 2014 showing significant improvements in comparison to the logarithmic minimum mean square error approach [2]. And Du et al. [3] won the 4^{th} CHiME speech separation and recognition challenge [4] using deep Convolutional Neural Networks (CNN). There exist many more examples of DNN for SE. Especially CNNs create decent results in the process of speech recognition [5], image denoising [6, 7], robust speech recognition [8] and SE [9].

Our work built up on those previous studies and tried to further enhance the results of state-of-the-art ASR systems. A CNN was used for noise reduction of speech on the feature level. The neural network consisted solely of convolutional layers learning to map artificially noised audio features onto clean features. Our experiments were to show that denoising using CNNs on the feature level is possible and promising for future research.

1.2 Content

In this thesis we start in Chapter 2, by explaining all theoretical background knowledge needed to understand our field of research. We will introduce the foundation of digital audio processing, ASR and SE in Section 2.1. And Artificial neural networks, DNNs and CNNs will be theorized in Section 2.2. Afterwards in Section 2.3 the important toolkits and datasets used in our thesis will be introduced.

In Chapter 3 our experiments will be provided. This chapter starts in Section 3.1 by giving an extensive overview on previous studies in the field of state-of-the-art SE and DNN. Our experiments included the development of a SE testing environment and the set up of all required datasets explained in Section 3.2. That environment was build on the Eesen[10] speech recognition toolkit and as datasets the WSJ corpora [11, 12] were used. As feature type we used filter bank coefficients. The noisy audio files were created by mixing a given clean audio file with another given noise file and as noise files we used the CHiME-4 challenge [4] noises. Finally the mixing was done on different Speech Noise Ratios (SNR). At this point all preparations for the experiments were concluded.
In the following Section 3.3 the setup of our denoising system, optimization and evaluation will be introduced. For the experiments we used a multi-layerd CNN consisting only of convolutional layers plus activation functions. The neural networks were implemented in Keras using Tensorflow [13] as back end. The training was realised by calculating features for noisy and clean speech and teaching the network to map the noisy features onto the clean ones. The ASR system was the standard system of Eesen having only the denoising network added. In the experiments we let our speech enhancer denoise noisy features and used the Eesen ASR system for speech recognition. As metric for the results we used the Word Error Rate (WER).
In Section 3.4 the hyperparameter optimization is provided. The optimization was done by training different CNNs, using them as denoising system and evaluating the denoised speech features using the speech recognizer. This process was done for the same validation dataset on different SNRs and the WER for each SNR level was determined. As the optimization winner we used the CNN with the lowest average WER. In the process of finding the

best network possible for our experimental setup we optimized the number of hidden layers, the batch size, the number of convolutional kernels in the hidden layers and the optimizer function.
In section 3.5 the experiments evaluation is provided. The evaluation was done comparing four different ASR setups. The clean speech recognizer, the CNN denoising system, a speech recognizer trained on noised speech, and an ASR system trained on the denoised feature vectors were compared. The WER was used as metric comparing the four systems on different SNRs.
In Section 3.6 we generalized the experiments. In the generalization process the optimized denoising CNN was used on four types of noise instead of one, too test if the denoising system generalizes onto different types of noise. Afterwards experiments on real world data were conducted to test if the denoising system generalizes onto real world noise.

Chapter 4 concludes the final results of our thesis. The chapter gives a brief overview on the achievements of this thesis and provides some ideas for future studies. The results of the experiments showed that the CNN denoising system created decent improvements on the intelligibility of speech noised on low and moderate SNRs. At the same time the system generalizes onto multiple noise types and real world data.

2 Theoretical Background

In this chapter the theoretical background knowledge needed to understand the findings in our thesis is given. In Section 2.1 we will explain everything concerning ASR. Section 2.2 is about artificial neural networks. And the last Section 2.3 contains all information needed to understand our working environment.

2.1 Speech Recognition

In this Section we are going to explain everything concerning ASR. First we will introduce the conversion of an analog audio signal into a digital audio signal in Subsection 2.1.1. Afterwards we will explain the method to calculate filter bank feature vectors in Subsection 2.1.2. Next a brief introduction on the process of speech recognition will be provided in Subsection 2.1.3 and directly afterwards on robust speech recognition in Subsection 2.1.4. At the end we will give an introduction to the speech noise ratio (SNR) in Subsection 2.1.5 and WER in Subsection 2.1.6.

2.1.1 Analog-to-Digital Converter

Sound as a real world signal is analogue. To be able to manipulate an audio signal on a computer it must first be converted into a digital signal. The conversion is done by an Analogue-to-Digital Converter (ADC).
An analogue audio signal is a smooth and continues sound signal. It is composed of multiple overlapping sinusoid functions with different amplitudes and frequencies. A digital audio signal is a sequence of discrete amplitudes each given to a discrete time. The conversion of an analogue audio signal to a digital one involves sampling and quantization.

Sampling reduces a continues time signal into a discrete time signal by measuring the value of the continues function f_s times per second. f_s in hertz (Hz) is the sample rate. Since sampling is a reduction, parts of the original information will be lost in the process. As shown by the Nyquist-Shannon sampling theorem [14], with a sampling rate of f_s a signal up to a frequency of $\frac{1}{2}f_s$ can be sampled without information loss. Meaning the original continues signal can be reconstructed. A sample rate of $f_s = 16000$ Hz was used

throughout this thesis as the sample rate of the given audio data.

Quantization in digital signal processing is a function, mapping an infinitely large number of input values onto a finite set of values. In audio processing the quantization is applied onto the frequency range of the audio signal. The quantization is done by dividing the input value range into a finite number of contiguous intervals. The discrete time signal gets mapped onto those intervals. The precision of the mapping and the interval size depends on the number of bits used to represent the input signal, called bit depth. The trainings data used in our project has a bit depth of 16 bits per sample.

2.1.2 Feature Extraction

There are various different types of features used in state-of-the-art speech recognition (for example Mel-frequency cepstral coefficients (MFCC), filter bank coefficients, etc.). In this work we use filter bank coefficients [15] with first and second order deltas.

Given an audio signal $x(n)$ the filter bank coefficients are calculated in three steps visualized in Figure 1. First of all the signal $x(n)$ is split into multiple time frames $x_0(n)$, $x_1(n)$, ..., $x_{k-1}(n)$. The distance between two frames is called hopsize and the frame size is called window length. In our setup the frames have a hopsize of 10 ms (160 samples) and a window length of 25 ms (400 samples). Therefore the frames overlap with each other. To prevent the leakage effect [16], each sample in a frame was multiplied with a weight w_i. The weights are calculated for each frame by a windowing function $w(n)$, $0 \leq n \leq 399$. In this thesis the hamming windowing function given in Equation 1 was used.

$$w(n) = 0.54 - 0.46 \cdot cos(\frac{2 \cdot \pi \cdot n}{399}) \tag{1}$$

Next the magnitude spectrum of each frame is calculated by using the fast Fourier transformation (FFT) as fast implementation of the discrete Fourier transformation (DFT). A DFT approximates the original signal by a linear combination of sinusoidal functions in the form of complex exponentials. The calculated functions are sine waves with given frequencies, amplitudes and phases. For a given frequency k and a signal $x(n)$ at time n with length N the functions phase and amplitude $|X_k(n)|$ are encode in a complex number

X_k which is calculated by the DFT Equation 2. The DFT calculation by Equation 2 has a runtime of $\mathcal{O}(N^2)$. The FFT is a method used to calculate the DFT in $\mathcal{O}(Nlog(N))$.

$$X_k = \sum_{n=0}^{N-1} x(n) \cdot \mathrm{e}^{-2i\pi kn/N} \tag{2}$$

Finally a triangular logarithmic filter bank maps the magnitude spectrum into filter bank coefficients $c_i(n)$. A filter bank is a set of bandpass filters which splits the continues frequency range into a set of frequency bands and calculates an amplitude value for each band. The number of frequency bands is m. The amplitude for a band is calculated as a weighted average of the samples inside the band. For a triangular filter bank the filter's center samples weight more than the corners.

Figure 1: Flow chart for the filter bank calculation

As mentioned before we used filter bank features with first and second order deltas. This means we combine the original filter bank features $c_i = (c_{i,0}, c_{i,1}, ..., c_{i,m-1})$ with its delta vector $c'_i = (c'_{i,0}, c'_{i,1}, ..., c'_{i,m-1})$ and the second order delta vector $c''_i = (c''_{i,0}, c''_{i,1}, ..., c''_{i,m-1})$. This way the number of features triples in comparison to standard filter bank features. The new feature vectors got the form $c_i = (c_{i,0}, ..., c_{i,m-1}, c'_{i,0}, ..., c'_{i,m-1}, c''_{i,0}, ..., c''_{i,m-1})$. The first and second order deltas of each feature $c_{i,j}$ are calculated as a weighted linear combination of $c_{i-2,j}$, ..., $c_{i+2,j}$. The deltas approximate the first and second order derivatives of the signal.

In the experiments the deltas were only used in the ASR system and not in the denoising system. This means the deltas were calculated after the denoising was complete and before the speech recognition.

2.1.3 Clean Speech Recognition

In the following we are going to explain the classic phoneme-based speech recognition pipeline visualized in Figure 2. The standard ASR algorithm consists of four main components.

1. Feature calculation
2. Acoustic model
3. Lexicon
4. Language model

The first component is the calculation of the feature vectors described in Subsection 2.1.2. Additionally an acoustic model is used to transform feature vectors into phoneme probabilities. Phonemes are sound units used to distinguish words. A lexicon is used to map phonemes onto words and a language model combines words into syntactic correct sentences. The acoustic model, lexicon and language model combined calculate the likeliest written sentence represented by the feature vectors.
In this work the Eesen [10] speech recognition toolkit was used as ASR system. Eesen uses a recurrent neural network (RNN) based approach for the acoustic model. The model is implemented as a long short-term memory (LSTM) [17] and trained using a variation of connectionist temporal classifications (CTC) [18]. In the case of Eesen the lexicon and the language model are both implemented as weighted finite state transducers (WFST) [19]. Under the usage of OpenFST [20] the two WFSTs were fused into a single search graph. The graph receives phonemes as input and calculates sentences as output. In this thesis we will refer to the trained acoustic model in combination with the lexicon and language model as ASR decoder.

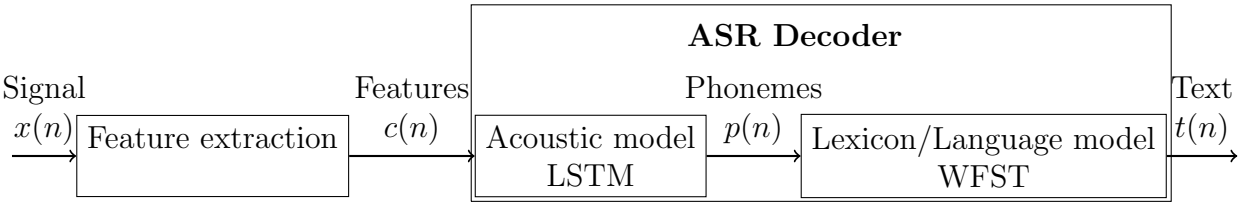

Figure 2: Flow chart for phoneme based speech recognition

2.1.4 Robust Speech Recognition

One of the major challenges in speech recognition are background noises. The field of ASR on noise influenced speech is called robust speech recognition. There exist different approaches for robust speech recognition using DNNs. One common approach is to train an automatic speech recognizer on noised trainings data. This way the recognizer will learn from start to ignore all kind of background sounds.
Another approach is to train a speech recognizer with a standard clean dataset and add a denoising algorithm into the speech recognition pipeline. This approach has some advantages important for us in comparison to the first approach. One advantage is that the denoising system can be activated or deactivated as needed. This is an important feature because speech recognizers solely trained on clean data give better results for clean speech recognition in comparison to systems developed for noisy ASR. This means a system build for robust speech recognition usually provides slightly worse results on clean data. If the denoising algorithm can be deactivated, a company worker or even an algorithm can decide whether the denoising system is needed or not. A second advantage of this approach is the existing ASR system must not be retrained or modified and is independent of the noise filter. This leads to two advantages, on the one hand experiments can be accomplished easier and on the other hand filter for different noises can be trained and applied as needed. Because of those reasons we decided to use the second approach of denoising input data, before applying a preexisting speech recognition system.

The input data can be denoised in two common ways. The first approach is to denoise the data stream on the signal level. This means the denoising algorithm receives a noisy digital audio signal as input and has to map that signal onto either a clean digital audio signal or onto clean feature vectors. The second approach is to denoise on the feature level. This means the algorithm receives noisy feature vectors as input and has to map those onto clean feature vectors. We decided to use the second approach because it was easier to implement and as a previous feasibility study [21] showed the denoising approach on filter bank features is possible. Therefore the extracted filter bank coefficients $c_i(n)$ get denoised to $\hat{c}_i(n)$ using a CNN. And those denoised filter bank coefficients are used as input for the ASR Decoder as visualized in Figure 3.

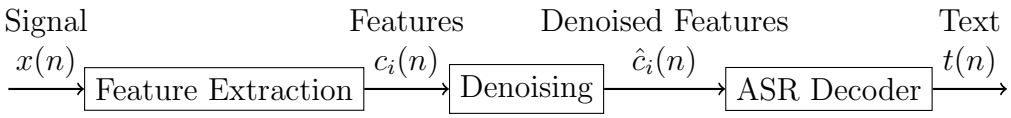

Figure 3: Flow chart for robust speech recognition

2.1.5 Signal-to-Noise Ratio

The Signal-to-Noise Ratio (SNR) is a measure to compare the level of a signal to the level of background noises. In the context of speech recognition the SNR is the ratio of the power of the speech signal s_{pow} in comparison the power of the noise signal b_{pow} in decibel (dB).

$$SNR = 10 \cdot log_{10}(\frac{s_{pow}}{b_{pow}}) \tag{3}$$

The power of a digital signal $x(n)$ with length N is the average square of its samples and can be calculated by Equation 4.

$$P = \frac{1}{N} \sum_{i=0}^{N-1} x(i)^2 \tag{4}$$

For the calculation of the SNR a clear distinction of the clean and the noisy signal is needed. There exist two common way to achieve a distinction between those two signals. The first approach is to divide the signal into a clean and a noisy component. This approach is very difficult and if we managed to properly divide a speech and a noise signal, there would be no need for a denoising algorithm. The more realistic approach is to start with a clean speech signal $s(n)$ and a noise signal $b(n)$. Those two signals get mixed by Equation 5 and create an artificial noisy speech signal $x(n)$.

$$x(n) = s(n) + b(n) \tag{5}$$

The SNR is an important measure to estimate the degree of noisiness in a signal. For example, a SNR of 30 dB means there is nearly no background noise, a SNR of 0 dB means the speech and noise power are the same and a SNR of -6 dB means that the background noise is stronger than the speech. One important fact is, that decreasing the SNR by a step of 6 dB means the noise volume is roughly doubled.

2.1.6 Word Error Rate

The WER is a metric for the performance of a speech recognition system. It is the percentage of wrongly recognized words and it is calculated using the Levenstein distance. Let $V = (v_0, v_1, ..., v_{n-1})$ and $W = (w_0, w_1, ..., w_{m-1})$, $n, m \in \mathbb{N}$ be two sentences with v_i, w_j being words for $0 \leq i \leq n-1$,$0 \leq j \leq m-1$. The Levenstein distance $lev_{V,W} = lev_{V,W}(n, m)$ of the sentences V and W is calculated recursively. The recursions base is given in Equation 6 and the recursion step in Equation 7.

$$lev_{V,W}(0, j) = j \text{ and } lev_{V,W}(i, 0) = i \tag{6}$$

$$lev_{V,W}(i, j) = min \begin{cases} lev_{V,W}(i-1, j) \\ lev_{V,W}(i, j-1) + 1 \\ lev_{V,W}(i-1, j-1) + I_{v_i \neq w_j} \end{cases} \tag{7}$$

$I_{v_i \neq w_j}$ is the indicator function being 0 if v_i equals w_j and 1 otherwise.

In simple terms the Levenstein distance tries to convert the original sentence V into the predicted sentence W by adding, subtracting or replacing words. The recursion finds the conversion path with the least operations. The Levenstein distance is the minimum number of operations needed to convert V to W.

To receive the WER the Levenstein distance is divided by the number of words in the original sentence.

$$WER = \frac{lev_{V,W}}{n} \tag{8}$$

Therefore a WER of 0 % means that V and W are identical and the spoken sentence was recognised correctly. On the contrary a WER above 100 % suggests that more words were incorrectly recognized than actually existed in the original sentence ($\Rightarrow m > n$). This can happen on data with strong noises. If for example the spoken sentence consists of 20 words and the speech recognizer recognizes 24 words and none of the recognized words is part of the original sentence, the WER is 120 %.

2.2 Artificial Neural Networks

In this Section we are going to explain everything concerning artificial neural networks. We will start off in Subsection 2.2.1 by introducing artificial neurons as fundamental components of neural networks. Afterwards in Subsection 2.2.2 we explain the way to combine single neurons into a neural network. Next the convolutional kernel as foundation for CNNs will be explained in Subsection 2.2.3. And directly afterwards in Subsection 2.2.4 we explain the functionality of CNNs. In 2.2.5 we explain the training of neural networks using the backpropagation algorithm and at the end in Subsection 2.2.6 different optimizer functions will be presented.

2.2.1 Artificial Neuron

An artificial neuron is a mathematical construct used as a key component for artificial neural networks. An artificial neuron consists of three parts. At the beginning the neuron receives an input in form of a vector. Than the neuron weights every value of the input, deciding which input part is important or unimportant for the neuron. Afterwards one output value is calculated, given the weighted input.

Mathematically this means n input values $x_0, x_1, ..., x_{n-1}$ are given and each input is multiplied by a weight $w_0, w_1, ..., w_{n-1}$. The results of the multiplications are summed and an activation function $f_a(x)$ is applied to receive the neurons output. Therefore a neuron's output is calculated by Equation 9.

$$y = f_a(\sum_{i=0}^{n-1} x_i \cdot w_i) \tag{9}$$

If the input $x = (x_0, x_1, ..., x_{n-1})$ and the weights $w = (w_0, w_1, ..., w_{n-1})$ are given as vectors the mathematical rule can be shortened to vector multiplication.

$$y = f_a(x^T w) \tag{10}$$

An example visualization for a single artificial neuron is given in Figure 4.

The activation function of a neuron is a real-valued function adjusting the output of the networks sum. Activation functions are an essential part of

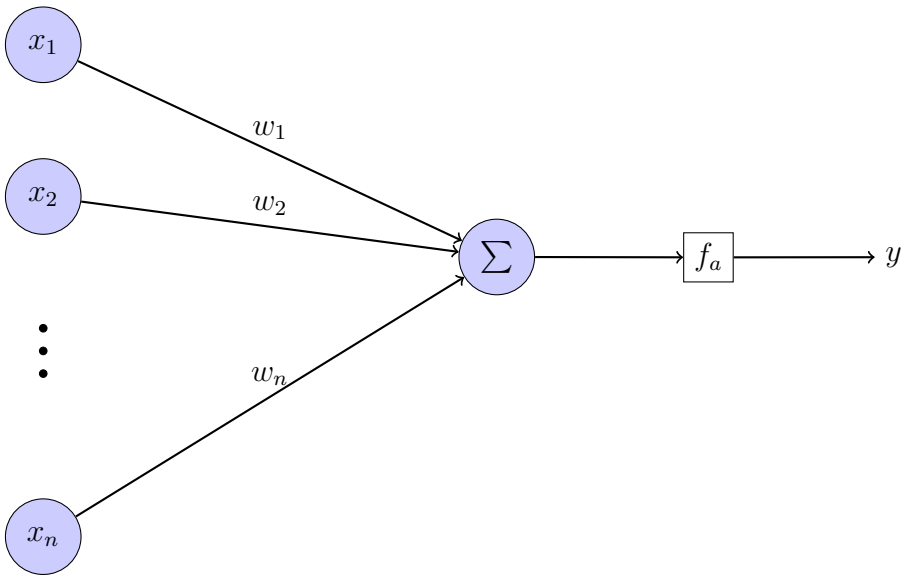

Figure 4: An artificial neuron including its activation function

DNNs. They are needed to add non-linearity into the network. Every DNN without activation functions is a linear regression. This means the network can only approximate linear functions. The problem hereby is that most modern problems can not be represented by a linear function.
On the other hand if the network got at least one hidden layer and an non linear activation function, the network is a universal function approximator[22]. This means the network can approximate any continues function, under the assumption of containing enough neurons. There exist a lot of different activation functions. One commonality of all functions is that their differential can be calculate efficiently, which is needed for the network training described in Subsection 2.2.5. In the following we will list the most common activation functions implemented in Keras[23].

- Exponential linear unit (elu) $f_a(x) = \begin{cases} exp(x) - 1 & \text{, if } x < 0 \\ x & \text{, else} \end{cases}$
- Softplus $f_a(x) = log(1 + e^x)$
- Softsign $f_a(x) = \frac{x}{|x|+1}$
- Recitified linear unit (relu) $f_a(x) = max(0, x)$

- Tanh $f_a(x) = tanh(x)$
- Sigmoid $f_a(x) = \frac{1}{1+e^{-x}}$
- Hard sigmoid $f_a(x) = max(0, min(x \cdot 0.2 + 0.5, 1))$
- Linear function $f_a(x) = x$
- Softmax $f_a(v) = \frac{e^v}{sum(e^v)}$

The softmax function is special because every neuron output value x is influenced by the other output values calculated on the same layer. The layer outputs are represented as vector v. Therefore the softmax function of a single value $x \in v$ is given as $f_a(x, v) = \frac{e^x}{sum(e^v)}$.

2.2.2 Artificial Neural Network

An artificial neural network is a combination of multiple artificial neurons into so called layers and connecting those into a network. Neural networks consist of three important types of layers. At the beginning of a network there is an input layer. As the name suggests this layer represents the input of the neural network. If the network has a 40 dimensional input, this layer contains 40 neurons, each neuron representing one input value. The input neurons each receive a single value as input and distribute the value into the first hidden layer. Therefore this layer neither got a sum nor an activation function.

After the input layer comes the hidden layers. Those are the actual thinking part of the network. Nowadays most neural networks are DNNs, which means they contain multiple hidden layers. Each hidden layer receives an input vector containing output values of the layer before. The size of the input vector depends on the number of neurons in the previous layer and also on the layer type. Some common types of neural layers will be explained later on.

The last layer in the neural network is the so called output layer. This layer is the same as a hidden layer with the peculiarity of its neuron count determining the size of the networks output vector. Therefore each output neuron receives multiple inputs from the previous layer, sums them up, adds an activation function and outputs a single value into the output vector. The input of the output layer also depends on the layer type.

For now we will give a short introduction to the most important layer types in state-of-the-art SE.

- Dense layer: Every neuron in a dense layer takes every output value of the previous layer as input. This means every neuron in a layer, is connected to every neuron in the subsequent layer. A network only consisting of dense layers is called fully connected neural network.
- Convolutional layer: A convolutional layer only takes a regionally connected part of the last layers output values as input. Also each convolutional layer consists of so called kernels each receiving the same input and being trained in parallel. For a detailed explanation see Subsection 2.2.4.
- Recurrent layer: Recurrent layers do not only receive the previous layers output as input but also part of their own output. This way the layers output also depends on previous outputs creating an artificial memory.

Figure 5 shows a fully connected neural network containing the input layer x, two hidden layers a and b and the output layer y. The input is n-dimensional and the output is m-dimensional. The first hidden layer a consists of i many neurons and the second hidden layer b consists of j many neurons.

2.2.3 Convolutional Kernel

In this section we are only going to explain two dimensional convolutions using three dimensional convolutional kernels. But the same mechanics can be easily adapted onto higher or lower dimensions.

A kernel is a three dimensional matrix commonly used as a filter in image processing. It can be used for blurring, sharpening, edge detection and many more. A kernel is used in a mathematical, two dimensional convolution onto a three dimensional input $x = (x_{i,j,k}) \in \mathbb{R}^3$, $1 \leq i \leq n$, $1 \leq j \leq m$, $1 \leq k \leq c$. In this thesis $n \in \mathbb{N}$ is the number of filter bank features representing one input file, $m \in \mathbb{N}$ is the size of the filter bank vectors and $c \in \mathbb{N}$ is the number of input channels. The number of features n is dependent on the input file. The convolution is applied onto the first and second dimension of the input.

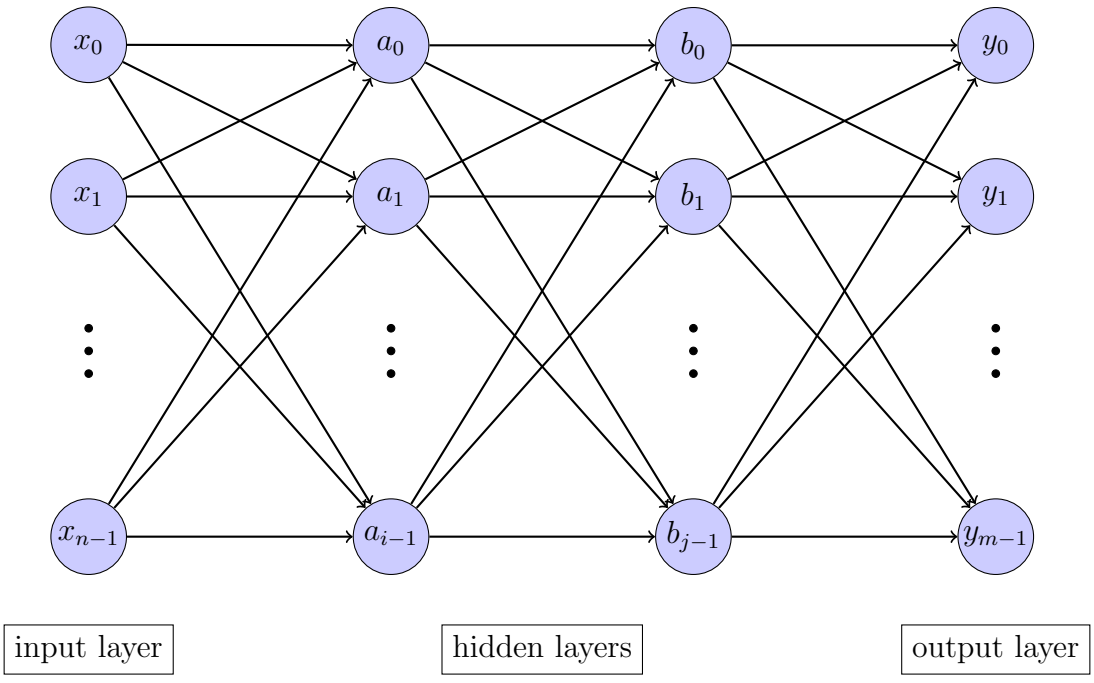

Figure 5: A fully connected neural network containing two hidden layers

The size of the first dimension of the kernel is from now on called kernel width K_w and the size of the second dimension is called kernel height K_h. If kernel width and kernel height are equal, and therefore the kernel is quadratic, we will call both of them kernel size K_s.

In Figure 6 a simple visualization of a convolutional kernel is given. The whole cube visualises an input sample with each small cube representing a single value. The red and blue area is the input to the kernel and at the same time the input of a single neuron in the convolutional layer. This means in the example a single neuron receives a $3 \times 3 \times 3 = 9$ dimensional input. The third dimension of the input is painted in blue, to visualize the the convolution is independent of the third dimension. Meaning the kernel always receives the full third dimension as input and in conclusion each neuron in the convolutional layer receives an input of $K_w \times K_h \times c$. From this point onward for simplicity our explanation will only refer to the first and second dimension of the kernel and explain everything following independent of the third dimension.

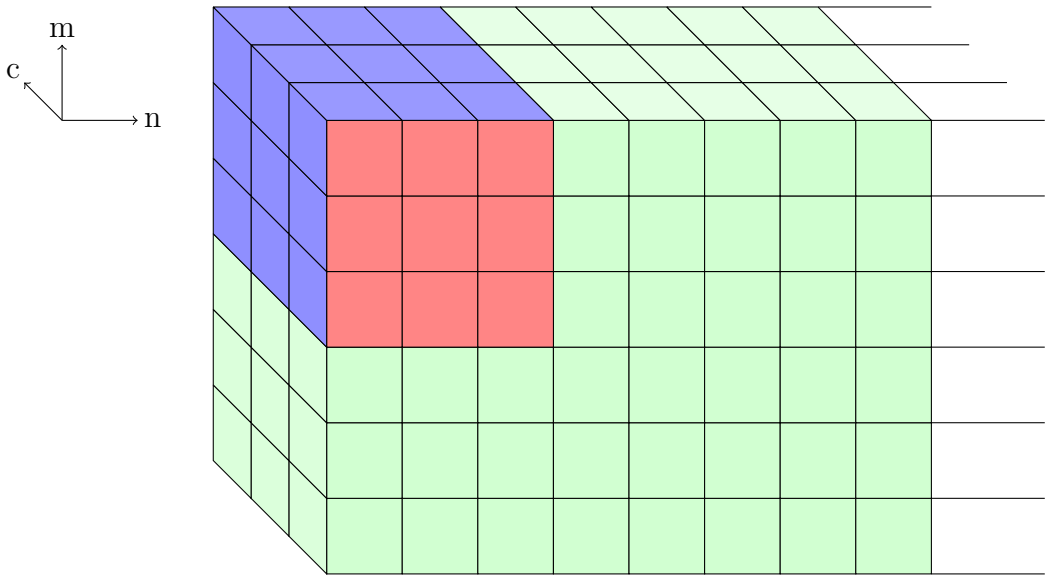

Figure 6: A cube representing the input and a single convolutional filter

One of the tricks of convolutional layers is that all neurons share the same weights. This means every neuron calculates the same output value for a given input and they could be regarded as only one filter used multiple times on the same input.

So far we explained the input a single neuron receives. If every neuron received exactly the same input, only a small part of the original features were used in the convolutional layer. Therefore in the following we will explain which part of the input is given to each neuron in the layer. Two important mechanics, stride and padding, are used to include the whole input matrix into the convolutional layer.

Stride is the mechanism of splitting the input into regions and use every region as input for a different neuron. The stride is the number of pixels each region is distanced to the neighbouring region on the input matrix. Normally the stride is smaller than the kernel size, meaning the inputs for different neurons overlap. Figure 7 shows a 3×3 kernel applied to a 8×5 dimensional input with a stride of two. The nine input fields coloured in red are given as input into one neuron producing the red output value. Same applies to the other colours. Because the stride is smaller than the kernel

size some neuron inputs overlap. There is for example the value in the third row and third column given as input to the red, blue, green and purple neuron. The most commonly used stride is a stride of one. This way a filter with kernel size centred around every input value is applied onto the input.

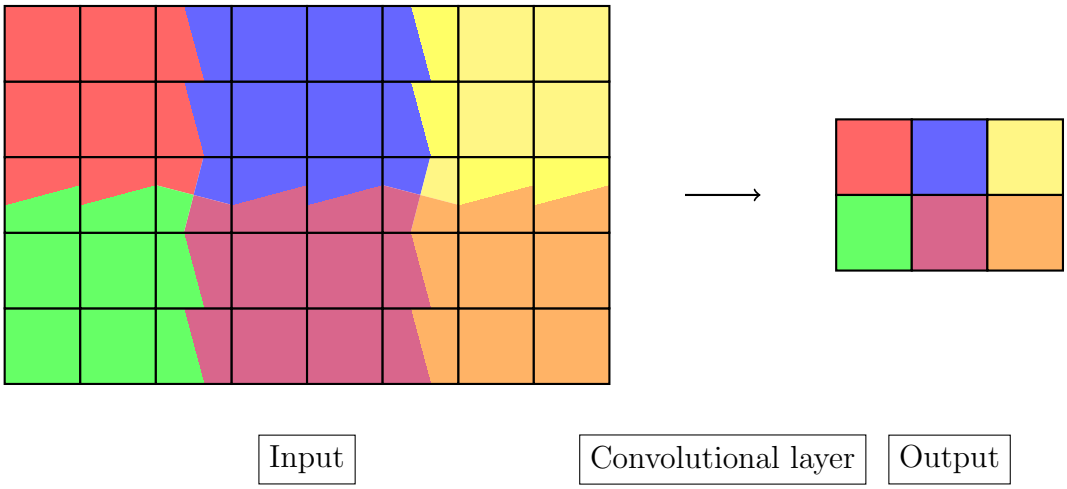

Figure 7: The input and output of a convolutional layer with a kernel size of 3×3, a stride of two and no padding

Applying a convolutional layer with a 3×3 kernel like shown in Figure 7 will decrease the output size in comparison to the input size even with a stride of one. The reason is that the filter in the top left corner is not centred around the top left corner input $x_{0,0}$, but rather around the input value $x_{1,1}$. If the kernel were to be applied to the top left corner input value $x_{0,0}$, five of the nine kernel inputs were out of the input bounds. Solutions for this border problem are paddings. Some common padding methods are listed below. In Figure 8 and example of a zero padding is provided.

- No padding: The kernel is only applied to input values, there is no outside the border (see Figure 7). This way the original dimension is reduced.
- Zero padding: The values outside the border are set to 0 (see Figure 8)
- Constant padding: The values outside the border are set to a constant k. Zero padding is a special and very common case of constant padding.

- Repeated padding: The values outside the border are set to the nearest input value.

Mathematically the input is given as $x = (x_{a,b,d}) \in \mathbb{R}^{n \times m \times c}$, $1 \leq a \leq n$, $1 \leq b \leq m$, $1 \leq d \leq c$. The input is distributed into overlapping areas $x_{a,b} \in \mathbb{R}^{kw \times kh \times c}, x_{a,b} \subset x$. The distance between the areas is the stride. The convolution is a matrix multiplication of each input area $x_{a,b}$ with the kernel weights $w \in \mathbb{R}^{kw \times kh \times c}$ creating an output $y_{a,b} \in \mathbb{R}$ as given by Equation 11 and combining them into the two dimensional output $y = (y_{a,b}) \in \mathbb{R}^{n \times m}$.

$$y_{a,b} = f_a(\sum(x_{a,b} \cdot w)) \tag{11}$$

In image processing the output of a kernel can be regarded as filtered image. This way each kernel represents an image processing algorithm highlighting a special characteristic of the input image. That is why the output of a convolutional kernel is referred to as feature.

119	-64	5	53
27	22	-5	7
119	-64	5	7
22	99	25	-111

(a) No padding

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	119	-64	5	53	0	0
0	0	27	22	-5	7	0	0
0	0	119	-64	5	7	0	0
0	0	22	99	25	-111	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(b) Zero padding

Figure 8: Example for zero padding

2.2.4 Convolutional Neural Network

CNNs are a special form of DNNs. Most of the networks hidden layers are convolutional layers. A convolutional layer consists of a set of convolutional kernels used to interpret the input tensor. By using multiple kernels the layer applies different convolutions on the input, each calculating its own independent output. A convolutional layer could for example use five kernels and thus receive a three dimensional input and create five independent two dimensional outputs. Those outputs are merged together into a three dimensional tensor. The follow up layer would than receive a three dimensional input containing five channels.
The idea of CNNs can best be explained on the problem of image recognition. In image recognition the input is three dimensional. It consists of two dimensions for the image pixels and one dimension for the colour channels. In image recognition each kernel would calculate a new two dimensional image containing information on different features. For example one kernel could highlight edges in the image and another sharpens the image. All calculated feature images would be taken as input for the next layer.
So in each convolutional layer, multiple kernel features are calculated and stacked. This stack is given to the next layer in the form of a new three dimensional tensor. This way a two dimensional convolutional layer receives a three dimensional input and creates a three dimensional output.
In this thesis the last convolutional layer will only contain one kernel and this way calculate a two dimensional output containing a new feature matrix.

2.2.5 Backpropagation

In the following we are going to explain the idea of the backpropagation algorithm. We will start by explaining the idea on a single neuron and afterwards generalize it onto a network.

Let $x = (x_0, x_1, ..., x_{n-1})$ be a given trainings sample input to a neuron with the desired output $y^* = (y_1^*, y_2^*, ..., y_{m-1}^*)$. Let the weights of one neuron be $w = (w_0, ..., w_{n-1})$ and the activation function f_a. Additionally in the following we will refer to the sum of all input values as $net = \sum_{i=0}^{n-1}(x_i \cdot w_i)$ and the output of the activation function as $out = f_a(net)$. The backpropagation algorithm consists of three steps.

1. Forward propagation

2. Backward propagation

3. Update weights

The forward propagation is quite simple. The sample is given to the neural network and an output $y = (y_0, y_1, ..., y_{m-1})$ is calculated. In the following we will refer to y as real output and to y^* as desired output. To evaluate the real output a loss function $E : \mathbb{R}^n \to \mathbb{R}$ is applied. In this thesis we used the common mean squared error given in Equation 12 as the loss function to calculate the loss L.

$$L = E(y, y^*) = \frac{1}{m} \sum_{j=0}^{m-1} (y_j - y_j^*)^2 \tag{12}$$

At this point the forward propagation is complete. Therefore the forward propagation calculates the output for a given sample and rates it using a loss function.

The back propagation step is the process of calculating the gradient of the loss L with respect to each weight w_i. That gradient is given as $\frac{\delta L}{\delta w_i}$ and can be calculated using the chain rule. For a variable z depending on y and at the same time y depending on x the chain rule is given in Equation 13.

$$\frac{\delta z}{\delta x} = \frac{\delta z}{\delta y} \cdot \frac{\delta y}{\delta x} \tag{13}$$

By applying the chain rule onto a single neuron one receives Equation 14. The weights influence on the loss is the sum of the gradient of each neuron evaluation state always in respect to the previous state.

$$\frac{\delta L}{\delta w_i} = \frac{\delta L}{\delta out} \cdot \frac{\delta out}{\delta net} \cdot \frac{\delta net}{\delta w_i} \tag{14}$$

We mentioned before that for efficient usage of the backpropagation algorithm the differential of the activation function must be efficiently calculable. The reason is that $\frac{\delta out}{\delta net} = \frac{\delta f_a(net)}{\delta net}$ and therefore the differential of f_a is needed for the back propagation step.

Equation 14 only applies for neurons with a single output target and therefore $L \in \mathbb{R}$ is a single value. However in case that the neuron had multiple output targets and their losses were given as $L_0, ..., L_{m-1} \in \mathbb{R}$ the

equation can be generalized by summing up the influence of the weight w_i onto each of those losses. The generalization is given in Equation 15. Because the influence from the weight w_i onto *out* does not change for the different losses the new equation can easily be optimized for faster calculation.

$$\frac{\delta L}{\delta w_i} = \sum_{j=1}^{m} (\frac{\delta L_j}{\delta out} \cdot \frac{\delta out}{\delta net} \cdot \frac{\delta net}{\delta w_i}) = \sum_{j=1}^{m} (\frac{\delta L_j}{\delta out}) \cdot \frac{\delta out}{\delta net} \cdot \frac{\delta net}{\delta w_i} \tag{15}$$

Up to this point we explained how to calculate the influence of a single weight of a neuron onto its output losses. This process can now be generalized for a whole network using the chain rule in the same manner as before.
For the generalization we define a neuron in a hidden layer of our network as neu_0. The neuron got the weights $w_{0,i}$ and additionally net_0 and out_0 defined the same way as before. In the same way we also define all neurons of the following layer receiving the output of neu_0 as neu_j, $1 \leq j \leq k$. Those neurons also got weights $w_{j,i}$ and the calculation steps net_j and out_j. For simplicity let additionally $w_{j,0}$ be the weight between neu_0 and neu_j.
In this setup the gradient of the loss in respect to one of the neurons weights $\frac{\delta L}{\delta w_{0,i}}$ can be calculated by the chain rule seen in Equation 16.

$$\frac{\delta L}{\delta w_{0,i}} = \frac{\delta L}{\delta out_0} \cdot \frac{\delta out_0}{\delta w_{0,i}} \tag{16}$$

In the formula, $\frac{\delta out_0}{\delta w_{0,i}}$ can be calculated as described before in Equation 14 and the interesting part is $\frac{\delta L}{\delta out_0}$. This one can again be calculated using the chain rule in a way described by Equation 17.

$$\frac{\delta L}{\delta out_0} = \sum_{j=1}^{k} (\frac{\delta L}{\delta net_j} \cdot \frac{\delta net_j}{\delta out_0}) \tag{17}$$

For $\frac{\delta L}{\delta net_j}$, if net_j is part of the output layer, it can be calculated the same way as described for a single neuron in Equation 15. If it is part of a hidden layer we can assume by induction that the value was already calculated in an earlier backpropagation step. So the only thing left to calculate is $\frac{\delta net_j}{\delta out_0}$ which is equal to the weight between neu_0 and neu_j as described by Equation 18.

$$\frac{\delta net_j}{\delta out_0} = w_{j,0} \tag{18}$$

So far we learned how to calculate the gradient of the loss function with respect to every single weight inside the neural network. The whole process was accomplished using the chain rule backwards starting at the output loss of the neural network.

The last step is to update each weight w_i using gradient decent on its gradient in respect to the final loss $\frac{\delta L}{\delta w_i}$. This way the updated weight w_i^* is given as $w_i^* = w_i - \gamma \frac{\delta L}{\delta w_i}$. In the formula, γ is the learning rate and determines the learning speed of the network. A high learning rate means that the network adjusts quickly to the trainings data but at the same time is less precise. A small learning rate on the other hand needs more time to train but gives more accurate results. Normally it is advantageous to have a high learning rate at the beginning of the training and a small learning rate at the end. That way the network trains efficiently fast and precise.

The backward propagation and its weight updates can either be performed on every sample, every batch or some given value of samples smaller than the batch size. Additionally the learning rate γ can be adapted whenever a weight update is performed. Deciding how to setup the learning rate, adjust the learning rate and the updating process is explained in Chapter 2.2.6.

2.2.6 Optimizer

An optimizer is a method used in the training process of a neural network. The optimizer decides on how to update a weight w_i with a given gradient $\frac{\delta L}{\delta w_i}$. The updated weight is w_i^*. A high learning rate can stop the network from converging and a low learning rate can cause the network to end up in a minor local minimum. Therefore optimizer have a great impact on the neural networks performance. There are multiple variations of optimizers. In the following we will take a look at the most important ones implemented in Keras. We will only give a brief overview on each optimizer, for a better understanding we suggest Sebastian Ruders summary of optimizers [24].

- **Stochastic Gradient Descent (SGD)** $w_i^* = w_i - \gamma \cdot \frac{\delta L}{\delta w_i}$
  SGD is the common gradient descend algorithm with γ as learning rate. It can be improved using momentum and decay.

- **Momentum** $v_j = (\alpha \cdot v_{j-1} + \gamma \cdot \frac{\delta L}{\delta w_i}),\ \ w_i^* = w_i - v_j$
  Momentum was introduced to reduce fluctuation while training. In momentum the past update change v_{j-1} is remembered and used to calculate the new update value v_j. Thereby α is the influence of the previous update onto the new one.

- **Nesterov accelerated gradient (NAG)** $v_j = (\alpha \cdot v_{j-1} + \gamma \cdot \frac{\delta L}{\delta w_i - \alpha v_{j-1}})$, $w_i^* = w_i - v_j$
  NAG is similar to momentum with a small change. Instead of calculating the gradient of the momentarily weight, the weight is changed according to the momentum $\alpha \cdot v$ and a future prediction is done. This way the new weight w_i^* is shifted in the momentum direction and afterwards adjusted according to the gradient.

- **Adaptive Gradient Algorithm (AdaGrad)**
  AdaGrad has a learning rate for each weight. The learning rate for weight w_i is modified based on all past gradients of w_i.

- **Adaptive Learning Rate Method (AdaDelta)**
  AdaDelta improves AdaGrad by only using a set part of the past weight gradients to adapt the learning rate. This can be achieved efficiently by adding the past squared gradients into one value with an exponentially decaying average. This way only a fraction of the original gradient remains after a given number of samples.

- **RMSprop**
  RMSprop is another approach like AdaDelta. The two algorithms were developed around the same time and are based on the same idea.

- **Adaptive Moment Estimation (Adam)**
  Adam is the AdaGrad algorithm with the add of momentum. This means the optimizer remembers a portion of the past squared gradients and additionally remembers a decaying portion of the past not squared gradients. Both values are used to influence the new weight.

- **AdaMax**
  AdaMax is a small variation of Adam using the maximums norm instead of the square of past gradients.

- **Nesterov-accelerated Adaptive Moment Estimation (Nadam)** Nadam is a combination of Adam and Nag using the Nag method of shifting the weight by its past momentum and afterwards adjusting it according to the gradient.

2.3 Environment

This section contains all information needed to understand our working environment. In Subsection 2.3.1 we give a short overview on the deep learning framework Keras. Afterwards the speech recognition toolkits Kaldi and Eesen will be explained in the Subsections 2.3.2 and 2.3.3. And in the last Subsection 2.3.4 we will introduce the Wall Street Journal (WSJ) corpus.

2.3.1 Keras

Keras is an open source, self-contained framework for deep learning written in Python. It uses either Tensorflow [13], CNTK [25] or Theano [26] as back end. The main goal of Keras is to provide a library for fast and easy experimentation on DNNs. The API is self-contained and therefore the user never has to interact with the underlying engine and does not need any knowledge on the back end system. Keras is build up on four principle guidelines.

1. **User friendliness.** "Keras is an API designed for human beings, not machines."[27]. To achieve this goal Keras reduces the number of steps needed for common use cases, provides good and easy understandable feedback upon errors and offers a standardized API with good documentation and many examples.

2. **Modularity.** Initialization schemes, cost functions, optimizers, neural layers, activation functions, regularization schemes are all autonomous. They can be combined to form all sort of different deep learning layouts.

3. **Easy extensibility.** New modules, algorithms and state-of-the-art research can be added without changing or adjusting most of the code. This makes Keras exceptionally interesting for research.

4. **Work with Python.** Python is easy adjustable, quick for testing, compact and the standard programming language for deep learning.

Therefore Keras is fully implemented in Python and can be used without the need of other languages.

2.3.2 Kaldi

Kaldi[28] is a popular, free, open-source toolkit for speech recognition research introduced by Daniel Povey et al. in the year 2011. Kaldi contains implementations for most standard techniques of speech recognition. The toolkit creates state-of-the-art results for many different speech tasks and especially for the WSJ dataset described in 2.3.4. The code is written in C++ and designed to be flexible, easy modifiable and extendible.

2.3.3 Eesen

Eesen is a common, free, open-source toolkit for speech recognition research introduced by Yajie Miao, Mohammad Gowayyed and Florian Metze from Carnegie Mellon University in 2015. Eesen is an extension of Kaldi, building up on the idea of creating a flexible research environment for speech recognition. The biggest difference between those two toolkits is, that Eesen replaces many elements of Kaldis ASR pipeline with a single Recurrent Neural Network (RNN). It replaces the Hidden Markov models, Gaussian mixture models, Decision trees and phonetic questions with a RNN directly mapping speech to text. Therefore Eesen is easier to use and consists of a more comprehensive source code. But even though the toolkit is slimmer, its results are state-of-the-art and comparable to Kaldi.

2.3.4 Wall Street Journal Corpus

The WSJ corpus [11, 12] is a data corpus used for continues speech recognition created by John Garofolo et al. in the years 1991 to 1994. The plan was to create a corpus for research on Large-Vocabulary Continues Speech Recognition (LVCSR) systems. The corpus consists of two parts WSJ0 and WSJ1, both can be obtained from the Linguistic Data Consortium (LDC) [29] under the catalog numbers LDC93S6B and LDC94S13B. The two sets consist mainly of read speech drawn from wall street journal news texts. The corpus contains 81 hours of transcribed English speech given as 1-channel by Pulse-code Modulation (PCM) compressed audio with a sample rate of 16000 hz. The speech was recorded in a sound safe environment.

3 Feature Denoising using CNNs

In this chapter we are going to present our experiments conducted on robust speech recognition using CNNs. We start of in Section 3.1 by giving an overview on previous research conducted on SE systems and neural networks, providing a motivation for our thesis. Afterwards we will describe the steps needed to prepare our dataset and working environment in Section 3.2. In Section 3.3 the experimental setup will be described. The optimization results will be provided in Section 3.4 and the evaluation is given in Section 3.5. At the end we will show that our system generalizes on different noises, speaker and even real world data in Section 3.6.
To create an easy adjustable working environment for noisy speech recognition using neural networks, our project is build modular. Each module represents a code asset independent of the rest of the code. This way the project can easily be adjusted for further experiments. For example the denoising could be done using waveforms instead of features or using a different feature type or a different dataset. Additionally the type of noise, the method to mix clean audio with noise, the noise levels, the denoising network or the speech recognizer are adjustable.

3.1 Literature Review

In this Section we give a brief overview on the current state of research. The Section was split into two parts. First we will introduce some important methods used for SE without neural networks. Afterwards we show the improvements done by neural networks in the fields of speech recognition and SE.

One classic approach for SE is the Wiener Filter [30] being used and studied up to date. The filter has recently been analysed by J. Benesty et al. [31], who found out that for a single-channel Wiener filter, the noise reduction is most of the time proportionate to the speech distortion.
Another classic SE algorithm is the spectral subtraction method [32]. The algorithm is used for example by Anuradha R. Fukane et al. [33], who used variations of the spectral subtraction method to reduce the musical noise distortion in hearing aids. Their experiments showed that their algorithms could be used to improve speech quality without affecting the intelligibility of the speech signal. Other modern examples for the spectral subtraction methods are the variations of S. S. Bharti et al. [34] and R. M. Udrea et al.

[35].
Other state-of-the-art methods for SE are signal subspace approaches [36]. Adam Borowicz [37] recently used a signal subspace approach for spatio-temporal prediction for multichannel SE. In their publication they used the signal subspace approach to further increase the results of temporal spatial prediction. The evaluation showed that the signal subspace approach outperformed the conventional time-domain [38] method by providing lower speech distortions but at the same time higher noise attenuation.

All these classic approaches give decent results up to date and are still used in a lot of different applications. Nevertheless as well as in many other fields of research, experiments including neural networks were performed. And often DNNs were either able to keep up with the conventional methods or outperform those classic algorithms. In the following we give a brief overview on neural network based SE methods.
Y. Xu et al. [1] tested a DNN with up to four layers for SE. The goal of their experiments was to enhance the noised audio quality. Their setup improved the speech quality on an SNR range of -5 to 20 dB and outperformed the classic log minimum mean squared error approach [2] and shallow neural networks.
F. Weninger et al. [39] used a long short-term memory (LSTM) Recurrent Neural Network (RNN) for SE tasks. Their experiments were conducted on the CHiME-2 [4] speech recognition task achieving a 13.76% average WER for SNRs of -6 to 9 dB. These may be the best results for the CHiME-2 task up to date.
J. Chen et al. [40] proposed a DNN with more than 20 million tunable parameters trained with 10.000 noises. Their system was developed for patients needing a hearing aid or having a cochlear implant to improve speech intelligibility in noisy environments. The results showed an absolute increase on the number of correctly recognized words of 27 % for hearing-impaired listeners at a SNR of 0 dB with babble noise.

Continuing with the study of neural networks for SE we decided to use CNNs as SE system. This is reasonable because CNNs had outstanding performances in other areas of research connected to SE.
Ossama Abdel-Hamid et al. [41] used CNNs for phoneme recognition on the TIMIT [42] dataset. In their studies CNNs create significant improvements on the Phoneme Error Rate (PER) in comparison to DNNs. In their experiments

CNNs reduced the PER by 6 to 10 % on the TIMIT phoneme recognition and the voice search large vocabulary speech recognition tasks.
Ying Zhang et al.[5] from university of montreal also used CNNs for phoneme recognition and showed that CNNs can achieve state-of-the-art performance comparable to LSTMs and other RNNs. Theire setup used nearly the same number of parameters for the CNN and RNNs, but the CNNs could be trained faster achieving a 2.5 times faster training speed. At the same time the CNNs received comparable results as the RNNs.
Dimitri Palaz et al. [43] used CNNs for speech recognition directly on raw speech as input. They showed that CNNs create significant improvements on the Aurora [44] test set on different SNRs in comparison to standard Hidden Markov models (HMM) in combination with artificial neural networks. They also showed that the features learned by the CNN could generalize across different databases.

While our experiments were ongoing, others released studies on CNNs for SE. In the following we will give a brief overview of their promising results.
S. R. Park et al. [9] used a fully connected CNN for SE. They proposed a convolutional encoder decoder network and tested their results on the TIMIT dataset with 27 different types of noise clips. Their experiments showed that CNN can achieve similar or better results in comparison to Feedforward Neural Networks (FNN) and RNNs while at the same time containing less parameters.
S. Fu et al. [45] proposed a fully convolutional neural network based speech enhancer on raw waveforms. Their experiments suggested that the CNN, beside having a drastically parameter reduction to only 0.2% in comparison to DNN, created better results.

Up to this point there exist many more studies of neural networks and CNNs for SE. So many that we can not list them all, but for a good overview on neural network based SE we suggest the paper of Z. Zhang et al. [46]. They give a very brief overview on the current developments of robust speech recognition.

3.2 Data Preparation

In this section we are going to explain everything needed for the data preparation and setting up of our working environment.

The first thing we needed to do was selecting a dataset for our experiments. We decided to use the WSJ corpus [11, 12] containing clean speech. The WSJ corpus is a standard set for ASR and also it is one of the sets with preexisting scripts in Eesen. For further information about the WSJ dataset see Chapter 2.3.4.

For the experiments we needed clean and noisy training, validation and testing data. The WSJ *train_tr95* and *train_cv05* subsets were used for training, the *test_dev93* subset for validation and the *test_eval92* subset for testing. The training data was later used to train the denoising system and also the clean speech recognizer. The validation set was used for the hyper parameter optimization and the test set was used for the evaluation of our final network.
As noise files we used the babble background noise of the 4^{th} CHiME Speech Separation and Recognition Challenge [4] recorded in a pedestrian area. The noise mainly contains background noises of talking people.
The data preparation was done in three main steps:

1. Convert the audio files into riff wave format
2. Combine speech signals and noise signals at different SNRs
3. Calculate the filter bank features and create Eesen information folders

The WSJ dataset was originally stored in the NIST Sphere [47] file format which is not supported by most modern programs and toolkits. Thus for further processing of the speech files we needed to convert them into a modern standard format. Therefore we used the Sphere conversion tools [48] provided by LDC, to convert the files into riff wave files.
For further processing the noise and speech files needed to be in the exact same format. To achieve this we converted the given noise files into the format used by the speech files using the sox [49] command-line interface. The WSJ audio files consist of one audio channel each and use a sample rate of 16 kHz with a precision of 16 bit and therefore a bit rate of 256 kbit per second. The noise files were converted to the riff wave format with the same number of channels, sample rate and bit rate as the WSJ audio files.

The next step was the creation of noisy speech audio. The speech files were artificially noised by mixing each speech signal with one noise signal.

In the following we take a look at the noising of a single speech file given as digital audio signal $s = (s_1, s_2, ..., s_n)$. For the mixing one of the noise files was randomly chosen and given as digital audio signal $b^* = (b_1^*, b_2^*, ..., b_m^*)$. One requirement was that the given speech file was shorter than the noise file $n < m$. Therefore the first step was to cut the noise signal into the same length as the speech signal. For the trimming a random function $rand(j, k)$ was used, returning a random integer between $j \in \mathbb{Z}$ and $k \in \mathbb{Z}$, both included. The starting point of the trimming was given as $b_{start}^* = rand(1, m - n)$ and the end point was $b_{end}^* = b_{start}^* + n$. Next the noise was cut to $b = (b_1, b_2, ..., b_n) = (b_{start}^*, b_{start+1}^*, ..., b_{end}^*)$ using sox. Up to this point a clean speech signal and a noise signal in the same format with the same length were given.
Next in line was to mix these two signals with different SNR levels. The SNR level was given as SNR in dB. For the mixing with a concrete SNR we needed the power of both audio signals. The power of both audio signals was calculated by Equation 4 and is defined as s_{pow} for the speech signal and b_{pow} for the noise signal.
Finally for the mixing we needed to calculate a relation α of the strength used for the speech signal and the strength of the noise signal. The relation was dependent on the two signals powers and calculated to assure that by adding the noise with the ratio α onto the speech signal, the output signal had the desired SNR. The ratio can be calculated by Equation 19.

$$\alpha = \sqrt{\frac{s_{pow}}{b_{pow} \cdot 10^{SNR}}} \tag{19}$$

Using the ratio we could at last create the noised speech signal $x = (x_1, x_2, ..., x_n)$ by adding the speech and noise signal as done in Equation 20.

$$x_i = s_i + b_i \cdot \alpha \tag{20}$$

The data was noised to SNRs of 30 dB, 24 dB, 18 dB, 12 dB, 6 dB, 0 dB and -6 dB. Steps of 6 dB were used because by increasing an audio signal by 6 dB it is approximately twice as strong as before. Thus the noise of a SNR of 18 dB is supposed to sound twice as loud as the noise of a 24 dB SNR. Additionally to the noisy datasets a clean set containing the original speech files without noise and a SNR_mix set containing evenly distributed files from the different noisy sets (including the clean set) were created. Not all training, validation and testing sets were needed with all SNRs. The list of created datasets is given in Table 1.

SNR [dB] / Dataset	Clean	mixed	30	24	18	12	6	0	-6
train_tr95	✔	✔	✗	✗	✗	✗	✗	✗	✗
train_cv05	✔	✔	✗	✗	✗	✗	✗	✗	✗
test_dev93	✔	✔	✔	✔	✔	✔	✔	✔	✔
test_eval92	✔	✔	✔	✔	✔	✔	✔	✔	✔

Table 1: The datasets artificially created and used in the experiments

The final preparation step was to create an information folder for each of our datasets. The Eesen toolkit uses a special data structure to represent datasets. Each dataset is represented by a single folder containing seven files. The files are *spk2gender*, *spk2utt*, *utt2spk*, *text*, *wav.scp*, *feats.scp* and *cmvn.scp*. An utterance id is a unique id used to identify each speech sample. The *spk2utt* and *utt2spk* files are mappings of a speaker to all utterance ids spoken by that person. The *spk2gender* file maps a speaker to his gender. The *text* file maps each utterance id to the spoken sentence in text form. The *wav.scp*, *feats.scp* and *cmvn.scp* files contain paths to the audio file, feature matrix and Cepstral Mean and Variance Normalization (CMVN) [50] coefficients of each utterance. The information folder for the clean datasets were prior created by the Eesen preparation.
All Eesen information folders for the noisy datasets were created by the method explained in the following. For the explanation the clean information folder will be referred to as original folder and the new noisy information folder will be referred to as noisy folder. The text, *spk2gender*, *spk2utt* and *utt2spk* files were copied from the original information folder into the noisy one. The filter bank features and CMVNs were calculated using the preexisting *make_fbank.sh* and *compute_cmvn_stats.sh* Eesen scripts. By using the scripts the *feats.scp* and *cmvn.scp* files were created as well. The last file was created by writing the utterance ids jointly with their respective noisy speech files into the *wav.scp*.
Additionally to the information folders we needed the filter bank features in the form of Python matrices for the training, testing and evaluation of our denoising network. Unfortunately the filter bank features were given in a compressed, binary format. Therefore using the *copy-feats* command we converted the features into a human readable format. We than applied

a self written script and converted the human readable filter bank features into a NumPy [51] matrix. For each audio sample a single file containing a $N \times 40$ dimensional feature matrix was created and later used for the network training, validation and testing.

3.3 Experimental setup

In the following we are going to explain the setup of our conducted experiments. Those experiments were based on the Eesen toolkit[10] using the standard setup of Eesen. The only modification was the added CNN denoising part. Therefore our experimental pipeline consisted of three main steps.

1. Train the denoising and speech recognition systems
2. Denoise the noisy datasets
3. Use the speech recogniser on the denoised data

The layout of our networks and the setup of the training environment will be introduced in Subsection 3.3.1. In Subsection 3.3.2 we introduce the setup of our optimization and evaluation process and in Subsection 3.3.3 we explain how the batches were artificially created.

Our denoising system is a fully convolutional neural network, meaning we got a CNN only consisting of convolutional layers. The network was implemented in Keras [23] using Tensorflow [13] as a back end. The input features were $N \times 40$-dimensional filter bank features, where N is variable to fit the size of each input file and corresponds to the number of frames. The features were calculated using Eesen's filter bank calculation. The output features were also $N \times 40$-dimensional filter bank features with reduced noise.

3.3.1 Training

In our setup there were two networks which need to be trained. The first network was our denoising system and the second one was the Eesen speech recognizer.
For the training, the denoising CNN learned to map the mixed *train_tr95* set to the clean *train_tr95* set in form of filter bank features. This way the network learned to map noisy features with all different SNRs onto clean features. A batched training with each batch represented by one speech file was used. A batch is a set of input samples used to train the network at once. In the backpropagation algorithm the gradient of one batch is calculated before the weights are updated. After each sample was used once for the network training an epoch ends. As regularization a simple variation of early stopping was implemented. After each epoch the trained network was validated on the mixed *test_dev93* dataset and the MSE of the calculated and the real clean *test_dev93* set was used as metric. The training stopped if the MSE on the test set did not improve for three epochs. Since the training was done by Keras, the Keras internal back-propagation algorithm was used.

The training and denoising algorithms were enclosed in self-contained scripts. The training script only received the hyper parameters and layer information of the CNN as input and the denoising script received input data and the trained network as input. The hyper parameters and layer information were given to the training script in form of a self made format explained in the following.

[<Loss> <Optimizer> <#Batches> <#Layers> <Layers>]

The format was enclosed by square brackets and contained different informations divided by spaces. In this setup <Loss> is the loss function used in the training process. <Optimizer> is the optimizer used by the back propagation algorithm. <#Batches> is the number of batches used in each training step. <#Layers> is the number of hidden layers contained in the denoising network. These are the hyper parameter needed to define the CNN. Beside the hyper parameters there exist also the parameters of each hidden layer. Those were given as a list in the <Layers> section. <Layers> contains for each hidden layer information in the following format.

<#Kernels> <Width> <Height>

In this format <#Kernels> is the number of kernels used in this layer and therefore additionally the number of input channels the next layer receives. <Width> is the width of the kernels and <Height> is the height of the kernels.
One example network configuration could be given as:

[mse adadelta 10 2 10 5 5 softplus 1 7 7 linear]

This network consist of two hidden layers. The first hidden layer consists of ten kernels with a size of 5×5 and a softplus activation function. The second hidden layer consists of one kernel with a size of 7×7 and a linear activation function. The network is trained with mean square error as loss function and an adadelta optimizer receiving ten batches at a time.

The stride was set to one and padding is set to zero padding. The learning rate was not adjusted and depends on the optimizer. Every other parameter not mentioned here was kept unchanged and therefore was left as the Keras or Tensorflow default value.

CNN's often consist of multiple convolutional layers followed by pooling layers. Pooling layers reduce the dimensionality of the image. In our case we did not use any pooling layers, because we wanted the output dimensionality to be exactly the same as the input dimensionality. Also a lot of networks add fully connected layers at end of their network. This is for example useful for image recognition problems, where CNNs calculate and filter important information out of the input and fully connected layers are used to interpret the information. In our case we only wanted to calculate new information out of existing one and let the speech recognizer interpret it. Therefore only using convolutional layers seemed more appropriate and also showed promising results before [9, 45].

The Eesen speech recognizer was trained on the standard Eesen setup for phoneme based speech recognition. The standard setup trained the recognizer using the *train_tr95* dataset as trainings set and the *train_cv05* set as validation set with a learning rate of 0.00004.

3.3.2 Optimization and Evaluation setup

The optimization of hyper parameters was conducted on the WER calculated by the output of the speech recognition system and not on the output of the denoising system. That means our denoising system was trained on the *train_tr95* dataset and used to denoise the *test_dev93* dataset. Afterwards the speech recognition system was applied on the denoised data and the WER was determined as described in Subsection 2.1.6. The hyperparameters creating the best average WER as final result were chosen for further experiments. The denoising and speech recognition were applied to the clean, mixed, SNR 30, SNR 24, SNR 18, SNR 12, SNR 6, SNR 0 and SNR -6 *test_dev93* datasets. As optimization metric the average WER of all datasets was used. Because the fluctuation of the WER is stronger on audio files with strong noise we assumed that our metric supports low value SNRs more than high value SNRs. Thus we assume that by using a different metric the denoising system could improve on low noised data while taking cuts on highly noisy data.

Because each training, denoising and decoding cycle was time-consuming the optimization was parallelized. The only information needed for each individual, parallel step was the network layout described in Section 3.3.1. The data preparation as well as the training of the speech recognizer was processed once beforehand. The pretrained recognizer network was reused in every parallel run. This way each parallel process consisted of four steps.

1. Train the denoising CNN with a given network layout
2. Denoise the mixed *test_dev93* dataset
3. Decode the denoised dataset
4. Output the WER in %

For the evaluation the clean *test_eval92* set and the *test_eval92* sets noised with a SNR of 30 dB, 24 dB, 18 dB, 12 dB, 6 dB, 0 dB and -6 dB were denoised using the denoising CNN. Afterwards all those datasets were decoded and the WER was calculated.

As the start configuration for our optimization we used the following network visualized in Figure 9.

[mse adadelta 1 7 10 10 10 softplus 10 10 10 softplus 10 10 10 softplus 10 10 10 softplus 10 10 10 softplus 10 10 10 softplus 1 10 10 linear]

This network used a mean squared error loss function, which is the most common loss function. As optimizer we used an adadelta optimizer, because it does not need any hyperparameter optimizations for decent results. We decided to use seven convolutional hidden layers as starting point and optimize the number of hidden layers as first optimization step. Each layer, but the last one, uses ten kernels with a size of 10×10 and softplus as activation function. The last layer only uses one kernel to create a two dimensional output equivalent to the networks input dimensions. Also the last layer uses a linear activation function, because softplus calculates values in the region of [0,1], but our filter bank features were not downscaled to this numerical area. So an activation function returning values in the 16-bit integer range was needed and we decided to use a linear activation function. As starting point for the batch size we decided to use one file per batch, since it was the fastest to implement and seemed good enough for the first optimization steps. But of course the number of batches was later optimized as well.
The input layer was $N \times 40 \times 1$ dimensional. The last hidden layer always consisted of only one kernel and therefore the output layer was also $N \times 40 \times 1$ dimensional.

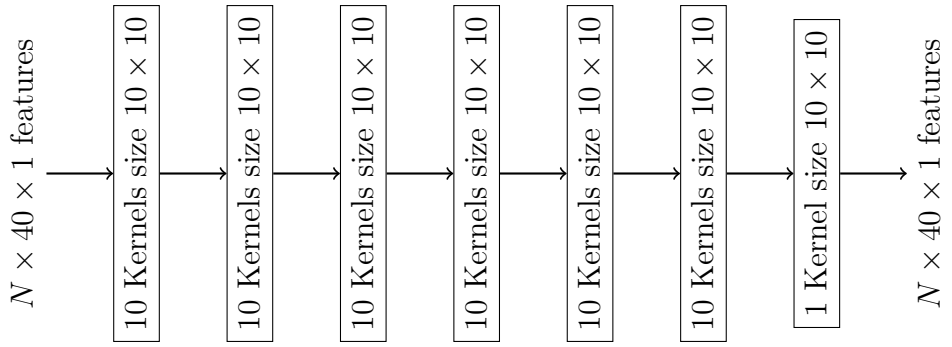

Figure 9: The starting CNN for our optimization process

For the evaluation we trained four different types of ASR systems. The first ASR setup was the original Eesen speech recognizer used in the WSJ example. The recognizer was trained on the clean *train_tr95* dataset and no denoising system was added. Of course this ASR system is not fit for robust speech recognition and was mostly used as state-of-the-art baseline

comparison system for the other ASR systems. The assumption for the standard speech recognizer was that it creates the best results on clean audio files but got a quickly decreasing performance on low SNR levels. In the evaluation we referred to this ASR system as clean ASR.

The second setup as well did not use a denoising system. The standard Eesen speech recognizer was trained on the mixed noise *train_tr95* dataset instead of the clean one. This way the robust ASR system was supposed to learn speech recognition under noisy conditions and be able to ignore the noise. The assumption for this system is to create decent results for every noise level and therefore be a good comparison for our denoising CNN. In the evaluation we referred to this ASR system as noise trained ASR.

The third setup was our developed robust ASR system based on the standard Eesen speech recognizer with an added denoising CNN. We hoped for our setup to vastly increase the performance of the original ASR on the moderate SNR levels and therefore transform noisy data into clean data. In the evaluation we referred to this ASR system as denoise CNN ASR.

The fourth setup was a mix between the second and third system. We used our denoising CNN plus the standard Eesen speech recognizer trained on the output data create by the denoising system. This means instead of training the speech recognizer on clean speech we used our denoised filter bank features as training data. The assumption for this system was to even further increase the performance of our denoising setup. In the evaluation we referred to this ASR system as denoise trained ASR.

For the evaluation we used the clean and the SNR -6 dB, 0 dB ,6 dB, 12 dB, 18 dB, 24 dB and 30 dB versions of the *test_eval92* set. We will additionally present the results of the validation *test_dev93* dataset. Further more the WER was used as performance metric.

3.3.3 Batches

Up until now we have not explained how we created our batches. We thought of three possible solutions to create batches containing information of multiple input files. The first and most obvious idea was to create a four dimensional

tensor and place all batches in the fourth dimension. That way the first dimension would be the number of feature vectors depended on the length of the input file. The second dimension would be the feature vectors with a size of 40. The third dimension would be the channel size which is one. And the fourth dimension would be the feature tensors of the batches. The size of the fourth dimension would than be equal to the batch size.
The problem of this approach was that the first dimension is depending on the size of the input file and therefore different for each file. Hereby the problem was that we could not simply add the input tensors together unless all input tensors got the same size. One solution could be to fill all shorter tensors with zeros at the end. The problem here is, that all audio files are around ten seconds long, but the distance between files often reaches one or two seconds. By adding zeros to resize the files to the same length, up to one sixth of the resulting batch input tensor would be filled with zeros and strongly influence the trainings results.
The second approach was to concatenate the audio files and afterwards calculate the feature vectors. Because the batch size changed, this approach would lead to the batch creation and the calculation of the feature vectors within the parallel processing of the training. Though the calculation of feature vectors is time consuming and therefore for time efficiency we decided to not use this approach either.
The third method gives nearly the same results as the second approach without the loss of time. Instead of the audio files we concatenated the feature tensors of multiple sample files in the first dimension. This way we received one big feature tensor.

3.4 Optimization

In this section we explained our optimization process. The initial network of the experiments was described in Subsection 3.3.2 and was given as:

[mse adadelta 1 7 10 10 10 softplus 10 10 10 softplus 10 10 10 softplus 10 10 10 softplus 10 10 10 softplus 10 10 10 softplus 1 10 10 linear]

Optimizing multiple hyperparameters in parallel creates better results but is more time-consuming. For time saving we decided to optimize only one parameter at a time. We assumed that the number of hidden layers has the biggest influence onto the ASR results, therefore we started by optimizing

the number of hidden layers in Subsection 3.4.1. Afterwards we decided to optimize the batch size in Subsection 3.4.2 and the number of kernels in 3.4.3. We also tried out different optimizer in Subsection 3.4.4. This way our optimization steps were given as follows.

1. Number of hidden layers
2. Batch size
3. Number of Kernels
4. Optimizer

All optimizations were concluded on the *test_dev93* validation set and the metric was the average WER of the sets.

3.4.1 Number of Hidden Layers

The first hyper parameter optimization was done on the number of hidden layers. The hidden layers were all convolutional layers. Every layer but the last one had 10 kernels with the size of 10×10 and softplus activation functions. The last layer had one kernel of the size 10×10 and a linear activation function. If for example the number of hidden layers was set to one, the network consisted only of the last layer with one kernel. If the number was set to six, the network consisted of five hidden layers with ten kernels plus the last layer with one kernel.
At the beginning the hidden layer optimization was done on a number of hidden layers from one to ten, but those results were indistinct. Therefore we expanded the optimization range and tested the number of hidden layers up to twenty.

As the experiments visualized in Figure 10 showed a number of ten hidden layers gave the best results. As expected one hidden layer with only one kernel is not enough for proper speech denoising and thus created poor results. After adding more hidden layers the WER quickly dropped to an average value of 35.36%. For comparison, the average WER of the clean speech recognizer is 42.645% on the validation set. After reaching the minimum with ten hidden layers the WER started to raise again and showed erratic movements. We assume that the network became to big and started to overfit.
The optimized ten layered CNN is given as:

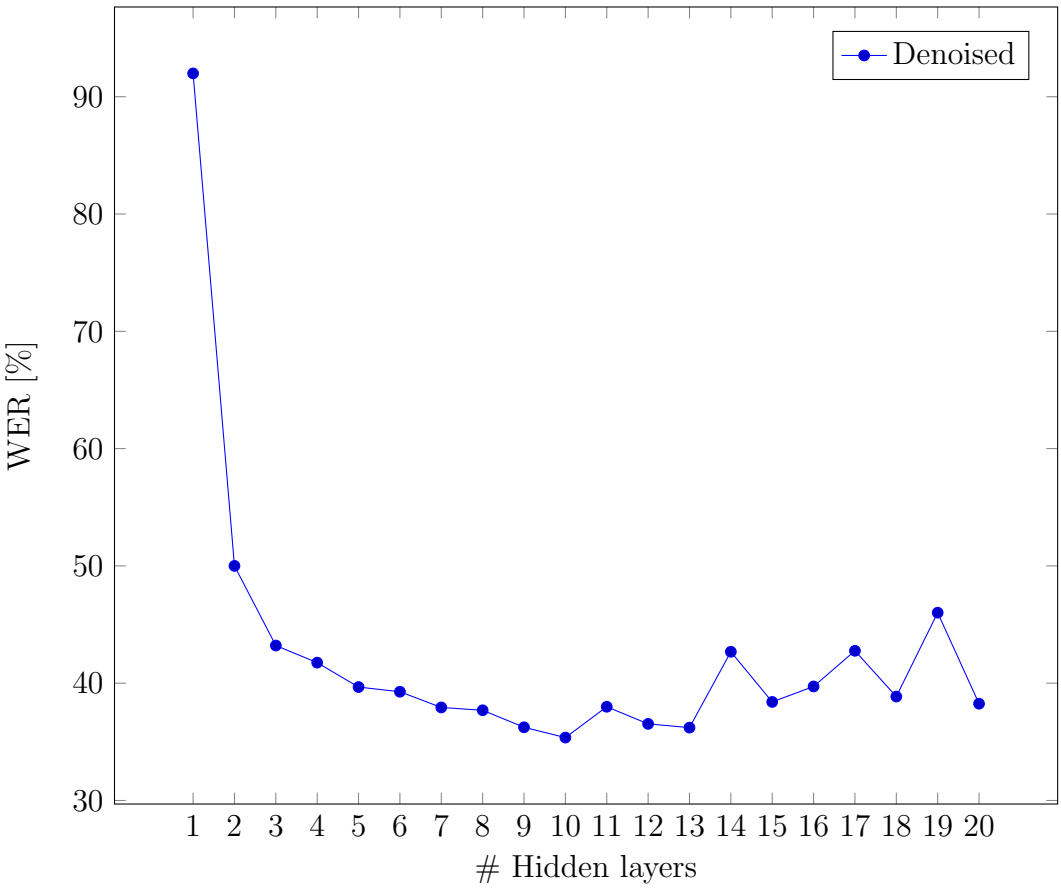

Figure 10: Optimizing the number of hidden layers

[mse adadelta 1 10 10 10 10 softplus 10 10 10 softplus 10 10 10 softplus 10 10 10 softplus 10 10 10 softplus 10 10 10 softplus 10 10 10 softplus 10 10 10 softplus 10 10 10 softplus 1 10 10 linear]

3.4.2 Batch Size

The second optimization step was done on the batch size. The batches were implemented as described in Section 3.3.3. The batch optimization was done in two steps. First we tested out exponential batch sizes of 1, 2, 4, 8, 16, 32 samples per batch and visualized the results in Figure 11. In the experiment we found out that a small batch size created good results while increasing

batch sizes increased the error rate. We ended the experiment with a batch size of 32, because larger feature tensors were too huge for our main memory and additionally previous results already showed that a batch size larger than 32 should return worse results. Afterwards we tested batch sizes between one and ten and visualized the results in Figure 12. Those experiment showed the same results as the previous one. A batch size of one created the best results and therefore the optimized network is the same as the optimized result of the previous optimization step.

[mse adadelta 1 10 10 10 10 softplus 10 10 10 softplus 10 10 10 softplus 10 10 10 softplus 10 10 10 softplus 10 10 10 softplus 10 10 10 softplus 10 10 10 softplus 10 10 10 softplus 1 10 10 linear]

Normally one would assume that a batch size of one is bad, because of missing diversity for each update in the training process. But in our experiments a small batch size created the best results. We thought of two possible reasons why the best batch size was unexpectedly one. The first reason was the cut between batches disturbing the speech recognition. The sudden change of background noise and speech between two samples caused the training to be worsened. The second reason was, that the tensors of each sample already contained a lot of data and with a rising batch size the amount of data became too much for proper training. Of course we could not determine the real reason and only gave assumptions.

3.4.3 Number of Kernels

The third optimization step was done on the number of kernels. The number of kernels was set equal for all hidden layers but the last one. Thus these experiments influenced the first nine hidden layers of the denoising network. A number of kernels of three would for example represent a CNN with 10 hidden layers. The first nine layers consists of three 10×10 sized kernels and the last layer consists of a single 10×10 sized kernel.
Same as the batch optimization we started of by exponential step size of the parameters. The results visualized in Figure 13 showed that the optimal kernel size should be between 4 and 16. Thus for the non exponential testing we decided to try out 1 to 20 kernels in each layer. The results of these experiments visualized in Figure 14 were as expected. With small kernel sizes the network created a high WER and therefore was bad at ASR. With a

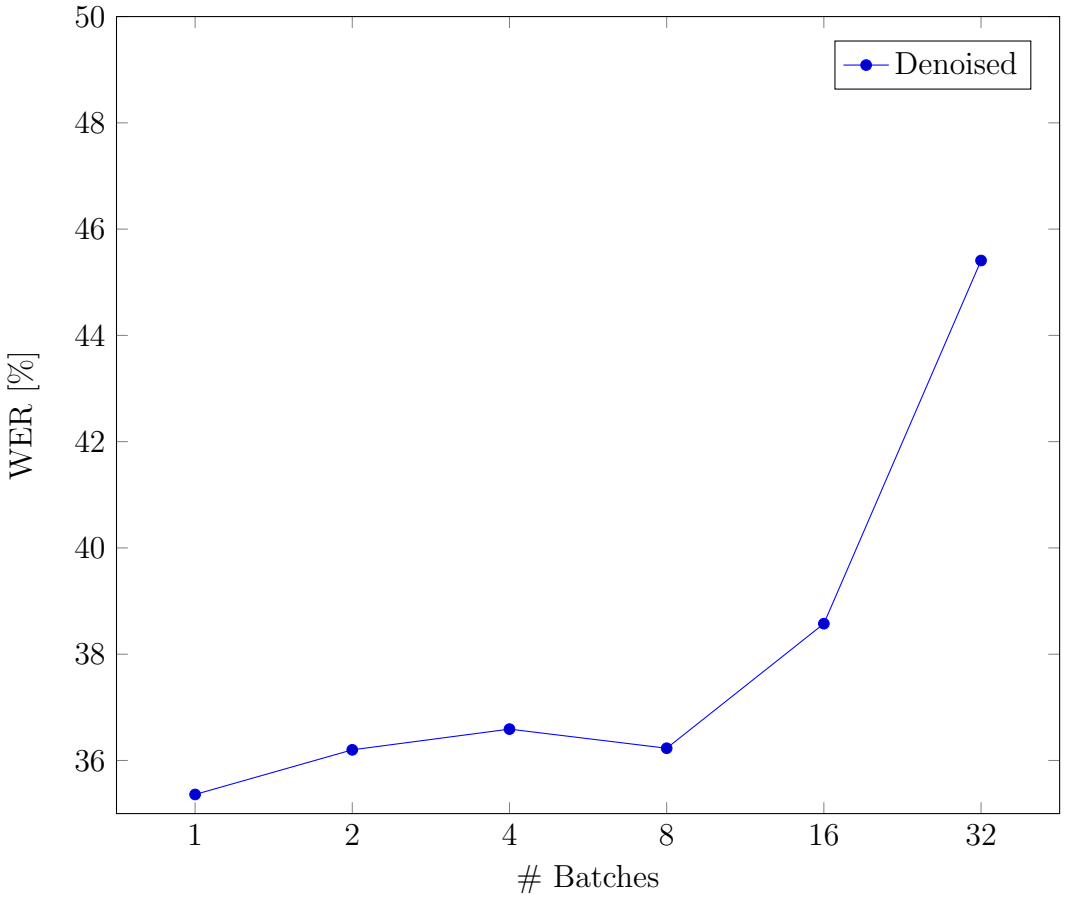

Figure 11: Optimizing the batch size, exponential step size

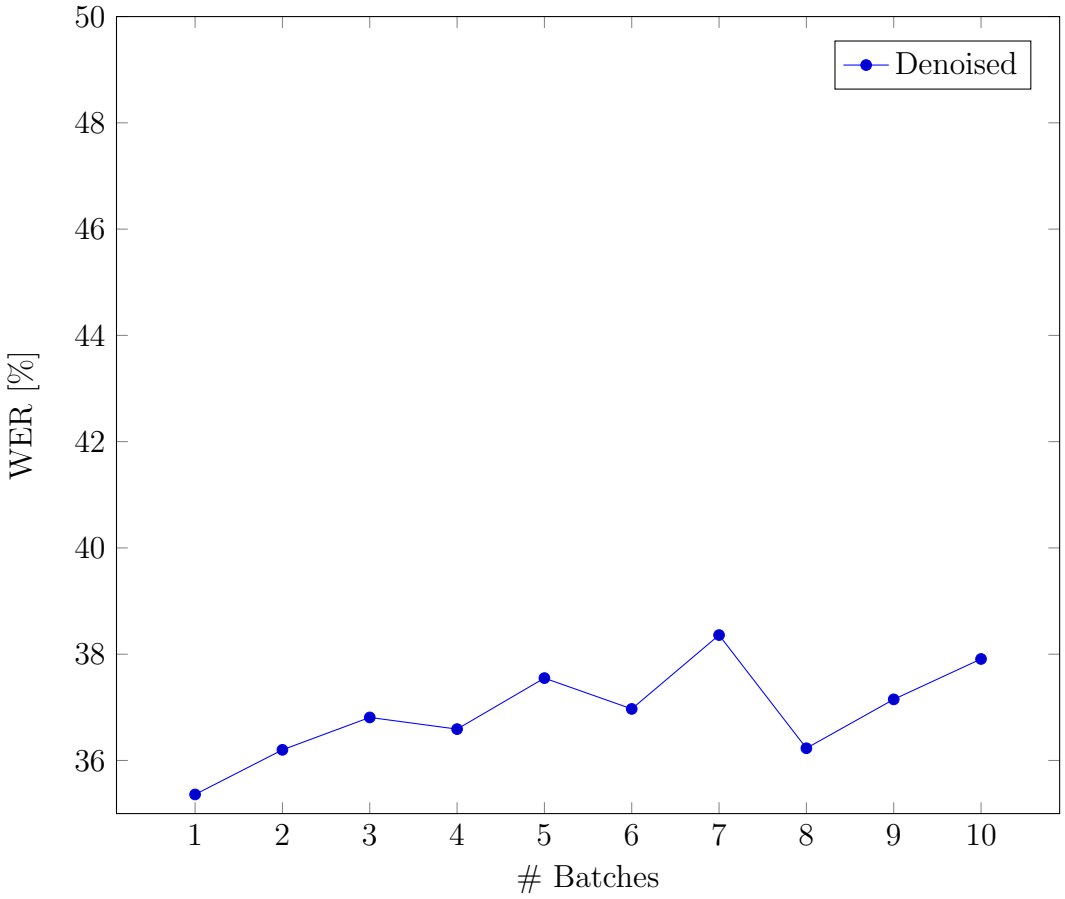

Figure 12: Optimizing the batch size

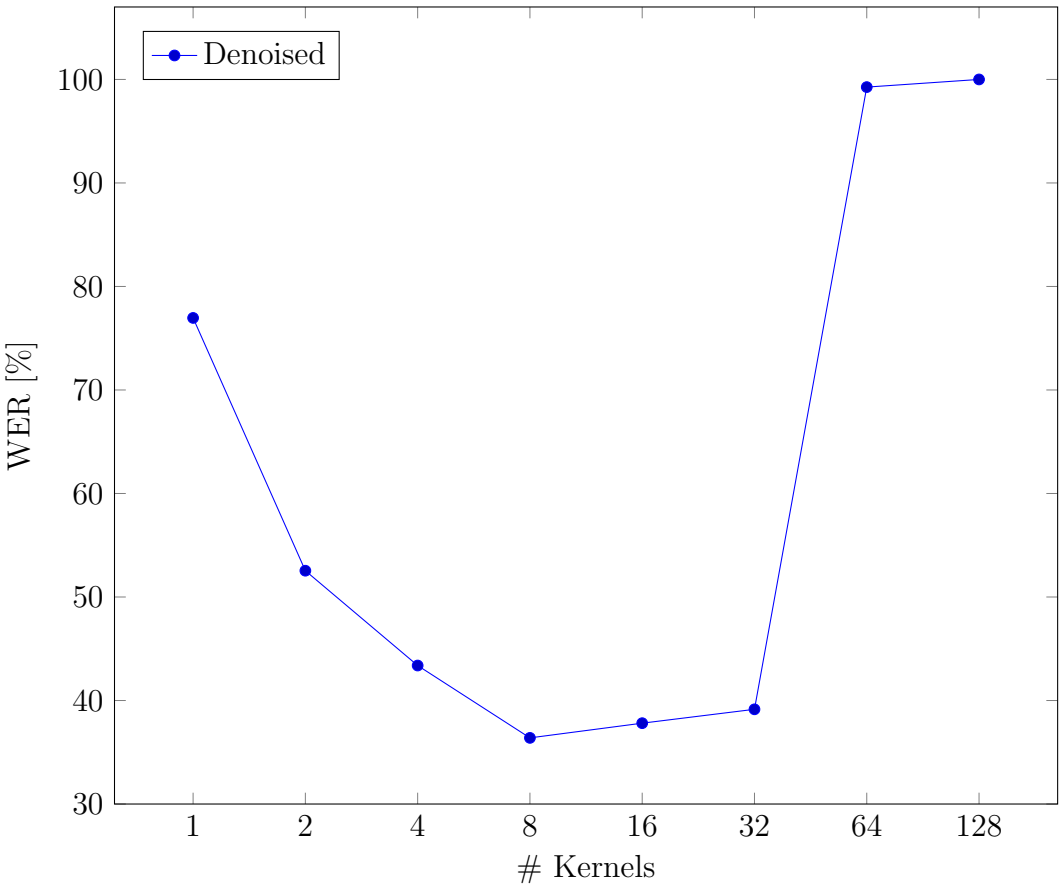

Figure 13: Optimizing the kernel size, exponential step size

rising number of kernels used in the CNN the WER decreases until reaching a minima on 14 Kernels per layer. By further increasing the kernel count the WER increased again and showed erratic movements. We assume that the networks with low numbers of kernels were too small to learn proper speech denoising and the big networks were too big and started to overfit.

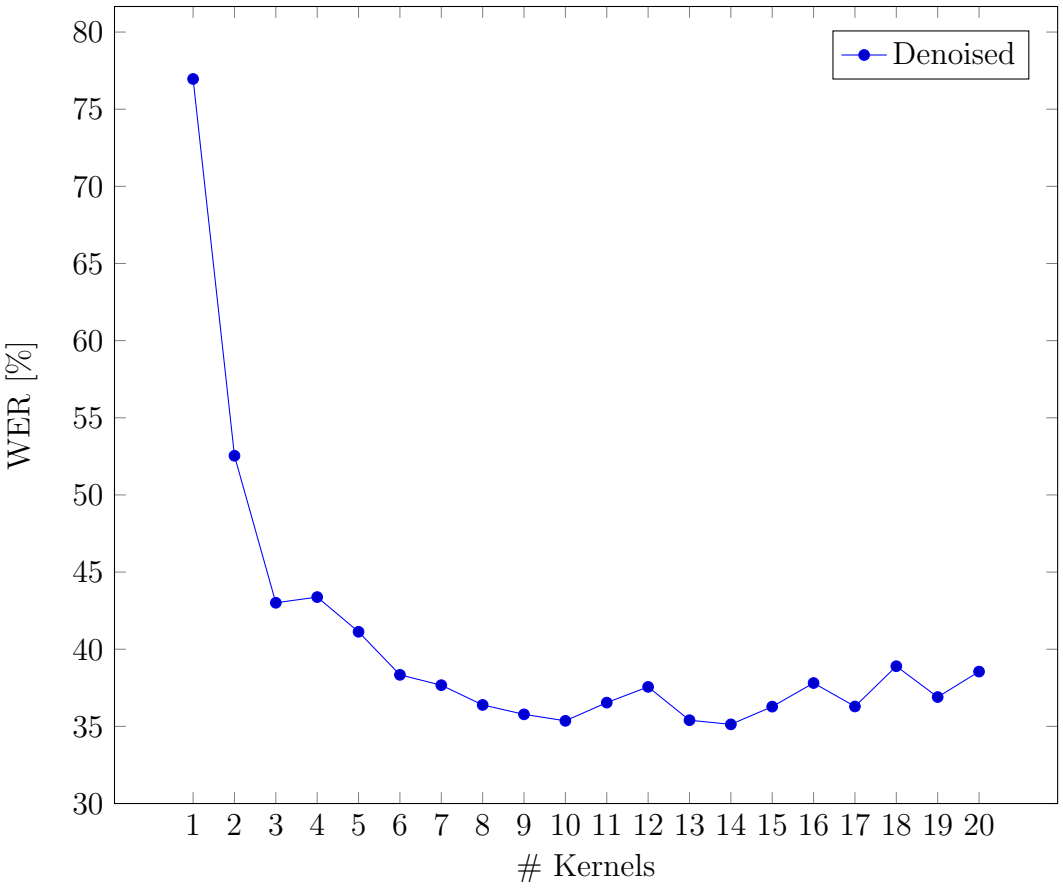

Figure 14: Optimizing the kernel size

3.4.4 Optimizer

The last optimization step was on the optimizers. We wanted to try out to further improve our results by training the denoising system with different optimization functions. For explanation on the methods see Section 2.2.6. All function parameters were left at the preset values of Keras. The results were underwhelming and are listed in Table 2. Only the originally used Adadelta optimizer created decent results. The ASR denoising system with other optimization functions were either by far worse than the Adadelta setup or failed.
We could only think of one reason why the different methods failed to create proper results. We assume that the hyper parameters of the optimizer themselves needed optimization.
The final denoising CNN after the optimization process is depicted in Figure 15.

Optimizer	Adadelta	RMSProp	Adam	AdaMax	Nadam
Average WER [%]	35.13	90.6	100	50.13	99.26

Table 2: Optimizing the optimizer on the validation set

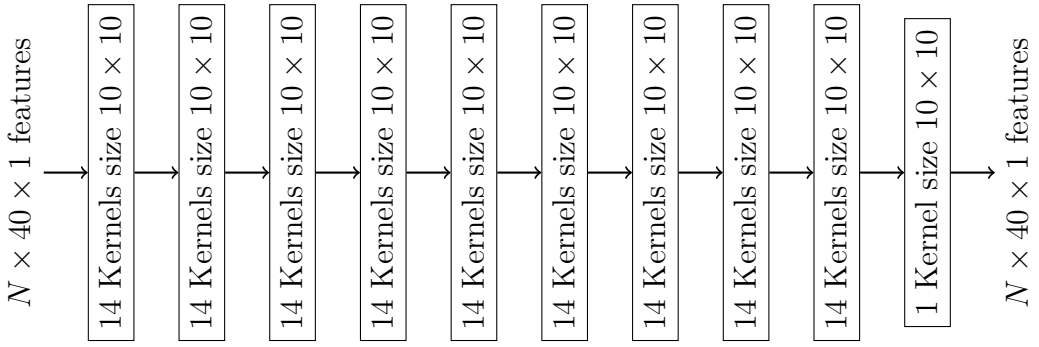

Figure 15: The final denoising CNN after the optimization process

3.5 Evaluation

In this section we will present the experimental results. The final results of the validation set are visualized in Figure 16 and the final results of the test set are visualized in Figure 17. The two graphs are strongly similar verifying that the denoising system most likely did not overfit. In the following we will only inspect the more important results of the evaluation.
On all SNRs of 18 dB and lower our CNN based denoising ASR system created the best results. On a SNR of 6 dB our denoising system decreased the recognition WER by an absolute value of more than 20 % transforming a strongly noised speech signal into a usable state. And on a SNR of 12 dB the WER was decreased from 23.78 % to 15.08 %. Thus the denoising CNN managed to vastly increase the audio quality and intelligibility on moderate SNRs. The system also vastly decreased the WER on 0 dB but unfortunately even after improving the WER by an absolute value of 27.8 % the resulting speech signal was not good enough to be used in most real world applications. Additionally the CNN denoising setup is the only ASR system able to decrease the WER on a SNR of -6 dB to less than 100 %.
Of the three ASR system built for robust speech recognition the denoising CNN creates the best results on all SNRs including the clean data. Only the clean ASR system gives better results on the clean and high SNR speech signals. But this was to be expected, because the denoising system learned to map noisy audio data onto clean one, which made it nearly impossible for our system to improve the clean data. Even though on a SNR of 24 dB or 30 dB our denoising systems WER is only a single percentage worsen than the clean setup. Only for clean speech our system looses by a WER difference of 2.39 %, which is acceptable most of the time.
The conclusion of these results is that CNN denoising works well on low and moderate SNRs while at the same time only marginally decreasing an ASRs performance on high SNRs.
One possible reason why the denoising setup was observably worse on the clean data in comparison to the high SNR data may be visible in Figure 18. Even a high SNR value of 30 dB contains noises visibly influencing the audio spectrum. The denoising system was trained on an uniform distribution of eight SNR levels including the clean audio. Our guess is that the training of the denoising system got strongly influenced by the overrepresented noisy speech signals. Therefore the system became noticeable worse at handling clean speech signals.

Another mentionable result of our experiments are the clearly visible improvements created by the denoising system and visualized in Figure 18. Especially on SNRs of 6 dB and 18 dB the noisy filter bank features contained only slightly visible speech characteristics and after the denoising process the speech signal was restored. The graphic additionally visualizes that only slight changes occurred on the clean feature matrices.

Speech-Noise-Ratio	clean	30	24	18	12	6	0	-6
Clean ASR	11.99	12.12	13.72	17.42	29.95	60.23	92.83	102.90
Noise tr. ASR	18.38	17.50	17.18	18.39	23.89	41.30	78.59	117.43
Denoise CNN ASR	15.35	14.56	14.94	16.74	22.12	36.71	68.05	92.53
Denoise tr. ASR	14.65	14.91	15.47	17.84	26.49	48.94	83.73	107.80

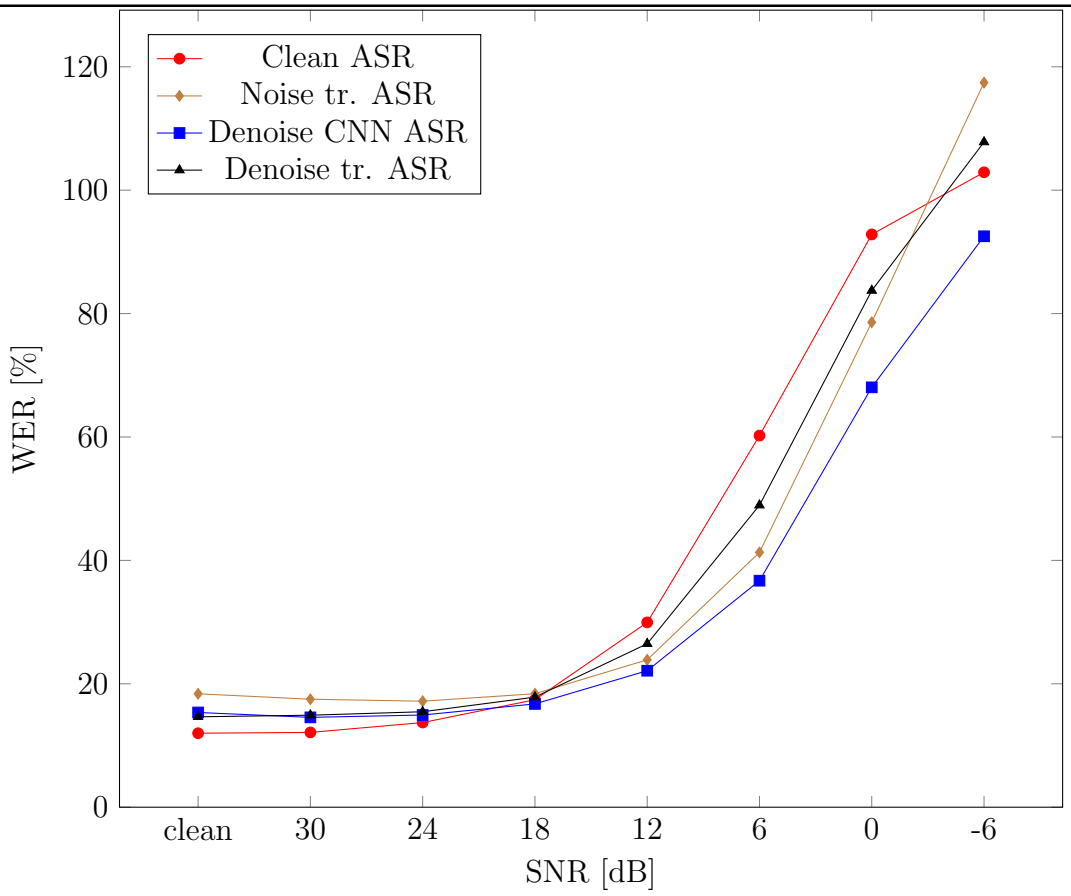

Figure 16: Results of the four ASR systems on the validation set

Speech-Noise-Ratio	clean	30	24	18	12	6	0	-6
Clean ASR	7.76	8.44	9.57	13.11	23.78	46.93	84.83	100.04
Noise tr. ASR	12.10	11.02	11.02	12.42	15.52	27.50	62.13	106.50
Denoise CNN ASR	10.15	9.87	10.49	11.73	15.08	26.05	57.03	87.91
Denoise tr. ASR	11.38	10.54	11.02	13.52	19.32	35.74	70.53	101.38

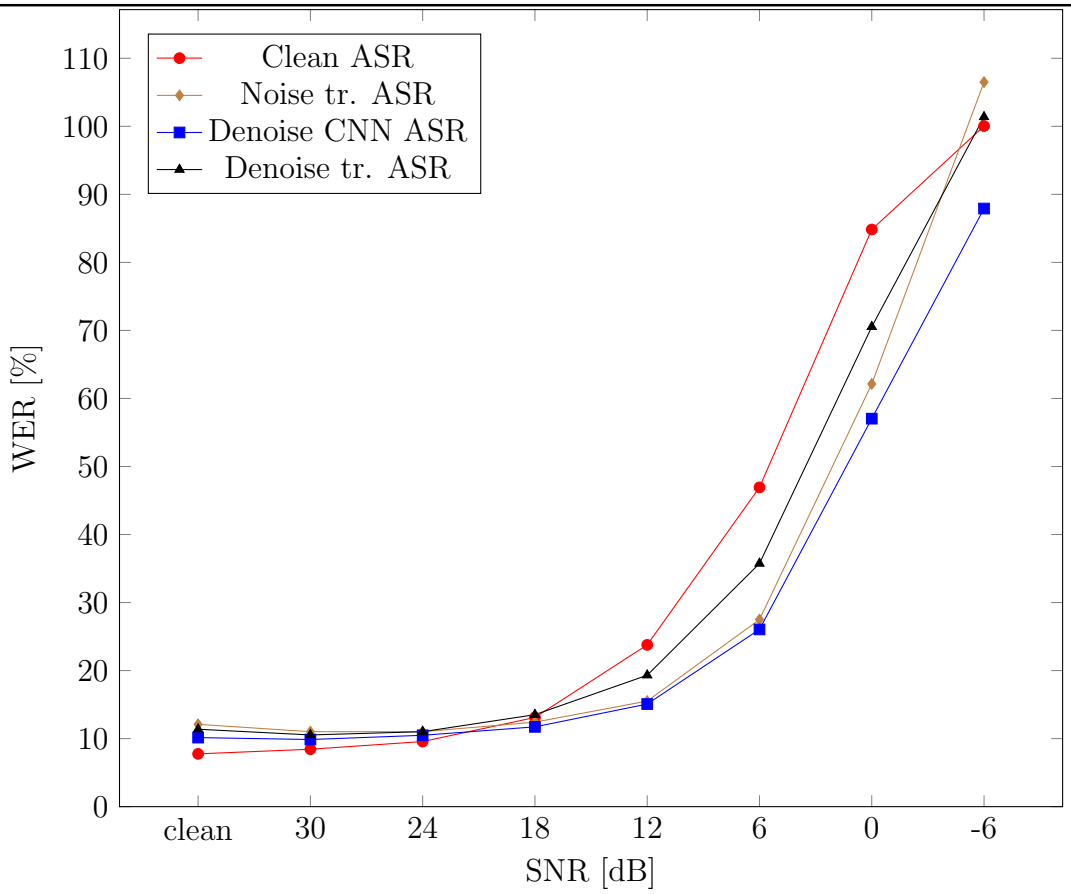

Figure 17: Results of the four ASR systems on the test set

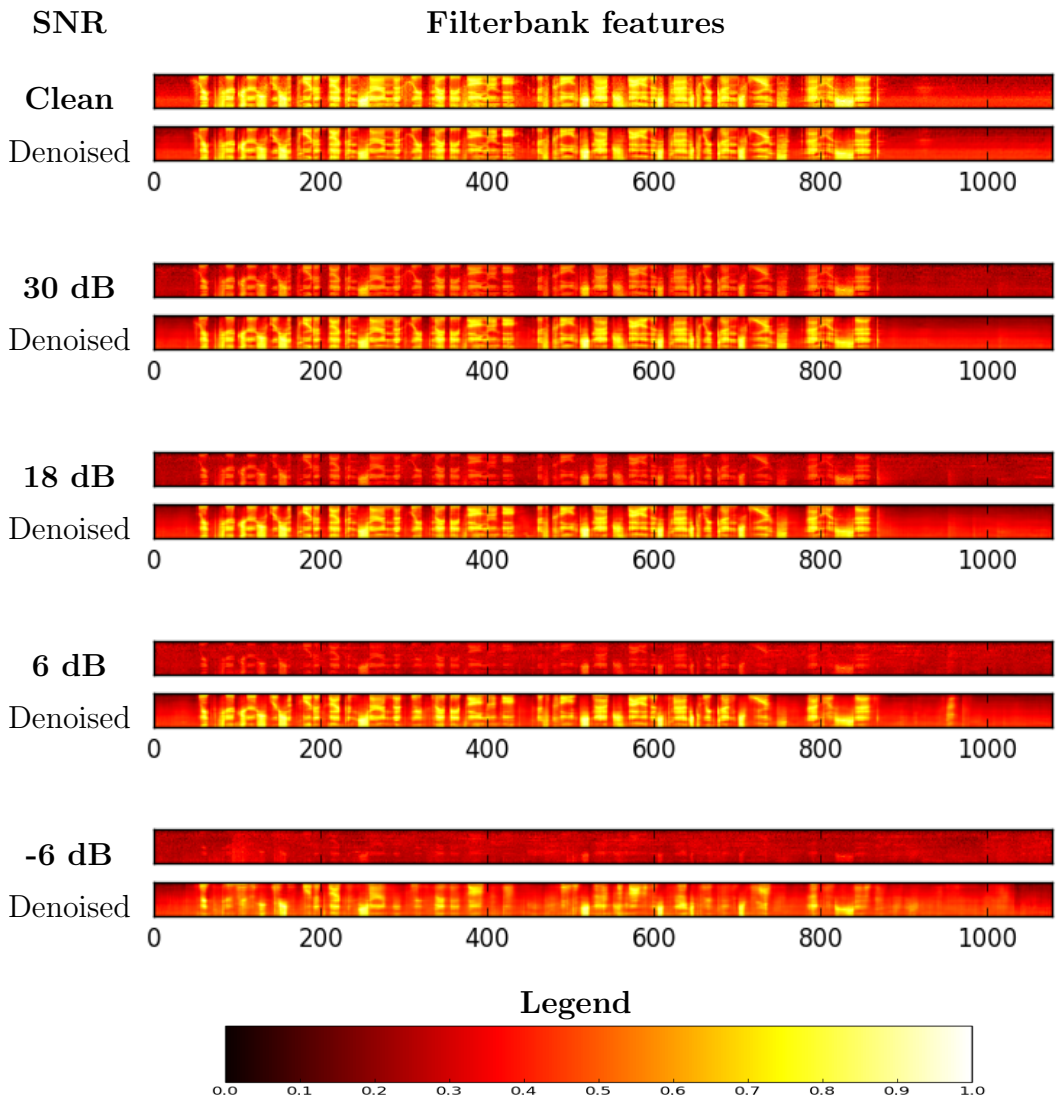

Figure 18: Normalized filter bank features on different SNRs noisy and denoised. The original speech is utterance 440c040a of the WSJ eval_92 set.

3.6 Generalization

Two of the major problems of our experimental setup were that we only used a single noise type for the training and we tested our setup on the exact same noise. Therefore it was possible that the denoising CNN only learned to remove the exact noise pattern of our training noise. To prove that our network is a general working denoising system we used the 4^{th} CHiME speech seperation and recognition challenge[4]. Hereby we wanted to prove two things.

1. The denoising system generalizes onto different types of noises
2. The denoising system generalizes onto real world problems

To achieve that goal, we prepared our dataset, trained and tested the ASR and denoising system the same way as before, described in Sections 3.2 and 3.3. The only difference was that we used all eight hours of the CHiME background audio recorded on four different locations. The locations were a in a café, on a street, on a bus and in a pedestrian area.
The result of this experiment, visualized in Figure 19, clearly show the generalization of the denoising CNN on different noise types. They are similar to the results for pedestrian noise in Figure 17. On a SNR of 24 dB and higher the clean ASR system created the best results but starting at 18 dB and lower our denoising system noticeably improves the WER. On an SNR of 12 dB the WER is increased by an absolute value of more than 10 % and on a SNR of 6 dB even more than 20 %. Interestingly the denoising system trained on four noises creates a little bit worse results on clean and low SNRs but remarkably better results on high SNRs than the CNN trained only on pedestrian noise. Therefore the denoising system generalizes onto different types of noises.

But even by generalizing our denoising system onto four types of noise instead of a single one, we still used the same noise signals for training and testing. To solve this problem we decided to test our algorithm on real world data. Therefore we decided to use the real dataset of the CHiME challenge. The real dataset consists of 1320 utterances from four speakers recorded in four different noisy locations. This means the test set was not artificially created by mixing speech and noise, but instead consisted of real world audio recorded with six microphones on one tablet. Also the four speakers did not overlap with any speakers of the training or validation set. This means neither

the noise nor the speech of the training and testing set overlapped in any way. Only the four locations were the same as used for the training, in a café, on a street, on a bus and in a pedestrian area. The results of this experiment were promising. The clean ASR system achieved a WER of 86.31 % and that results was improved by the denoising system by more than an absolute of 20 % to reach a WER of 65.91 %. Therefore the denoising system not only generalizes to unknown speaker and noises but it additionally generalizes to real world data. We assume that the results of the CHiME challenge are by far worse than the results of the artificially created data, because the CHiME challenge had a different method for audio recording. The WSJ speech used for the training process was recorded with a professional microphone in a quiet environment. The CHiME audio in opposite was recorded with six comparably low quality microphones. Sometimes speaker held their finger on a microphone or it was blocked by obstacles. The recording method created attenuations and reverberations. Since the original ASR was not trained to handle bad quality speech the WER on the CHiME challenge dataset was noticeably worse.

The results of these experiments were that the CNN denoising system generalizes to different speaker, noises and to a real world environment. In all experiments the CNN managed to vastly improve the speech recognition performance on moderate and low SNRs.
For a final overview we presented the results of the three important experiments, the evaluation on the pedestrian noised data, on the data noised with four types of noises and the results of the CHiME challenge, in Table 3. All results were given as average WER in %.

System / Dataset	Clean ASR	Denoise CNN ASR	Noise tr. ASR	Denoise tr. ASR
Pedestrian	32.72 %	25.37 %	28.69 %	30.38 %
Four noises	31.45 %	22.04	-	-
Real CHiME	86.31 %	65.91	-	-

Table 3: Results of the four ASR systems on the test set containing pedestrian noise only, the test set containing all four noise types and the real world test set of the CHiME challenge. All results are the average WER in %.

Speech-Noise-Ratio	clean	30	24	18	12	6	0	-6
Clean ASR	7.76	8.03	9.69	13.77	24.93	47.86	76.86	94.19
Denoise CNN ASR	10.51	10.10	10.35	11.54	14.78	24.26	44.87	71.97

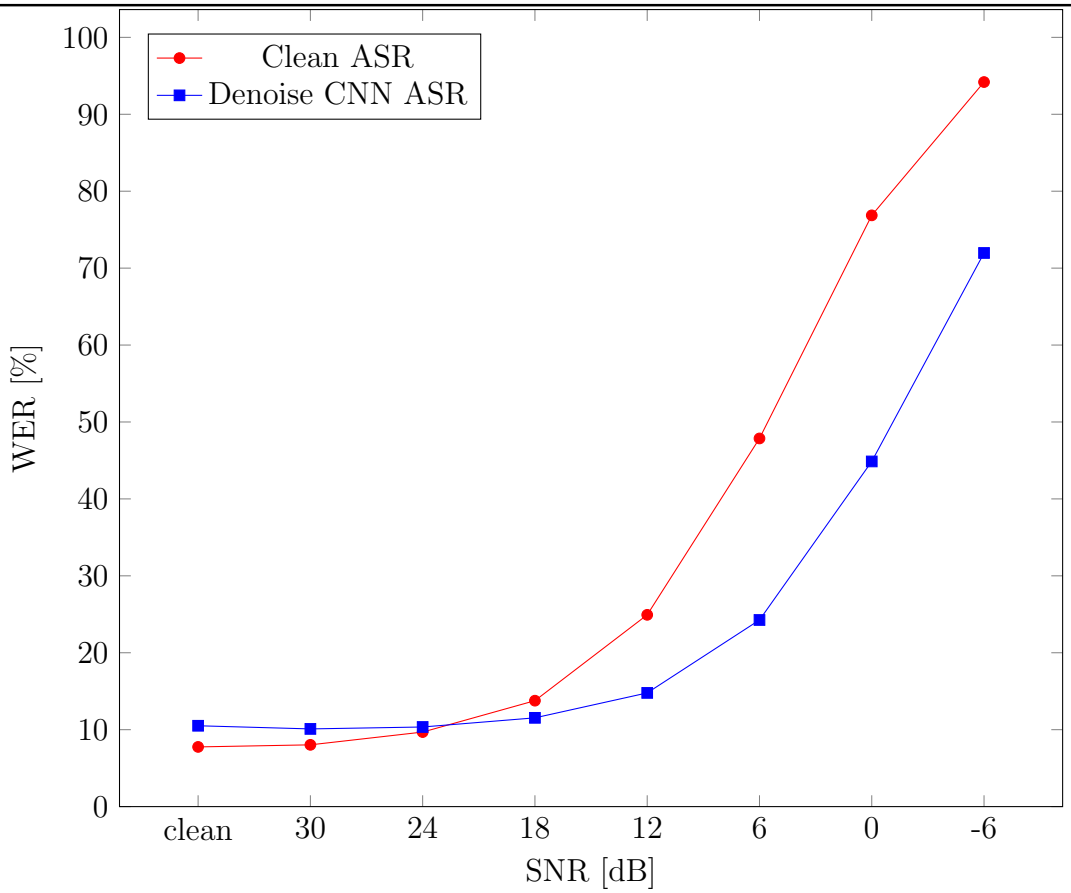

Figure 19: Results of the clean and denoised ASR systems trained and evaluated with four types of noises on the test set

4 Conclusion and outlook

In this thesis we investigated the usage of CNNs as SE method on the feature level of noisy speech data. The motivation for our experiments were past studies in which CNNs were used as improved speech recognizer, speech enhancement system on the spectral level or image denoising system.
For the experiments we built a SE pipeline by adding a denoising system into the WSJ example of the Eesen toolkit. We created noised speech by mixing the WSJ corpus with noise files of the 4^{th} CHiME challenge on different SNRs. Afterwards we trained a fully convolutional neural network implemented in Keras as denoising system to map noisy filter bank features onto clean ones. The denoising system was optimized using a validation set and evaluated using a test set of the WSJ corpus.
For the evaluation we compared the CNN denoising system with a speech recognizer trained on clean speech, an ASR system trained on noisy speech and an ASR system trained on denoised speech features. In the experiments the denoising CNN outperformed the ASR trained on noisy and denoised feature vectors. The denoising approach created better results than the other two systems on all datasets and all SNRs. The CNN setup also outperformed the clean setup on a SNR of 18 dB and below. Only on a SNR of 24 dB and higher the clean speech recognition system expectedly performed slightly better than our denoising setup. On an SNR of 6 dB the CNN system managed to decrease the WER of previously noisy data by an absolute value of more than 20 % from previous 46.93 % down to 26.05 %, improving the ASR output by a large margin. Also on a SNR of 12 dB the CNN manages to decrease the WER by an absolute value of more than 8 %.
Additionally we tested our system on a generalized setup, training it with four different types of noises instead of previously a single noise type. The results of those experiments were partly even better than the results on pedestrian noise, decreasing the average WER of the four noise type dataset by more than 9 % from 31.45 % down to 22.04 %. With this result we proved that our denoising system generalizes onto different types of noises.
We additionally tested the denoising CNN on the CHiME real world dataset proving that the CNN also generalizes onto real world data. On the dataset the denoising system decreased the WER of the clean speech recognizer from 86.31 % down to 65.91 %.

All in all the results of our thesis reach the best case scenario. At the

beginning of our experiments using CNNs for SE was only a conceptual idea. In this thesis we proved that this concept works and is very promising. Even an easy CNN without much optimization performs state-of-the-art SE while at the same time generalizes to different types of noises and even real world data. Therefore we guess that CNNs have a bright future and much room for further improvements in ASR.

For the future built up on this thesis we have some suggestions. For example one could try out if the CNN also generalizes onto different feature types. We would like to see if our CNN or a similar CNN can be used for SE on MFCCs. Another thing is that we did not have enough time for a layer size optimization. We would like to know how much optimizing the different layer sizes can increase the performance of the denoising system. Another factor we are interested in, is the real performance of the denoising CNN on the CHiME challenge with the CHiME baseline speech recognizer. We would like to know how far our system can compete with the other challengers. Also interesting would be to train and test the CNN on multi channel audio, which could be easily implemented, because CNNs are perfectly suited for multi channel input.

Finally we hope that this thesis gave a good introduction into feature denoising using CNNs for noisy speech recognition and maybe the promising results encourage more studies into this field of research.

References

[1] Yong Xu, Jun Du, Li-Rong Dai, and Chin-Hui Lee. A regression approach to speech enhancement based on deep neural networks. *IEEE/ACM Transactions on Audio, Speech, and Language Processing, vol. 23 no. 1 pp. 7-19*, 2015.

[2] Y Ephraim and D Malah. Speech enhancement using a minimum mean-square error log-spectral amplitude estimator. *IEEE Trans. Acoustics, Speech and Signal Processing, vol. ASSP-33, pp. 443-445*, 1985.

[3] Jun Du, Yan-Hui Tu, Lei Sun, et al. The ustc-iflytek system for chime-4 challenge. *The 4th CHiME Speech Seperation and Recognition Challenge*, 2014.

[4] Emmanuel Vincent, Shinji Watanabe, Jon Barker, and Ricard Marxer. An analysis of environment, microphone and data simulation mismatches in robust speech recognition. *Computer Speech and Language*, 2016.

[5] Ying Zhang, Pezeshki Mohammad, Philémon Brakel, et al. Towards end-to-end speech recognition with deep convolutional neural networks. *arXiv:1701.02720v1 [cs.CL]*, 2017.

[6] Lovedeep Gondara. Medical image denoising using convolutional denoising autoencoders. *arXiv:1608.04667v2 [cs.CV]*, 2016.

[7] Kai Zhang, Wangmeng Zuo, Yunjin Chen, Deyu Meng, and Lei Zhang. Beyond a gaussian denoiser: Residual learning of deep cnn for image denoising. *arXiv:1608.03981v1 [cs.CV]*, 2016.

[8] Mike Kayser and Victor Zhong. Denoising convolutional autoencoders for noisy speech recognition. *Stanford University*, 2015.

[9] Se Rim Park and Jin Won Lee. A fully convolutional neural network for speech enhancement. *arXiv:1609.07132v1 [cs.LG]*, 2016.

[10] Yajie Miao, Mohammad Gowayyed, and Florian Metze. Eesen: End-to-end speech recognition using deep rnn models and wfst-based decoding. *arXiv:1507.08240v3 [cs.CL]*, 2015.

[11] John Garofolo, David Graff, Doug Paul, and David Pallett. Csr-i (wsj0) sennheiser ldc93s6b. *Philadelphia: Linguistic Data Consortium*, 1993.

[12] John Garofolo, David Graff, Doug Paul, and David Pallett. Csr-ii (wsj1) sennheiser ldc94s13b. *Philadelphia: Linguistic Data Consortium*, 1994.

[13] Martín Abadi, Ashish Agarwal, Paul Barham, et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[14] Clause E. Shannon. A mathematical theory of communications. *Bell System Technical Journal. 27*, 1948.

[15] Ryan J. Cassidy and Julius O. Smith III. Auditory filter bank lab. `https://ccrma.stanford.edu/realsimple/aud_fb/What_Filter_Bank.html`, 2008. Accessed: 08.09.2017.

[16] Todd K. Moon. Lecture 9: Aliasing and leakage. `http://ocw.usu.edu/Electrical_and_Computer_Engineering/Signals_and_Systems/9_2node2.html`, 2011. Accessed: 08.09.2017.

[17] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation, vol. 9 no. 8, pp. 1735-1780*, 1997.

[18] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. *Neural computation, vol. 9 no. 8, pp. 1735-1780*, 1997.

[19] Mehryar Mohri, Fernando Pereira, and Michael Rifley. Weighted finite-state transducers in speech recognition. *Computer Speech and Language, vol. 20, no. 1, pp. 333-336*, 2002.

[20] Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. OpenFst: A general and efficient weighted finite-state transducer library. In *Proceedings of the Ninth International Conference on Implementation and Application of Automata, (CIAA 2007)*, volume 4783 of *Lecture Notes in Computer Science*, pages 11–23. Springer, 2007. `http://www.openfst.org`.

[21] Jens Fröschel. Feature denoising using cnns for noisy speech recognition - project work, 2017.

[22] Kurt Hornik. Approximation capabilities of muitilayer feedforward networks. *Neural Networks, vol. 4 pp. 251-257*, 1990.

[23] François Chollet. Keras. `https://github.com/fchollet/keras`, 2015. Accessed: 08.09.2017.

[24] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv:1609.04747v2 [cs.LG]*, 2017.

[25] CNTK Development Team. An Introduction to Computational Networks and the Computational Network Toolkit. *Microsoft Technical Report MSR-TR-2014-112*, 2014.

[26] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[27] François Chollet. Keras documentation. `https://keras.io/`, 2015. Accessed: 08.09.2017.

[28] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, et al. The kaldi speech recognition toolkit. In *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*. IEEE Signal Processing Society, December 2011. IEEE Catalog, no. CFP11SRW-USB.

[29] Linguistic data consortium. `https://www.ldc.upenn.edu/`, 1992. Accessed: 08.09.2017.

[30] Norbert Wiener. Extrapolation, interpolation, and smoothing of stationary time series. *The MIT press*, 1949.

[31] Jacob Benesty, Jingdong Chen, Yiteng (Arden) Huang, and Simon Doclo. *Study of the Wiener Filter for Noise Reduction*, pages 9–41. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[32] S. Boll. Suppression of acoustic noise in speech using spectral subtraction. *IEEE Transactions on Acoustics, Speech, and Signal Processing , vol. 27 no. 2 pp. 113-120*, 1979.

[33] Anuradha R. Fukane and Shashikant L. Sahare. Enhancement of noisy speech signals for hearing aids. *International Conference on Communication Systems and Network Technologies*, 2011.

[34] Shambhu Shankar Bharti, Manish Gupta, and Suneeta Agarwal. A new spectral subtraction method for speech enhancement using adaptive noise estimation. *3rd International Conference on Recent Advances in Information Technology (RAIT)*, 2016.

[35] Mihnea Radu Udrea, Nicolae Dragos Vizireanu, and Silviu Ciochina. An improved spectral subtraction method for speech enhancement using a perceptual weighting filter. *Digital Signal Processing, vol. 18 no. 4 pp. 581-587*, 2008.

[36] Yariv Ephraim and Harry L. Van Trees. A new spectral subtraction method for speech enhancement using adaptive noise estimation. *IEEE Transactions on Speech and Audio Processing, vol. 3 no. 4*, 1995.

[37] Adam Borowicz. A signal subspace approach to spatio-temporal prediction for multichannel speech enhancement. *EURASIP Journal on Audio, Speech, and Music Processing*, 2015.

[38] Yiteng Huang, Jacob Benesty, and Jingdong Chen. Analysis and comparison of multichannel noise reduction methods in a common framework. *IEEE Transactions on Audio, Speech and Language Processing, Vol. 16 No. 5*, 2008.

[39] Felix Weninger, Hakan Erdogan, Shinji Watanabe, et al. Speech enhancement with lstm recurrent neural networks and its application to noise-robust asr. *12th International Conference on Latent Variable Analysis and Signal Separation (LVA/ICA)*, 2015.

[40] Jitong Chen, Yuxuan Wang, Sarah E. Yoho, DeLiang Wang, and Eric W. Healy. Large-scale training to increase speech intelligibility for hearing-impaired listeners in novel noises. *The Journal of the Acoustical Society of America, vol. 139 pp. 2604-2612*, 2016.

[41] Ossama Abdel-Hamid, Abdel-rahman Mohamed, Hui Jiang, et al. Convolutional neural networks for speech recognition. *IEEE/ACM Transactions on audio, speech, and language processing, vol. 22, no. 10*, 2014.

[42] John S. Garofolo, Lori F. Lamel, William M. Fisher, et al. Timit acoustic-phonetic continous speech corpus. *Philadelphia: Linguistic Data Consortium LDC93S1*, 1993.

[43] Dimitri Palaz, Mathew Magimai.-Doss, and Ronan Collobert. Analysis of CNN-based Speech Recognition System using Raw Speech as Input. In *Proceedings of Interspeech*, 2015.

[44] Hans-Günter Hirsch. Aurora speech recognition experimental framework. `http://aurora.hsnr.de/index-2.html`, 2006. Accessed: 08.09.2017.

[45] Szu-Wei Fu, Yu Tsao, Xugang Lu, and Hisashi Kawai. Raw waveform-based speech enhancement by fully convolutional networks. *arXiv:1609.07132v1 [cs.LG]*, 2016.

[46] Zixing Zhang, Jürgen Geiger, Jouni Pohjalanainen, et al. Deep learning for environmentally robust speech recognition: An overview of recent developments. *arXiv:1705.10874v2 [cs.SD]*, 2017.

[47] The NIST file format: waveforms (.wv1, .wv2). `http://svr-www.eng.cam.ac.uk/reports/ajr/TR192/node11.html`, 1995. Accessed: 17.07.2017.

[48] Sphere conversion tools. `https://www.ldc.upenn.edu/language-resources/tools/sphere-conversion-tools`. Accessed: 08.09.2017.

[49] Chris Bagwell et al. Sound eXchange. `http://sox.sourceforge.net/`, 1991. Accessed: 20.07.2017.

[50] P. Pujol, D. Macho, and C. Nadeu. On real-time mean-and-variance normalization of speech recognition features. *IEEE International Conference on Acoustics Speech and Signal Processing Proceedings*, 2006.

[51] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science and Engineering, Vol. 13, pp. 22-30*, 2011.