Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Master Thesis

Real-time speech separation with deep attractor networks on an embedded system

Marc Siemering

marc.siemering@gmail.com

MIN-Fakultät
Fachbereich Informatik
Arbeitsbereich Signalverarbeitung

Studiengang:	Informatik
Matrikelnummer:	7118655
Abgabedatum:	27.11.2020

Erstgutachter:	Prof. Dr.-Ing. Timo Gerkmann
Zweitgutachter:	M.Sc. David Ditter
Betreuer:	M.Sc. David Ditter

Abstract

In this work, we investigate the applicability of the Online Deep Attractor Network (ODANet) for real-time speech separation on an embedded system with limited processing resources. To optimize the ODANet for a resource constrained environment, we extensively evaluate two different reduction methods. First, we present a detailed analysis of complexity reduction via hyper-parameter tuning of the ODANet and second, we introduce a compression method for long short-term memory (LSTM) layers to the ODANet architecture. While our results suggest that real-time capability is possible for a desktop computer with these methods, it is not achievable for an embedded device like the NVIDIA Jetson Nano while maintaining an acceptable separation performance. In further findings, we show that the utilized compression method for LSTMs is superior to hyper-parameter tuning in terms of finding a good trade-off between low processing time and separation performance. Furthermore, we want to highlight that this work is the first to our knowledge to give an extensive description of a singular value decomposition based compression method for LSTMs including an open-source implementation available at `https://github.com/sp-uhh/compressed-lstm`.

Contents

Acronyms

ADAM	Adaptive Moment Estimation
ADANet	Anchored Deep Attractor Network
AECNN	Auto-Encoder Convolutional Neural Network
BLSTM	bidirectional long short-term memory
CLSTM	compressed long short-term memory
CNN	convolutional neural network
CODANet	Compressed Online Deep Attractor Network
CPU	central processing unit
DANet	Deep Attractor Network
dB	decibel
DNN	deep neural network
DPCL	Deep Clustering
EM	expectation maximisation
FFT	fast Fourier transform
FP16	16 bit floating point
GFLOPS	giga floating point operations per second
GPU	graphics processing unit
IBM	ideal binary mask
IFFT	inverse fast Fourier transform
IRM	ideal ratio mask
ISTFT	inverse short-time Fourier transform
LPDDR	low-power double data rate synchronous dynamic random access memory
LSTM	long short-term memory
LVCSR	large vocabulary conversational speech recognition
MB	megabyte
ML	machine learning
MSE	mean squared error
NN	neural network
ODANet	Online Deep Attractor Network
PC	personal computer
PESQ	perceptual evaluation of speech quality score
PIT	Permutation Invariant Training

RMSprop	Root Mean Square Propagation
RNN	recurrent neural network
SAR	sources-to-artifacts ratio
SDR	source-to-distortion ratio
SI-SNR	scale-invariant source-to-noise ratio
SIR	source-to-interferences ratio
SNR	sources-to-noise ratio
STFT	short-time Fourier transform
SVD	singular value decomposition
TASNet	Time-domain Audio Separation Network
uPIT	utterance-level Permutation Invariant Training
WER	word error rate
WFM	'wiener-filter' like mask
WSJ0	Wall Street Journal

List of Figures

List of Tables

List of Symbols

Symbol	Description	Dimension
$\mathbf{a}_i$	Attractor point for speaker i	$\mathbb{R}^{1\times K}$
$\mathbf{a}_{t-1,i}$	Attractor point for speaker i at time $t-1$	$\mathbb{R}^{1\times K}$
$\mathbf{A}_p$	Attractor points for combination p (ADANet)	$\mathbb{R}^{C\times K}$
$\mathbf{b}^l$	Bias of layer l	$\mathbb{R}^{1\times N^l}$
$\mathbf{b}_\mathrm{f}$	Training parameter of ODANet's dynamic weighting	$\mathbb{R}^{1\times K}$
$\mathbf{b}_\mathrm{g}$	Training parameter of ODANet's dynamic weighting	$\mathbb{R}^{1\times K}$
C	Number of Speakers	$\mathbb{N}$
$\mathcal{C}$	Cost function	
$\mathbf{D}$	Distance between embedding points and attractor points (DANet)	$\mathbb{R}^{C\times FT}$
$\mathbf{d}_i$	Distance between embedding points and the attractor point of speaker i	$\mathbb{R}^{1\times FT}$
$\mathbf{D}_p$	Distance between embedding points and anchor point combination p (ADANet)	$\mathbb{R}^{C\times FT}$
$\mathbf{E}$	Embedding	$\mathbb{R}^{K\times FT}$
$\mathbf{e}_\mathrm{noise}$	Error signal in time domain for SI-SNR	$\mathbb{R}^{T}$
$\mathbf{e}_\mathrm{artif}$	Artifacts error signal in time domain for SDR	$\mathbb{R}^{T}$
$\mathbf{e}_\mathrm{interf}$	Interference error signal in time domain caused by e.g. other speakers for SDR	$\mathbb{R}^{T}$
$\mathbf{e}^*_\mathrm{noise}$	Noise error signal in time domain caused by sensor noise for SDR	$\mathbb{R}^{T}$
f	Frequency index	$\{1, 2, \ldots, F\}$

Symbol	Description	Dimension
F	Number of frequency bins	$\mathbb{N}$
$\mathbf{\Gamma}_p$	Similarity matrix containing the pairwise similarity between the attractor points in the combination p	$\mathbb{R}^{C\times C}$
$\mathbf{\Gamma}_{p_{i,j}}$	Element in row i and column j of similarity matrix $\mathbf{\Gamma}_p$	$\mathbb{R}$
γ_p	Maximal pairwise similarity between the attractor points in the combination p	$\mathbb{R}$
$\mathbf{h}_t^l$	Output of the l-th layer	$\mathbb{R}^{1\times N^l}$
$\widetilde{\mathbf{h}}_t^l$	Projected/compressed output of the l-th layer	$\mathbb{R}^{1\times r^l}$
j	Anchor point index	$\{1, 2, \dots N\}$
$\mathbf{J}_\mathrm{f}$	Training parameter of ODANet's dynamic weighting	$\mathbb{R}^{K\times K}$
$\mathbf{J}_\mathrm{g}$	Training parameter of ODANet's dynamic weighting	$\mathbb{R}^{K\times K}$
K	Embedding dimension	$\mathbb{N}$
l	Layer index	$\mathbb{N}$
$\mathcal{L}$	Loss function	
λ	Compression threshold	$[0, 1]$
$\mathbf{M}$	Masks assign each time-frequency point to the sources in proportion to its share in the mixture	$[0, 1]^{C\times FT}$
$\widehat{\mathbf{M}}$	Estimated masks	$[0, 1]^{C\times FT}$
$\widehat{\mathbf{m}}_i$	Estimated mask for speaker i	$[0, 1]^{1\times FT}$
$\boldsymbol{\mu}$	Mean of K-dimensional normal distribution	$\mathbb{R}^K$
N	Number of Anchor points	$\mathbb{N}$
N_l	Number of units in layer l	$\mathbb{N}$
$\mathcal{N}_K$	K-dimensional normal distribution	
o	Number of operation of a general RNN	$\mathbb{N}$
$\tilde{o}$	Number of operation of a compressed RNN in general	$\mathbb{N}$
o^*	Number of operation of a general LSTM	$\mathbb{N}$

Symbol	Description	Dimension
$\tilde{o}^*$	Number of operation of a compressed LSTM in general	$\mathbb{N}$
$\boldsymbol{\psi}_j$	Anchor point number j	$\mathbb{R}^{1\times K}$
$\boldsymbol{\Psi}_p$	Anchor point combination p	$\mathbb{R}^{C\times K}$
p	Anchor point combination index	$\{1, 2, \dots \binom{N}{C}\}$
$\mathbf{P}^l$	Projection matrix $P^l = \widetilde{U_h^l}\widetilde{\Sigma_h^l}$	$\mathbb{R}^{N^l\times r_l}$
r_l	Rank of layer l	$\mathbb{N}$
$s_i(t)$	True speech signal function of speaker i in time-domain	$\mathbb{R}\to\mathbb{R}$
$\mathbf{s}$	True speech signal array of speaker i in time-domain	$\mathbb{R}^T$
$\hat{\mathbf{s}}$	Estimated speech signal array of speaker i in time-domain	$\mathbb{R}^T$
$\mathbf{s}_{\text{target}}$	Orthogonal projection of $\hat{\mathbf{s}}$ onto $\mathbf{s}$	$\mathbb{R}^T$
$\mathcal{S}_i(f,t)$	True complex spectrogram function of speaker i	$\mathbb{R}^2\to\mathbb{C}$
$\mathcal{S}_i$	True complex spectrogram matrix of speaker i	$\mathbb{C}^{1\times FT}$
$\widehat{\mathcal{S}}_i$	Estimated complex spectrogram matrix of speaker i	$\mathbb{C}^{1\times FT}$
$S_i(f,t)$	True magnitude spectrogram function of speaker i	$\mathbb{R}^2\to\mathbb{R}$
$\mathbf{S}_i$	True magnitude spectrogram matrix of speaker i	$\mathbb{R}^{1\times FT}$
$\mathbf{\Sigma}_h^l$	SVD diagonal matrix containing singular values of $\mathbf{W}_h^l$	$\mathbb{R}^{N^l\times N^l}$
$\widetilde{\mathbf{\Sigma}_h^l}$	Truncated SVD diagonal matrix containing the largest r_l singular values of $\mathbf{W}_h^l$	$\mathbb{R}^{r_l\times r_l}$
$\mathbf{\Sigma}$	Covariance matrix of K-dimensional normal distribution	$\mathbb{R}^{K\times K}$
σ_{j}^l	j-th singular value of SDV with $\sigma_1^l \geq \sigma_2^l \geq \ldots \geq \sigma_{N^l}^l$	$\mathbb{R}$
t	Time index	$\{1, 2, \dots, T\}$
T	Number of time frames / samples	$\mathbb{N}$
τ	Context window size (ODANet)	$\mathbb{N}$
$\mathbf{U}_h^l$	SVD matrix containing left-singular vectors of $\mathbf{W}_h^l$	$\mathbb{R}^{N^l\times N^l}$
$\widetilde{\mathbf{U}_h^l}$	Truncated SVD matrix containing the first r_l left-singular vectors of $\mathbf{W}_h^l$	$\mathbb{R}^{N^l\times r_l}$

Symbol	Description	Dimension
$\mathbf{U}_{\mathrm{f}}$	Training parameter of ODANet's dynamic weighting	$\mathbb{R}^{F\times K}$
$\mathbf{U}_{\mathrm{g}}$	Training parameter of ODANet's dynamic weighting	$\mathbb{R}^{F\times K}$
$\mathbf{V}_h^l$	SVD Matrix containing right-singular vectors of $\mathbf{W}_h^l$	$\mathbb{R}^{N^l\times N^l}$
$\widetilde{\mathbf{V}_h^l}$	Truncated SVD matrix containing the first r_l right-singular vectors of $\mathbf{W}_h^l$	$\mathbb{R}^{r_l\times N^l}$
$\mathbf{W}_x^l$	Kernal of layer l	$\mathbb{R}^{N_{l-1}\times N_l}$
$\mathbf{W}_{ix}^l$	Kernal of LSTM input gate (layer l)	$\mathbb{R}^{N_{l-1}\times N_l}$
$\mathbf{W}_{ox}^l$	Kernal of LSTM output gate (layer l)	$\mathbb{R}^{N_{l-1}\times N_l}$
$\mathbf{W}_{fx}^l$	Kernal of LSTM forget gate (layer l)	$\mathbb{R}^{N_{l-1}\times N_l}$
$\mathbf{W}_{cx}^l$	Kernal of LSTM cell (layer l)	$\mathbb{R}^{N_{l-1}\times N_l}$
$\bar{\mathbf{W}}_x^l$	Combined kernel of LSTM (layer l)	$\mathbb{R}^{N_{l-1}\times 4N_l}$
$\mathbf{W}_h^l$	Recurrent kernal of layer l	$\mathbb{R}^{N_l\times N_l}$
$\mathbf{W}_{ih}^l$	Recurrent kernal of LSTM input gate (layer l)	$\mathbb{R}^{N_l\times N_l}$
$\mathbf{W}_{oh}^l$	Recurrent kernal of LSTM output gate (layer l)	$\mathbb{R}^{N_l\times N_l}$
$\mathbf{W}_{fh}^l$	Recurrent kernal of LSTM forget gate (layer l)	$\mathbb{R}^{N_l\times N_l}$
$\mathbf{W}_{ch}^l$	Recurrent kernal of LSTM cell (layer l)	$\mathbb{R}^{N_l\times N_l}$
$\bar{\mathbf{W}}_h^l$	Combined recurrent kernel of LSTM (layer l)	$\mathbb{R}^{N_l\times 4N_l}$
$\mathbf{W}_{\mathrm{f}}$	Training parameter of ODANet's dynamic weighting	$\mathbb{R}^{N_4\times K}$
$\mathbf{W}_{\mathrm{g}}$	Training parameter of ODANet's dynamic weighting	$\mathbb{R}^{N_4\times K}$
$x(t)$	Mixture signal function in time-domain	$\mathbb{R}\to\mathbb{R}$
$\mathcal{X}(f,t)$	Complex mixture spectrogram function	$\mathbb{R}^2\to\mathbb{C}$
$\mathcal{X}$	Complex mixture spectrogram matrix	$\mathbb{C}^{1\times FT}$
$X(f,t)$	Magnitude mixture spectrogram function	$\mathbb{R}^2\to\mathbb{R}$
$\mathbf{X}$	Magnitude mixture spectrogram matix	$\mathbb{R}^{1\times FT}$
$\mathbf{Y}$	True source assignment	$[0,1]^{C\times FT}$
$\hat{\mathbf{Y}}$	Estimated source assignment	$[0,1]^{C\times FT}$

Symbol	Description	Dimension
$\hat{\mathbf{y}}_i$	Estimated source assignment for speaker i	$[0,1]^{1\times FT}$
$\hat{\mathbf{Y}}_p$	Estimated source assignment for anchor point combination p	$[0,1]^{C\times FT}$
$\mathbf{Z}_h^l$	Recurrent kernal back-projection matrix of layer l	$\mathbb{R}^{r_l\times N^l}$
$\mathbf{Z}_x^{l+1}$	Kernal back-projection matrix of layer $l+1$	$\mathbb{R}^{r_l\times N_{l+1}}$
$\bar{\mathbf{Z}}_x^{l+1}$	Combined back-projection kernel of LSTM (layer l)	$\mathbb{R}^{r_l\times 4N_{l+1}}$
$\bar{\mathbf{Z}}_h^l$	Combined back-projection kernel of LSTM (layer l)	$\mathbb{R}^{r_l\times 4N_l}$

1. Introduction

In real-world speech signal processing applications we often encounter situations where multiple speakers are active in the signal at the same time, but we are actually interested in the speech source signal of one or multiple speakers in the mixture. For this so-called problem of speech separation, great advances have been made by the research community in recent years using neural networks (NNs). Still, real-time processing with NNs requires a lot of computational resources, and so far there is little research like [1] considering the deployment of speech separation algorithms based on NNs on embedded devices like mobile phones or hearing aids with limited resources available. In this research work, we strive to close this gap by optimizing the Online Deep Attractor Network (ODANet) [2] for real-time speech separation on an embedded system through hyperparameter tuning and compression with a low-rank matrix factorization technique for NNs.

Application fields of the speech separation are for instance automatic meeting transcription or multi-party human-machine interactions for example with speech assistants [3]. In these application areas it is currently common practice that the audio signals are sent to a cloud computer and processed there. However, there is a growing interest in processing the audio signal on mobile devices without the need for an internet connection and sending the audio data to a cloud computer, which can strongly increase the latency [4]. Hearing aids are another conceivable field of application for speech separation algorithms where a short processing time is crucial and the available processing and memory resources are very limited.

Mathematically the speech separation problem is defined as separating C speech source signals $s_i(t)$ in a mixture

$$x(t) = \sum_{i=1}^{C} s_i(t) \tag{1.1}$$

as shown in the left part of figure 1.1, which illustrates a general speech separation system for two speakers. Many early NN-based approaches to the speech separation problem like Deep Clustering (DPCL) [5], Permutation Invariant Training (PIT) [3], or the Deep Attractor Network (DANet) [2, 6, 7] apply a short-time Fourier transform (STFT) on the time-domain mixture signal $x(t)$ and solve the speech separation problem in the time-frequency domain, where the result of the STFT is the complex mixture spectrogram

$$\mathcal{X}(f,t) = \sum_{i=1}^{C} \mathcal{S}_i(t,f) \tag{1.2}$$

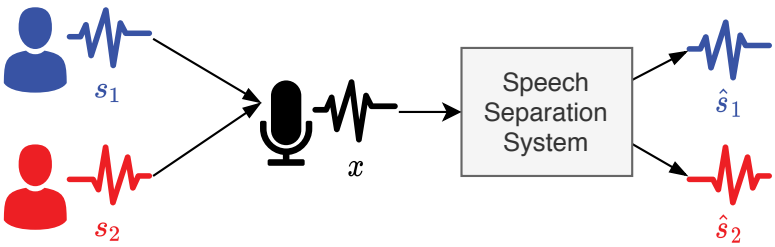


Figure 1.1.: Speaker 1 and speaker 2 speaking into the same microphone, which records the mixture x of the two overlapping signals (s_1 and s_2). The goal of the speech separation system is to estimate the original speech signals of speakers 1 and 2. (Icons from flaticon.com)

which is the sum of the complex speech source spectrograms $\mathcal{S}_i(t,f)$. DPCL, PIT, and the DANet solve the speech separation problem by generating masks $\mathbf{M}$ for the magnitude mixture spectrogram

$$X(f,t) = |\mathcal{X}(f,t)|. \tag{1.3}$$

For each speaker i the mask $\mathbf{m}_i$ indicates the affiliation of each time-frequency bin $X(f,t)$ in the mixture spectrogram to the speaker i [7]. The goal is to estimate the ideal binary mask (IBM), the ideal ratio mask (IRM), or the 'wiener-filter' like mask (WFM) which are defined as

$$IBM_{i,ft} = \delta(|\mathbf{S}_{i,ft}| > |\mathbf{S}_{j,ft}|) \quad \forall j \neq i \tag{1.4}$$

$$IRM_{i,ft} = \frac{|\mathbf{S}_{i,ft}|}{\sum_{j=1}^{C} |\mathbf{S}_{j,ft}|} \tag{1.5}$$

$$WFM_{i,ft} = \frac{|\mathbf{S}_{i,ft}|^2}{\sum_{j=1}^{C} |\mathbf{S}_{j,ft}|^2} \tag{1.6}$$

where $\mathbf{S}_i \in \mathbb{R}^{1\times FT}$ is the matrix containing the magnitude spectrogram of speaker i and $\delta(x) = 1$ if the expression x is true and $\delta(x) = 0$ if the expression x is false [7]. F and T are the number of frequency bins and the number of time frames of the STFT respectively. The source spectrograms $\mathcal{S}_i \in \mathbb{C}^{1\times FT}$ are estimated as

$$\widehat{\mathcal{S}}_i = \mathcal{X} \odot \widehat{\mathbf{m}}_i \tag{1.7}$$

where $\odot$ is the element-wise multiplication, $\widehat{\mathbf{m}}_i$ is the estimated mask for speaker i and $\mathcal{X} \in \mathbb{C}^{1\times FT}$ is the complex spectrogram.

Figure 1.1 illustrates the speech separation problem for two speakers and Figure 1.2 shows how a mask can be applied to the mixture in the time-frequency domain to separate two speakers.

The discussed time-frequency domain approaches [2,3,5–7] use deep neural networks (DNNs), which are NN with an input layer, an output layer and at least one hidden layer

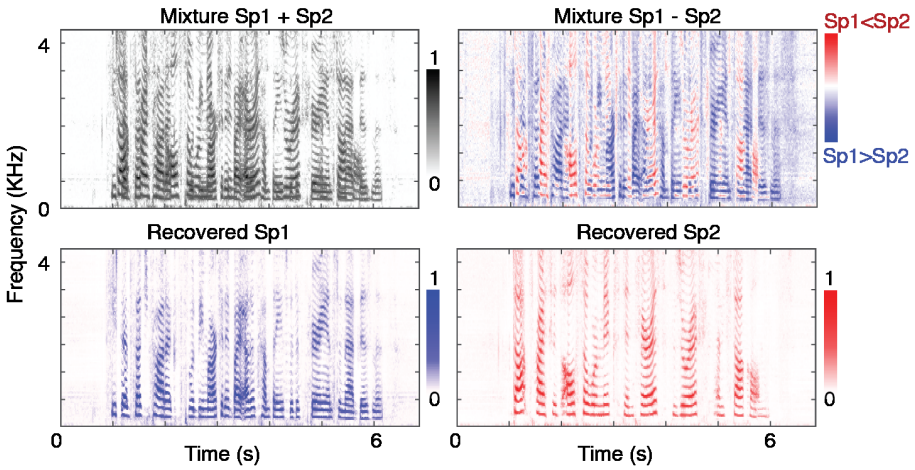


Figure 1.2.: Spectrograms of mixture (top left), mask (top right), and separated speaker 1 (bottom left) and speaker 2 (bottom right). Figure from [6].

in between. More particularly they use long short-term memory (LSTM) layers, which are recurrent neural network (RNN) layers. In RNNs the individual network cells have recurrent connections, which means each cells has a connection to itself with a time delay. Because of its recurrent connections and further mechanisms LSTMs are well suited for NN learning problems where the input has a meaningful time dimension [8], which is the case for speech signals.

For real-time processing it is required that the used system is causal, which means the system should only use information from current and past time steps and not look at future time steps when processing the current time step [9]. Furthermore, the latency between the audio input and the processed audio output should not be larger than 30 *ms* [10]. In this work, we aim for processing times below 8 *ms* per time frame, which is equal the hop size we use in the STFT. A NN with LSTMs can only be causal if unidirectional LSTMs are used in the NNs. However, most of the mentioned systems use bidirectional long short-term memory (BLSTM) and processes the audio data offline as a non-causal system. Even if unidirectional LSTMs are used, most of the networks [3, 5–7] process the time-frequency domain data not frame by frame but require meta-frames, which makes them non-causal systems.

That is why in this work we try to optimize the ODANet by Han et al. [2], a causal system, for real-time processing on an embedded system like the NVIDIA Jetson Nano. For this work we have chosen the NVIDIA Jetson Nano as the test system because this single-board computer has a similar form factor as the Raspberry Pi and is additionally equipped with a GPU [11]. To optimize the ODANet for embedded systems we try to reduce the number of weights especially in the LSTM layers. For this we investigate two methods, firstly hyper-parameter tuning and secondly compressing the LSTMs with the compression method proposed by Prabhavalkar et al. [4] based on low-rank matrix factorization.

Our findings suggest that the compression method by Prabhavalkar et al. is better suited for finding a good trade-off between low processing time and speech separation performance. By reducing the weights of the ODANet, among other things through the introduction of the Compressed Online Deep Attractor Network (CODANet), our work improves the real-time capability of ODANet on embedded systems. However, real-time speech separation on the selected embedded system, the NVIDIA Jetson Nano, is not achieved in this work.

The rest of this work is structured as follows. In chapter 2 we present related speech separation algorithms based on NNs, analyze different compression method for NNs, and look at related work that also aimed to optimize signal processing NNs for embedded systems. In the methods chapter 3 we explain the ODANet and its evolution from the DANet over the Anchored Deep Attractor Network (ADANet) in detail. Furthermore, we describe the used compression method by Prabhavalkar et al. [4] and the resulting compressed long short-term memorys (CLSTMs). Then we introduce the CODANet as well as an optimized way to initialize the anchor points in the ADANet, which is also used in the ODANet and CODANet. In chapter 4 we present the experimental setup used to train our NN implementations, the methods we used to evaluate the experiments, followed by the experimental results. Finally, we discuss the results in chapter 5 and conclude the work in chapter 6.

2. Related Work

In this chapter, we first briefly describe DNN-based single-channel speech separation algorithms related to the DANet. Second, we look at the different methods for the compression of NNs in general and explain our choice for the compression method of by Pravhavalkar et al. [4]. In the third part of this chapter, we present related work on the optimization of NN-based signal processing algorithms for embedded systems with numerical results.

2.1. Speech Separation Algorithms

In recent years there has been great progress in the research on single-channel speech separation with DNNs. These DNN-based algorithms can be categorized into algorithms operating in the time-frequency domain and algorithms operating directly in the time domain. Well known time-frequency domain approaches are (among others) DPCL by Hershey et al. [5] and PIT by Yu et al. [3].

The DPCL network maps each time-frequency bin into a higher dimensional embedding space $\mathbf{E} \in \mathbb{R}^{K \times FT}$ with the training objective to minimize the cost function

$$\mathcal{C}_{\mathbf{Y}}(\mathbf{E}) = \|\mathbf{E}^T\mathbf{E} - \mathbf{Y}^T\mathbf{Y}\|_F^2 \tag{2.1}$$

which calculates the distance in the Frobenius norm $\|\cdot\|_F$ between the true affinity matrix $\mathbf{Y}^T\mathbf{Y}$ and the estimated affinity matrix $\mathbf{E}^T\mathbf{E}$. [1] With this cost function the network learns to map time-frequency bins belonging to the same speaker into similar regions in the embedding space while separating time-frequency bins belonging to different speakers. During training, this is achieved using the true source assignments $\mathbf{Y}$, while during inference the network separates the sources by applying k-means in the embedding space, where the k-means clusters are equivalent to the estimated source assignments.

The DPCL method is further improved by Isik et al. [12] and Wang et al. [13] who present the DPCL++ and Chimera++ networks, respectively. Wang et al. [14] also present a low-latency DPCL version by replacing the bidirectional LSTMs with unidirectional LSTMs and changing the STFT window size from 32 *ms* to 8 *ms*. The cluster centers for k-means are calculated for the first 1.5 seconds of the signal during, which the network

[1] The notation from [5] is adapted to match the notation in this work. In [5] the embedding space is $\mathbf{V} \in \mathbb{R}^{FT \times K}$ in our work the embedding space is $\mathbf{E} \in \mathbb{R}^{K \times FT}$. In [5] the true source assignment is $\mathbf{Y} \in \mathbb{R}^{FT \times C}$ in our work the true source assignment is $\mathbf{Y} \in \mathbb{R}^{C \times FT}$.

is not able to generate any output. Afterwards, these cluster centers are used to generate the cluster assignments frame by frame for the rest of the utterance. In comparison the ODANet has a more dynamic way to calculate the cluster centers (attractors) online by updating them using a exponentially weighted moving average (see section 3.1.3).

PIT by Yu et al. [3] directly generates a mask from the magnitude spectrogram of the mixture input $\mathbf{X}$ using a DNN. This mask is then used to estimate the source signals $\hat{\mathbf{S}}_i$. Because there is a high probability of a permutation in the output (e.g. $\hat{\mathbf{S}}_1$ is a good estimate for $\mathbf{S}_2$ and vice versa), which would cause a high error in the training objective, which is to minimize the mean squared error (MSE) between $\hat{\mathbf{S}}_i$ and $\mathbf{S}_i$ for $i = 1, \ldots, C$, PIT minimizes the MSE for the permutation with the lowest MSE. Kolbaek et al. present an advanced version of PIT called utterance-level Permutation Invariant Training (uPIT) [15], which predicts phase sensitive masks instead of amplitude masks. Furthermore, the network architecture is improved by using LSTMs instead of non-recurrent DNNs or convolutional neural networks (CNNs), which are used in PIT and limit the network to a fixed input and output size. Because of this architecture change, uPIT is able to predict different input and output sizes and is also causal if unidirectional LSTMs are used.

One limitation of most DNN-based speech separation approaches in the time-frequency domain is that their performance is limited by the IBM, IRM, or WFM on the magnitude spectrogram, while the phase information for each source is taken from the mixture. In contrast, the Time-domain Audio Separation Network (TASNet) by Luo and Mesgarani [16] shows that time domain approaches can outperform time-frequency domain approaches because the former do not have this limitation. In addition to the performance advantages TASNet has also an advantage in the algorithmic latency because it does not require an STFT and works directly on the time domain signal on segments as short as 5 *ms* [16].

The TASNet uses an encoder-decoder framework and performs the speech separation by masking the encoder outputs, which are non-negative weights of the base signals contained in the audio segment to be separated [16]. The NN architecture of TASNet is further improved in [17–19]. In the NNs described in these papers, the base signals for the encoder and decoder are learned by the networks as trainable parameters. Ditter and Gerkmann [20] show that using a deterministic multi-phase gammatone filterbank instead of the learned base signals in the encoder-decoder framework further improves the performance. Other recent time-domain approaches to the speech separation problem are Wave-U-Net [21], FurcaNeXt [22], SepFormer [23], and Wavesplit [24].

Despite the advantages of the time-domain approaches, we focus on the ODANet, which operates in the time-frequency domain, because the ODANet is a causal systems with a relatively small model size (compare table 5.1 on page 52) and because in preliminary work we have successfully implemented a time-frequency domain speech separation algorithm for real-time processing. Nevertheless, the compression method by Prabhavalkar et al. [4] can be applied to all DNN-based speech separation algorithms

using RNNs in particular LSTMs, which is the case for the original TASNet in [16]. In the discussion, chapter 5, we compare our results to the related work mentioned in this section in terms of the performance and the model size.

2.2. Compression Methods for Neural Networks

In their survey paper Choudhary et al. [25] summarize several approaches for the compression and acceleration of machine learning (ML) models. The four main approaches to compress and accelerate a NN are

- pruning,
- quantization,
- knowledge destillation,
- low-rank factorization.

The main goal of pruning is to reduce the storage-size of the network. This is achieved by zeroing out network weights that are below a certain threshold or redundant. However, though pruning reduces the number of weights and thereby the model size, pruning does not reduce the number of operations by default. Only if neuron pruning or layer pruning is applied, which removes entire neurons or layers of a DNN, the number of operations is reduced [25].

Quantization of the network weights can lead to a significant size reduction. On the one hand, uniform quantization can also reduce computing time, e.g. quantizing 32-bit weights to 16-bit or 8-bit weights if the hardware efficiently processes 16-bit or 8-bit operations. On the other hand, while a non-uniform quantization can greatly reduce the model size, it cannot reduce the computation time [25].

In knowledge distillation a smaller 'student' network is trained to generate the outputs of a larger 'teacher' network when shown the same data. Knowledge distillation works good for classification problems [25].

In low-rank matrix factorization weight matrices $\mathbf{W} \in \mathbb{R}^{m \times n}$ are factorized in to matrices $\mathbf{P} \in \mathbb{R}^{m \times r}$ and $\mathbf{Z} \in \mathbb{R}^{r \times n}$ using a singular value decomposition (SVD) so that $\mathbf{PZ}$ is the best rank r approximation of $\mathbf{W}$. This reduces the size and the number of operations for a specific matrix from $m \cdot n$ to $r \cdot (m + n)$ [25]. This method is explained in detail in section 3.1.4.

We choose low-rank matrix factorization for compressing the ODANet because it is a deterministic method to reduce the size and the number of operations of a NN and because it has been previously applied to RNNs in particular LSTMs by Prabhavalkar et al. for automatic speech recognition [4].

2.3. Signal Processing with Neural Networks on Embedded Systems

While the previous section presents compression methods for NNs in general, this section presents related work on signal processing with neural networks on embedded systems with numerical results. This work is closely related to our work in the sense that it also considers the optimization of NN-based speech enhancement systems for embedded systems [26, 27] or compares the processing time of image processing NNs on the NVIDIA Jetson Nano to the procssing time on a desktop with a graphics processing unit (GPU) [28].

In speech enhancement the goal is to retrieve the clean speech signal $s(t)$ from the signal

$$x(t) = s(t) + n(t) \tag{2.2}$$

in which the clean speech signal is distorted with noise $n(t)$. Although speech enhancement and speech separation are related problems, they differ in that in speech separation several similar speech signals are to be separated from each other, whereas in speech enhancement exactly one speech signal is to be separated from the noise signal, which often has slightly different characteristics. Nevertheless, since this work is one of the first works to optimize a NN-based speech separation system for real-time processing on an embedded systems, we consult the results from [26, 27] in our discussion.

Drakopoulos et al. [26] propose the Auto-Encoder Convolutional Neural Network (AECNN) for real-time speech enhancement on an embedded system, in particular the Raspberry Pi 3 Model B+. They optimize the AECNN for the embedded system by hyper-parameter tuning the number of CNN layers and the number of filters in the CNNs. They report execution times from 42 ms per frame of size 1024 samples at a sampling rate of 16 kHz for a network with 3 million parameters to 5.7 ms per frame for a network with 0.2 million parameters and a frame size of 128 frames at a sampling rate of 16 kHz. Drakopoulos et al. present the speech enhancement performance in terms of the perceptual evaluation of speech quality score (PESQ) [29] and the segmental sources-to-noise ratio (SNR) [26] but do not compare their results to a baseline or any other research. Among other results Drakopoulos et al. report that their network is 6.5% faster with a Tensorflow frontend than with a Keras frontend [26].

In contrast to Drakopoulos et al., who use hyper-parameter tuning for the embedded optimization, Fedorov et al. [27] propose TinyLSTM, which is optimized for embedded speech enhancement on hearing aids by applying pruning and integer quantization. Their baseline architecture consists of 2 unidirectional LSTM layers with 256 units in each layer and two fully connected layers with 128 units in each layer. With a total of 0.97 million parameters their baseline is about 12-times smaller than our ODANet baseline, which has almost 12 million parameters (see equation 3.44 on page 29 for the calculation of the number of parameters of the ODANet). With pruning and quantization from 32bit

floating point weights to 8 bit integer weights Fedorov et al. are able to reduce the model size from 3.7 megabyte (MB) to 0.31 MB, while the number of parameters is reduced to 34% of the baseline from 0.97 million to 0.33 million. This size reduction results in a time reduction from 12.52 *ms* to 4.26 *ms* per frame on a micro-controller unit hardware. Fedorov et al. are able to achieve this reduction in the model size with only a small reduction of 4% in the source-to-distortion ratio (SDR) speech enhancement performance on CHiME2 WSJ0 dataset [30].

Last but not least, we would like to mention related work on the NVIDIA Jetson hardware. Bianco et al. [28] compare the performance and run-time of image recognition DNNs on a desktop computer with a NVIDIA Titian X Pascal GPU to the performance and run-time of the same DNNs on the NVIDIA Jetson TX1 board, which has 256 GPU cores compared to 128 GPU cores on the NVIDIA Jetson Nano, but other than that comparable hardware specifications [11,31] (all relevant NVIDIA Jetson Nano hardware specifications are presented in section 4.1 on page 32). Bianco et al. report that the NVIDIA Jetson TX1 takes 5 to 35 times longer than the personal computer (PC) with the NVIDIA Titian X Pascal GPU to process a single image. We use these values as an orientation for our evaluation of the ODANet and the CODANet, which are presented in the next chapter.

3. Methods

This chapter introduces the methods we used for our implementation (section 3.1) as well as our proposal to combine and extend these methods to the Compressed Online Deep Attractor Network (CODANet) (section 3.2).

3.1. Utilized Methods

In this section, we describe the Deep Attractor Network (DANet) and its evolution from the first paper [6] over the Anchored Deep Attractor Network (ADANet) [7] to the Online Deep Attractor Network (ODANet) [2]. For each network we explain the general idea, the architecture, and implementation details of our implementation that differ from the original description.

Furthermore, we present the compression method by Prabhavalkar et al. [4] for RNNs that jointly compresses the recurrent and non-recurrent weight matrices of RNN layers with a low-rank matrix factorization and leads to the CLSTM.

3.1.1. Deep Attractor Network (DANet)

To solve the speech separation problem, the DANet by Chen et al. [6] learns to map time-frequency bins belonging to the same speaker close to the attractor point of that speaker in the embedding space. Similar to DPCL [5] the idea of the embedding space is that points belonging to the same speaker are mapped close to each other, while points belonging to different speakers are supposed to have a large distance[1] in the embedding space. In the DANet this mapping is not the explicit goal but rather are the mapping into the embedding space and the positions of the attractor points implicitly learned by the overall training target.

The idea behind the attractor points is that each attractor point attracts the time-frequency bins of a specific speaker assigned to it and at the same time repels time-frequency bins of other speakers. The idea of the attractors is based on the perceptual magnet effect observed by Kuhl [32]. Kuhl shows that humans can identify prototypes of speech categories if they are played the prototype sound and variations of it. In one of his experiments for example adults were plaid different variations of the $/i/$ vowel and asked to rate the goodness of it. This experiment showed that across all listeners the rating was similar and the prototype $/i/$ vowel was always rated best. In further experiments Kuhl

[1] In DANet the dot product is used as the distance measure.

shows that these prototypes of sound categories serve as reference mental representations (reference points) when listening to speech sounds. These reference points are called perceptual magnets. The idea of the DANet can be seen as learning perceptual magnets for speaker characteristics. In the DANet the perceptual magnets are called attractor points.

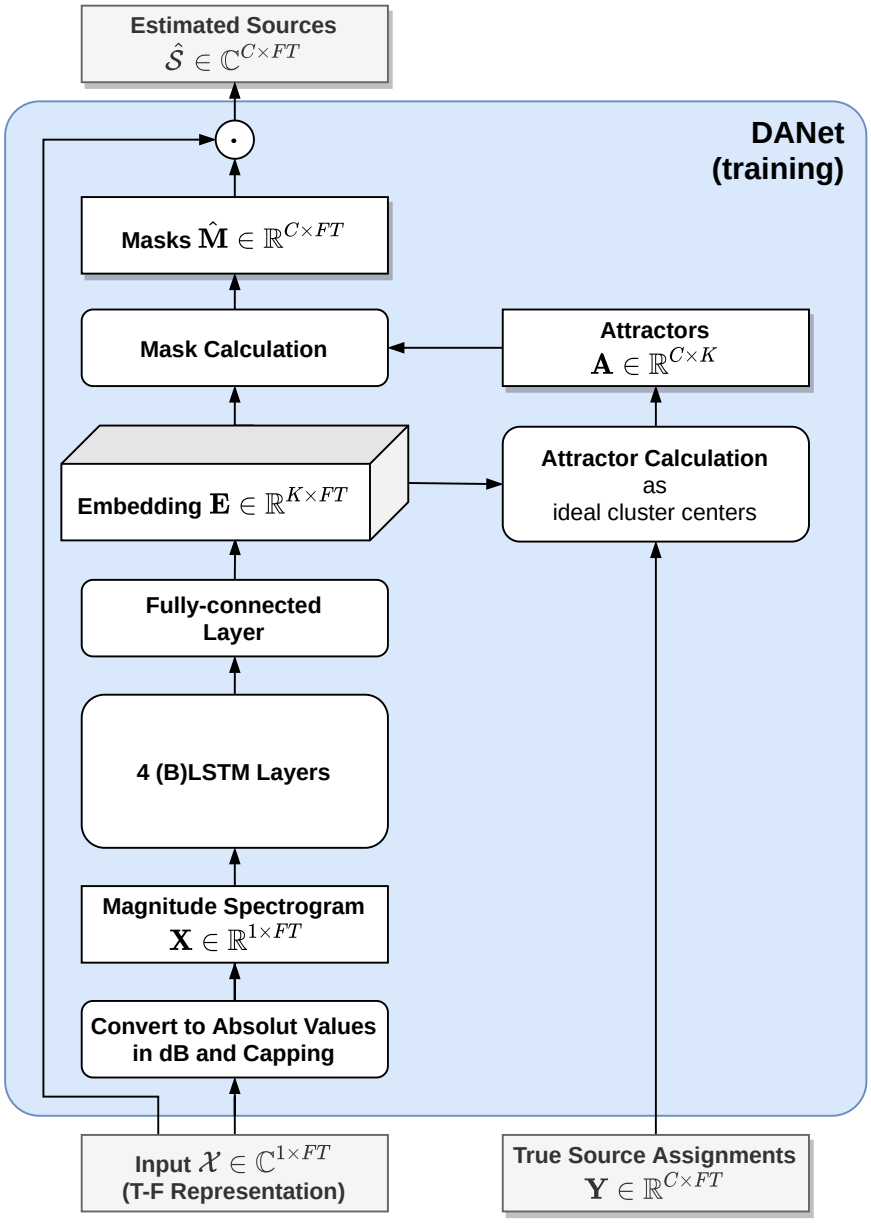


Figure 3.1.: DANet architecture (training). Graphic adapted from [7, p. 789].

In the following we describe the DANet architecture that tries to implement this idea. Figure 3.1 shows our DANet architecture during training based on [6]. The input to the network is the complex spectrogram $\mathcal{X} \in \mathbb{C}^{1 \times FT}$. This input is converted into the real valued magnitude spectrogram $\mathbf{X} \in \mathbb{R}^{1 \times FT}$ in decibel (dB). Furthermore, the network cappes the magnitude spectrogram in dB at -80 dB and 200 dB to avoid large negative and positive values. The magnitude spectrogram in dB is then mapped into the K-dimensional

embedding space $\mathbf{E} \in \mathbb{R}^{K \times FT}$ by four LSTM layers and one fully-connected layer.

During training the attractor points $\mathbf{A} \in \mathbb{R}^{C \times K}$ are calculated as the ideal cluster centers of the time-frequency bins belonging to the same speaker

$$\mathbf{a}_i = \frac{\mathbf{y}_i \mathbf{E}^T}{\sum_{f,t} \mathbf{y}_i} \quad i = 1, 2, \ldots, C \tag{3.1}$$

using the true source assignment $\mathbf{Y} \in \mathbb{R}^{C \times FT}$. The true source assignment can be the IBM, IRM, or the WFM.

Using the attractor points the DANet estimates the mask $\widehat{\mathbf{m}}_i \in \mathbb{R}^{1 \times FT}$ for speaker i by calculating the distance between the time-frequency bins and the attractor points in the embedding space

$$\mathbf{d}_i = \mathbf{a}_i \mathbf{E} \quad i = 1, 2, \ldots, C \tag{3.2}$$

and assigning each time-frequency bin to the source of the attractor point that it is closest to in the embedding space

$$\widehat{\mathbf{m}}_i = \mathcal{H}(\mathbf{d}_i) \quad i = 1, 2, \ldots, C \tag{3.3}$$

where $\mathcal{H}(\mathbf{d}_i)$ is the softmax or sigmoid function

$$\mathcal{H}(\mathbf{d}_i) = \begin{cases} \text{softmax}(\mathbf{d}_{i,ft}) = \frac{\exp(d_{i,ft})}{\sum_{j=1}^{C} \exp(d_{j,ft})} \\ \text{sigmoid}(\mathbf{d}_{i,ft}) = \frac{1}{\sum_{j=1}^{C} (1 + \exp(-d_{j,ft}))} \end{cases} . \tag{3.4}$$

Using the estimated masks $\widehat{\mathbf{M}} \in \mathbb{R}^{C \times FT}$ the estimated complex source spectrograms $\widehat{\mathcal{S}}_i \in \mathbb{C}^{1 \times TF}$ for each speaker i are calculated as

$$\widehat{\mathcal{S}}_i = \mathcal{X} \odot \widehat{\mathbf{m}}_i \quad i = 1, 2, \ldots, C \tag{3.5}$$

where $\odot$ is the element-wise multiplication.

The training objective of the DANet is to minimize the MSE loss function

$$\mathcal{L} = \frac{1}{C \cdot F \cdot T} \sum_{i=1}^{C} \| \mathcal{S}_i - \widehat{\mathcal{S}}_i \|_2^2 \tag{3.6}$$

between the estimated source signals $\widehat{\mathcal{S}}_i$ and the true source signals $\widehat{\mathcal{S}}_i$.

Because the true source assignments $\mathbf{Y}$ are unknown during the inference of the DANet, the network architecture changes from training to inference. Figure 3.2 shows the DANet inference architecture. The network architecture from the spectrogram input $\mathcal{X}$ to the embedding $\mathbf{E}$, which contains all trainable parameters, is the same for training and inference. What changes is the way the attractor points are calculated. During the inference the k-means clustering algorithm is used to identify clusters of time-frequency bins in the embedding space and the resulting cluster centers are chosen to be the attractor points.

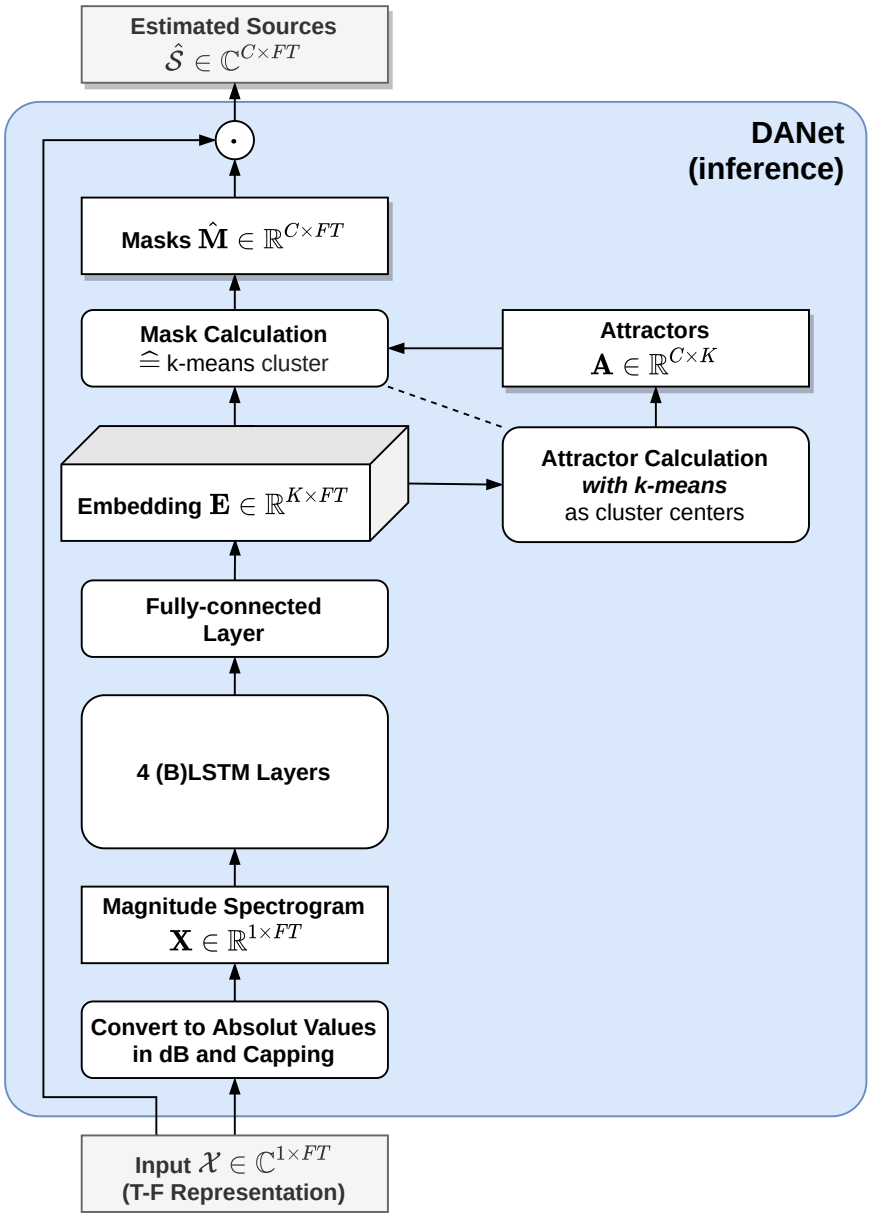


Figure 3.2.: DANet architecture (inference). Graphic adapted from [7, p. 789].

Using these attractor points the estimated binary mask is equivalent to the cluster assignments in k-means.

Our implementation differs from the DANet implementation from Chen et al. [6] in two ways. First, Chen et al. do not describe capping the magnitude values at -80 dB and 200 dB. Second, we use the complex spectrograms in the MSE loss function $\mathcal{L}$. In contrast, Chen et al. do not explicitly mention if they use the complex or real-valued magnitude spectrogram in the loss function. Furthermore, Chen et al. use the squared error and not the mean squared error for their loss function. We use the mean squared error to be able to better interpret the value of the loss function.

The main disadvantages of the DANet are that the architecture differs between training and inference and that it uses k-means during inference, which is computationally expensive. To overcome these disadvantages Luo et al. propose the Anchored Deep Attractor Network, which is presented in the next section.

3.1.2. Anchored Deep Attractor Network (ADANet)

Evaluating the experiments of the previously described DANet the authors, Chen, Luo, and Mesgarani, observed that the attractor points always lie in similar regions as shown in figure 3.3. Based on this observation, they proposed two further methods to find the attractor points in addition to k-means in [7].

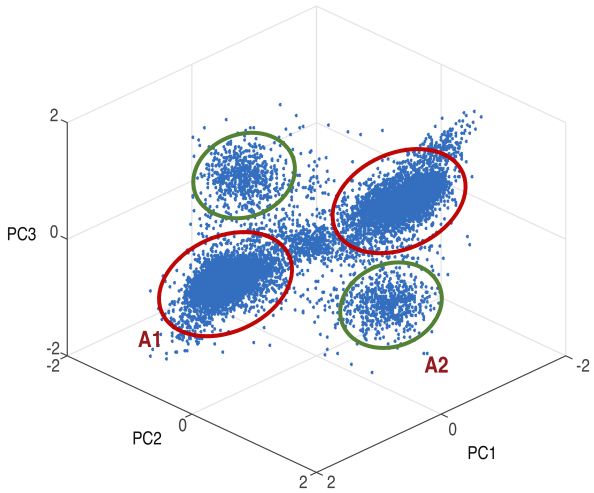


Figure 3.3.: PCA of attractor points from 10.000 utterances. A1 and A2 highlight regions of attractor point pairs. Edited graphic from [6, p. 249].

1. **Fixed attractor points:** The first idea mentioned in [6] and [7] is to use a pair of fixed attractor points (one attractor point for each speaker). E.g. the cluster centers of regions A1 in figure 3.3 could be used as the fixed pair of attractor points.

2. **Anchor points:** The second idea is to use anchor points to get an estimated source assignment $\widehat{\mathbf{Y}}$, which is then used to calculate the attractor points using

equation 3.1. The position of the anchor points are also learned during the training of the ADANet. This method can also be seen as reducing the k-means algorithm to one single expectation maximisation (EM) step with an intelligent initialization of the cluster centers for the expectation step with the anchor points.

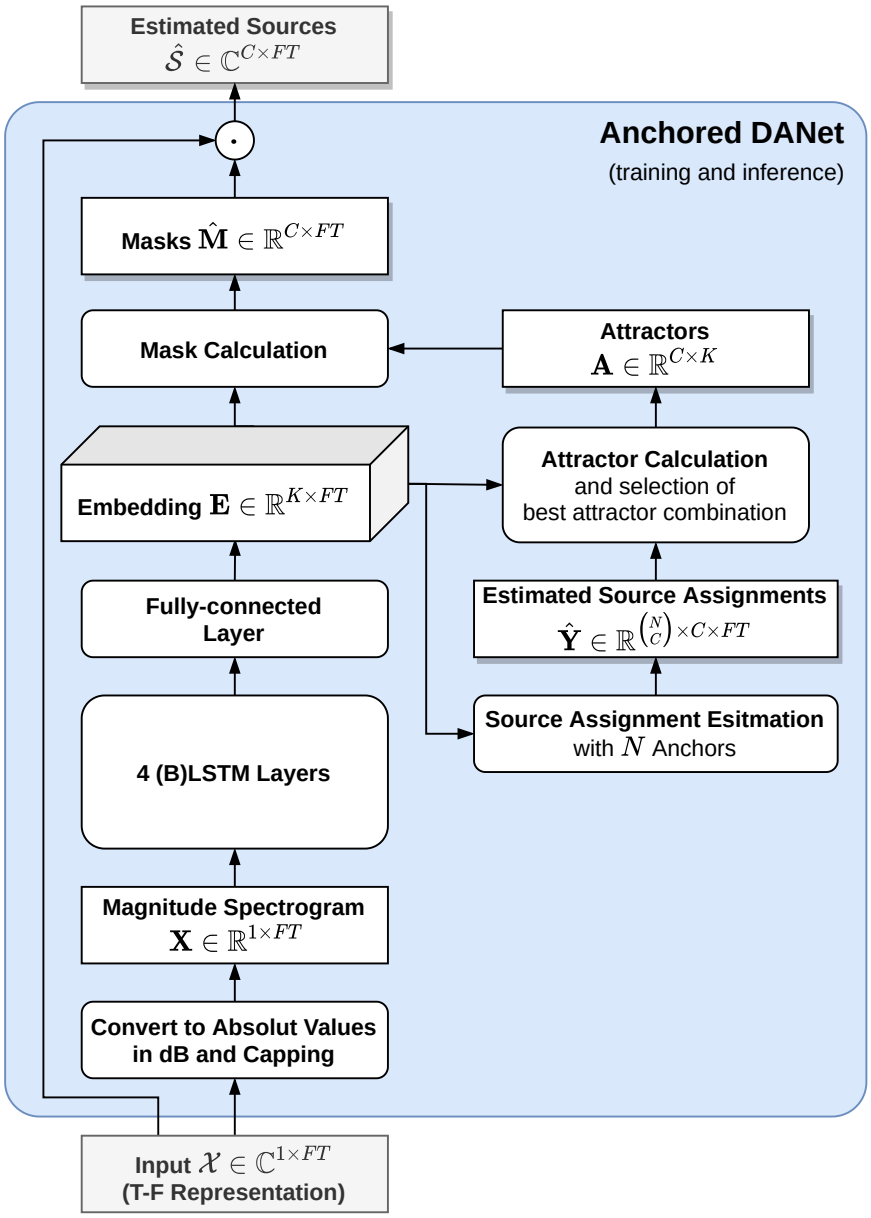


Figure 3.4.: ADANet architecture. Graphic adapted from [7, p. 789].

The second idea, which leads to the ADANet is explained in detail in this section. Figure 3.4 show the architecture of the ADANet. The left part is equivalent to the DANet. What changes is the way the attractor points are calculated illustrated on the right side of the figure. The ADANet uses N anchor points $\boldsymbol{\psi}_j \in \mathbb{R}^{1 \times K}$ to estimate the source assignment where N is chosen to be greater than or equal to the number of speaker C.

Each anchor point functions as a reference point for a probable attractor point position.

The anchor points are initialized randomly and the probable anchor point positions are learned during the training of the network, which strongly correlate with the attractor point positions.

For all $\binom{N}{C}$ combinations of the anchor points the resulting estimated source assignment is calculated in the following way. Let $\mathbf{\Psi}_p \in \mathbb{R}^{C\times K}$ be the p-th combination of C anchor points with $p = 1, 2, \dots \binom{N}{C}$. (E.g. $\mathbf{\Psi}_1 = [\boldsymbol{\psi}_1\ \boldsymbol{\psi}_2]^T$) For each combination p the distance between the time-frequency points and the anchor points of this combination is calculated similar to equation 3.2 as

$$\mathbf{D}_p = \mathbf{\Psi}_p \mathbf{E}, \quad p = 1, 2, \dots \binom{N}{C}. \tag{3.7}$$

Afterwards, the estimated source assignment for the combination p is calculated as

$$\hat{\mathbf{Y}}_p = softmax(\mathbf{D}_p). \tag{3.8}$$

In the following, equation 3.1 is used to calculate the attractor points $\mathbf{A}_p \in \mathbb{R}^{C\times K}$ for each possible source assignment p in the $\binom{N}{C}$ combinations resulting in the same number of possible attractor point combinations. Now the best attractor point combination is selected, which is the combination of attractor points where the distance between the two closest attractor points in the combination is maximal. This combination is found using the following equations.

A similarity matrix $\mathbf{\Gamma}_p \in \mathbb{R}^{C\times C}$ that contains the pairwise similarity between the attractor points in the combination p is calculated as

$$\mathbf{\Gamma}_p = \mathbf{A}_p \mathbf{A}_p^T \tag{3.9}$$

where $\mathbf{A}_p$ contains the row vectors of the attractor points in the combination with the index p. For example for 3 speakers ($C = 3$) $\mathbf{A}_1$ would have the following structure

$$\mathbf{A}_1 = \begin{bmatrix} \cdots & \mathbf{a}_1 & \cdots \\ \cdots & \mathbf{a}_2 & \cdots \\ \cdots & \mathbf{a}_3 & \cdots \end{bmatrix} \quad \in \mathbb{R}^{3\times K}.$$

This results in the following similarity matrix

$$\mathbf{\Gamma}_{p=1} = \begin{bmatrix} \langle \mathbf{a}_1, \mathbf{a}_1 \rangle & \langle \mathbf{a}_1, \mathbf{a}_2 \rangle & \langle \mathbf{a}_1, \mathbf{a}_3 \rangle \\ \langle \mathbf{a}_2, \mathbf{a}_1 \rangle & \langle \mathbf{a}_2, \mathbf{a}_2 \rangle & \langle \mathbf{a}_2, \mathbf{a}_3 \rangle \\ \langle \mathbf{a}_3, \mathbf{a}_1 \rangle & \langle \mathbf{a}_3, \mathbf{a}_2 \rangle & \langle \mathbf{a}_3, \mathbf{a}_3 \rangle \end{bmatrix}$$

where $\langle \mathbf{a}_i, \mathbf{a}_j \rangle$ denotes the dot product between attractor $\mathbf{a}_i$ and $\mathbf{a}_j$. This dot product measures the similarity between the two attractor points and is maximal if $\mathbf{a}_i$ and $\mathbf{a}_j$ are identical. The absolute value of the dot product is minimal if the two vectors $\mathbf{a}_i$ and $\mathbf{a}_j$ are orthogonal. The dot product reaches its minimum (a negative value) if the two vectors $\mathbf{a}_i$

and $\mathbf{a}_j$ are pointing in opposite directions from the origin.

The maximum value of the non-diagonal values of $\mathbf{\Gamma}_p$ is selected as

$$\gamma_p = \max\{\mathbf{\Gamma}_{p_{i,j}}\}, \quad i \neq j \tag{3.10}$$

which represents the attractor point pair in the combination p with the largest similarity.

Finally, the combination of attractor points $\mathbf{A}_p$, for which the largest in-set similarity γ_p between the two most similar attractor points is minimal, is selected as

$$\mathbf{A} = \underset{\mathbf{A}_p}{\operatorname{argmin}}\{\gamma_p\}, \quad p = 1, 2, \ldots, \binom{N}{C} \tag{3.11}$$

which is the combination of attractor points, for which the distance between the two closest attractor points in the combination is maximal. Now this best attractor point combination $\mathbf{A}$ is used to calculate the estimated masks and estimated signal sources, which is done analogously to the DANet.

The ADANet eliminates the disadvantages of the DANet by introducing anchor points, which allow the architecture to be the same for training and inference and reduce k-means clustering to one EM step, which is directly integrated in the network.

One difference between our ADANet implementation and the implementation by Lue et al., is that Luo et al. use a salient weight threshold keeping 90% of the salient time-frequency bins for the attractor calculation while the other 10% are not taken into account for the attractor calculation (equation 3.1). We do not investigate this difference because it only improves the scale-invariant source-to-noise ratio (SI-SNR) by less than 5% [7] and it is not clear how to implement this threshold in real-time. For example with a fixed salient weight threshold the results depend on the level of the audio signal. A relative threshold is more complicate to implement in real-time because it has to be updated using e.g. a moving average.

In the following section, we describe the ODANet a causal version of the ADANet.

3.1.3. Online Deep Attractor Network (ODANet)

The ODANet by Han et al. [2] has the same network architecture as the ADANet in terms of the LSTM layers and the dense layer but processes the spectrogram frame by frame. Because the network architecture including the trainable weights does not change, Han et al. suggest to use the weights of a previously trained ADANet in the ODANet. What changes is the way the attractor points are calculated.

For the first frame ($t = 1$) in the ODANet the attractor points are calculated using the anchor points just like in the ADANet. Therefore, the ODANet architecture for the first frame is similar to the ADANet architecture shown in figure 3.4. This figure is adapted specifically for the first frame of the ODANet in the appendix A.2 by changing the dimensions from FT to F indicating that the ODANet processes single frames.

For every following frame the previous attractor points $\mathbf{A}_{t-1}$ are used instead of the anchor points to estimate the source assignments

$$\hat{\mathbf{Y}}_t = \text{softmax}(\mathbf{A}_{t-1}\mathbf{E}_t). \tag{3.12}$$

Based on this source assignment estimation $\hat{\mathbf{Y}}_t$ the current attractor point position for speaker i is estimated equivalent to equation 3.1 as

$$\hat{\mathbf{a}}_{t,i} = \frac{\hat{\mathbf{y}}_{t,i}\mathbf{E}_t^T}{\sum_f \hat{\mathbf{y}}_{t,i}} \quad i = 1, 2, \ldots, C. \tag{3.13}$$

To avoid that the attractor points jump around between time frames (e.g. during silence frames), Han et al. propose to use an exponentially weighted moving average for updating the position of the attractor points as follows

$$\mathbf{a}_{t,i} := \alpha_{t,i}\hat{\mathbf{a}}_{t,i} + (1 - \alpha_{t,i})\mathbf{a}_{t-1,i}. \tag{3.14}$$

Furthermore, they propose two ways to calculate the update coefficient $\alpha_{t,i}$

1. context-based weighting,
2. dynamic weighting.

In context-based weighting the update coefficient $\alpha_{t,i}$ is calculated as

$$\alpha_{t,i} = \frac{\sum_f \hat{\mathbf{y}}_{t,i}}{\sum_{j=t-\tau}^{t} \sum_f \hat{\mathbf{y}}_{j,i}} \tag{3.15}$$

where τ is the number of time frames taken into the context-window. This means that if there is a large number of time-frequency bins belonging to speaker i in the current time frame t compared to the number of time-frequency bins belonging to speaker i in the previous τ time frames, then the update coefficient $\alpha_{t,i}$ for speaker i at time frame t is high. With a high update coefficient the current estimate $\hat{\mathbf{a}}_{t,i}$ has a large impact on the value $\mathbf{a}_{t,i}$ while the old value $\mathbf{a}_{t-1,i}$ is discounted stronger.

In the dynamic weighting the update coefficients are learned by the network as

$$\boldsymbol{\alpha}_{t,i} = \frac{\mathbf{g}_{t,i} \cdot \sum_f \hat{\mathbf{y}}_{t,i}}{\mathbf{f}_{t,i} \cdot \sum_{j=t-\tau}^{t-1} \sum_f \hat{\mathbf{y}}_{j,i} + \mathbf{g}_{t,i} \cdot \sum_f \hat{\mathbf{y}}_{t,i}} \tag{3.16}$$

where gate $\mathbf{g}_{t,i}$ and forget-gate $\mathbf{f}_{t,i}$ are calculated as

$$\mathbf{g}_{t,i} = \sigma(\mathbf{h}_{t-1}^{l=4}\mathbf{W}_\text{g} + \mathbf{X}_t\mathbf{U}_\text{g} + \mathbf{a}_{t-1}\mathbf{J}_\text{g} + \mathbf{b}_\text{g}) \tag{3.17}$$

$$\mathbf{f}_{t,i} = \sigma(\mathbf{h}_{t-1}^{l=4}\mathbf{W}_\text{f} + \mathbf{X}_t\mathbf{U}_\text{f} + \mathbf{a}_{t-1}\mathbf{J}_\text{f} + \mathbf{b}_\text{f}) \tag{3.18}$$

where $\mathbf{h}_{t-1}^{l=4} \in \mathbb{R}^{1\times N_4}$ is the output of the fourth LSTM layer with N_4 units at time $t-1$, $\mathbf{X}_t$ is the magnitude spectrogram at time t, and $\mathbf{a}_{t-1}$ is the attractor point for speaker i at time $t-1$. The other parameters $\mathbf{W}_g, \mathbf{W}_f \in \mathbb{R}^{N_4\times K}$, $\mathbf{U}_g, \mathbf{U}_f \in \mathbb{R}^{F\times K}$, $\mathbf{J}_g, \mathbf{J}_f \in \mathbb{R}^{K\times K}$, and $\mathbf{b}_g, \mathbf{b}_f \in \mathbb{R}^{1\times K}$ are trainable weights.

For this work, we do not implement the dynamic weighting because it only slightly improves the SDR by 5% according to [2, p. 363, table 1] and it is computationally more complex than the context based weighting mechanism.

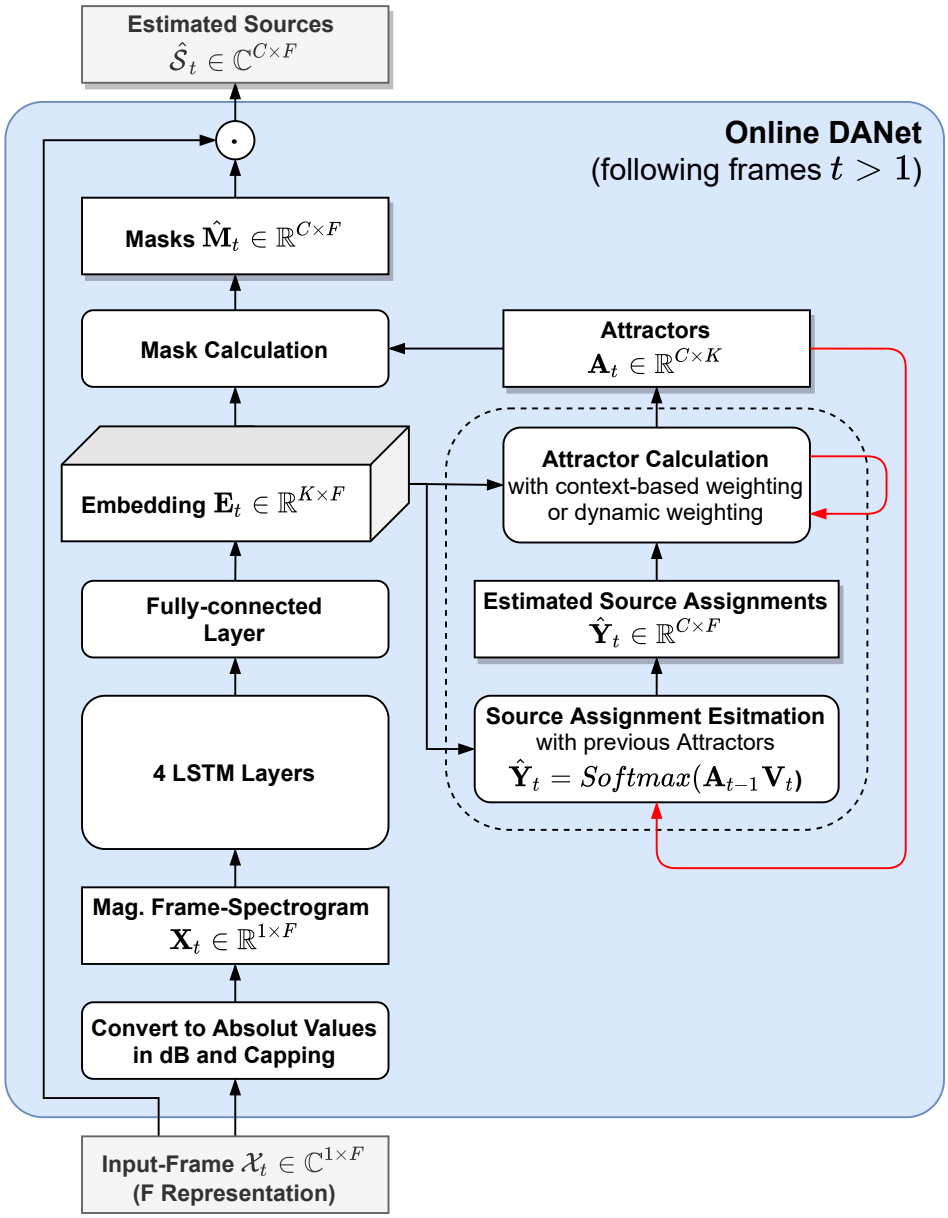


Figure 3.5.: ODANet architecture (following frames $t > 1$). Red lines represent recurrent connections with time delay. The dashed box with rounded edges represents a super-process that combines the entire attractor calculation for the following frames and the attractor calculation for the for the first-frame, which is shown in the appendix in figure A.2 on page 63. Graphic adapted from [7, p. 789].

Figure 3.5 shows the ODANet architecture for the frames following the first frame with $t > 1$. The red line from the attractors to the source assignment estimation represents the recurrent connection required for the source assignment estimation with the attractors from the previous time step (equation 3.12). Another recurrent connection is required for calculating the update coefficient in the context-based weighting (equation 3.15). Here the recurrent connection contains the history of

$$\sum_f \hat{\mathbf{y}}_{t,y} \tag{3.19}$$

for t in $t - \tau$ to $t - 1$.

The entire attractor calculation for the first frame and the following frames of the ODANet is implemented in a super-process represented by the dashed box with rounded edges in figure 3.5 and figure A.2 in the appendix on page 63. This super-process has a `first_frame` flag, which is set to zero for the first frame and one for every following frame. Based on this flag the super-process calculates the attractor points using the anchor points (for the first frame) or the previous attractor points for every following frame.

All weights in the ODANet are initialized with weights from a previously trained ADANet also known as curriculum training [2]. Due to time constrains we use the weights from the ADANet directly in our ODANet experiments without further training the ODANet.

3.1.4. Compressed Long Short-term Memory (CLSTM)

In this section, we first present the compression method for RNNs by Prabhavalker et al. [4] in general. Then we detail how this method can be applied to LSTMs and thus leads to the CLSTMs.

The compression method by Prabhavalker et al. is a generalization of the method from Xue et al. [33]. Both methods are low-rank factorization compression methods. In contrast to the method by Xue et al. [33], which only compresses the kernel using a SVD, the method by Prabhavalker et al. [4] jointly optimizes the recurrent kernel and kernel of the following layer. This is achieved by a low-rank factorization of the recurrent kernel $\mathbf{W}_h^l \in \mathbb{R}^{N_l \times N_l}$, splitting it into a projection $\mathbf{P}^l \in \mathbb{R}^{N_l \times r_l}$ and a back-projection matrix $\mathbf{Z}_h^l \in \mathbb{R}^{r_l \times N_l}$, followed by an adaption of the kernel $\mathbf{W}_x^{l+1} \in \mathbb{R}^{N_l \times N_{l+1}}$ of the next layer to match the projection matrix $\mathbf{P}^l$. [2]

This idea is illustrated in figure 3.6. Figure 3.6a shows layer l of a general RNN and figure 3.6b shows the compressed version. In the compressed version the output $\mathbf{h}_t^l \in \mathbb{R}^{1 \times N_l}$ is multiplied with the projection matrix $\mathbf{P}^l$ resulting in the projected output $\widetilde{\mathbf{h}}_t^l \in \mathbb{R}^{1 \times r_l}$, which is used in the recurrent part of layer l and passed to the next layer $l + 1$. In the following, we explain the calculation of the matrices for the compressed RNN.

[2] The superscripts l and $l + 1$ on the weight matrices are layer indices and not exponentials.

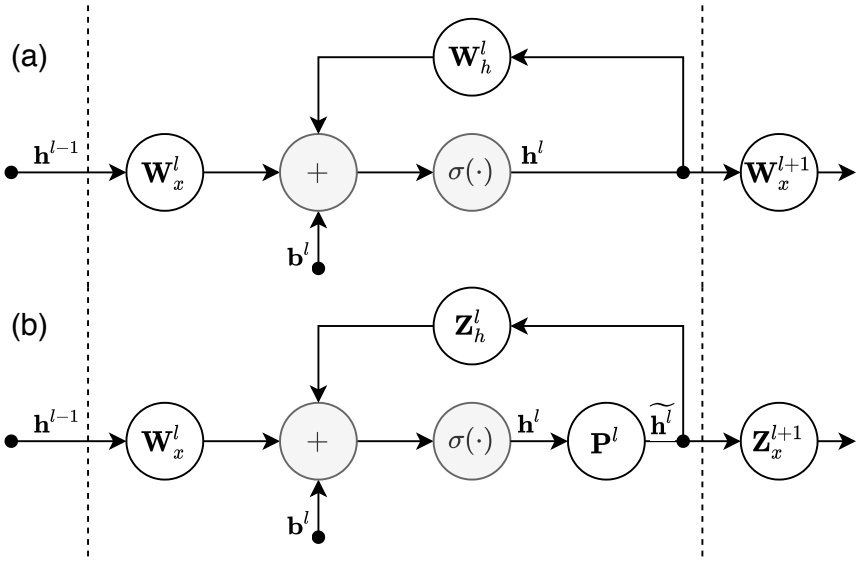


Figure 3.6.: The general RNN (a) is compressed in (b) by jointly factorizing recurrent kernel $\mathbf{W}_h^l$ and kernel $\mathbf{W}_x^{l+1}$ to $\mathbf{Z}_h^l$ and $\mathbf{Z}_x^{l+1}$ using the same projection matrix $\mathbf{P}^l$. The dashed lines represent borders between layers. The graphic is adapted from [4, p. 5971] (see footnote 3).

For the general RNN the outputs $\mathbf{h}_t^l \in \mathbb{R}^{1\times N_l}$ and $\mathbf{h}_t^{l+1} \in \mathbb{R}^{1\times N_{l+1}}$ are calculated as

$$\mathbf{h}_t^l = \sigma(\mathbf{h}_t^{l-1}\mathbf{W}_x^l + \mathbf{h}_{t-1}^l\mathbf{W}_h^l + \mathbf{b}^l) \tag{3.20}$$

$$\mathbf{h}_t^{l+1} = \sigma(\mathbf{h}_t^l\mathbf{W}_x^{l+1} + \mathbf{h}_{t-1}^{l+1}\mathbf{W}_h^{l+1} + \mathbf{b}^{l+1}) \tag{3.21}$$

for layer l with N_l units and layer $l+1$ with N_{l+1} units.[3] Where $\mathbf{b}^l \in \mathbb{R}^{1\times N_l}$ and $\mathbf{b}^{l+1} \in \mathbb{R}^{1\times N_{l+1}}$ are the biases of layer l and $l+1$.

In the compressed version the outputs $\mathbf{h}_t^l \in \mathbb{R}^{1\times N_l}$ and $\mathbf{h}_t^{l+1} \in \mathbb{R}^{1\times N_{l+1}}$ are calculated as

$$\mathbf{h}_t^l = \sigma(\mathbf{h}_t^{l-1}\mathbf{W}_x^l + \mathbf{h}_{t-1}^l\mathbf{P}^l\mathbf{Z}_h^l + \mathbf{b}^l) \tag{3.22}$$

$$\mathbf{h}_t^{l+1} = \sigma(\mathbf{h}_t^l\mathbf{P}^l\mathbf{Z}_x^{l+1} + \mathbf{h}_{t-1}^{l+1}\mathbf{W}_h^{l+1} + \mathbf{b}^{l+1}) \tag{3.23}$$

by replacing the recurrent kernel $\mathbf{W}_h^l \in \mathbb{R}^{N_l\times N_l}$ (from equation 3.20) with the projection matrix $\mathbf{P}^l \in \mathbb{R}^{N_l\times r_l}$ and recurrent kernel back-projection matrix $\mathbf{Z}_h^l \in \mathbb{R}^{r_l\times N_l}$. These matrices are calculated using a SVD

$$\text{SVD:}\quad \mathbf{W}_h^l = \mathbf{U}_h^l\mathbf{\Sigma}_h^l{\mathbf{V}_h^l}^T \tag{3.24}$$

$$\approx (\widetilde{\mathbf{U}_h^l}\widetilde{\mathbf{\Sigma}_h^l})\widetilde{\mathbf{V}_h^l}^T = \mathbf{P}^l\mathbf{Z}_h^l \tag{3.25}$$

[3] We change the notation from [4] in the following way. For all matrices with subscript x the layer index l is increased by one. The matrix vector multiplications are transposed because this is closer to the Keras implementation of a SimpleRNNCell [34].

resulting in $\mathbf{U}_h^l \in \mathbb{R}^{N_l \times N_l}$, the diagonal matrix $\mathbf{\Sigma}_h^l \in \mathbb{R}^{N_l \times N_l}$ containing the singular values, and $\mathbf{V}_h^l \in \mathbb{R}^{N_l \times N_l}$. Now the matrices ($\mathbf{U}_h^l$, $\mathbf{\Sigma}_h^l$, and $\mathbf{V}_h^l$) are truncated by taking only the first r_l columns of $\mathbf{U}_h^l$ and the first r_l rows of ${\mathbf{V}_h^l}^T$ corresponding to the r_l largest singular values. Using the truncated matrices $\widetilde{\mathbf{U}_h^l} \in \mathbb{R}^{N_l \times r_l}$, $\widetilde{\mathbf{\Sigma}_h^l} \in \mathbb{R}^{r_l \times r_l}$, and $\widetilde{\mathbf{V}_h^l}^T \in \mathbb{R}^{r_l \times N_l}$ the matrices $\mathbf{P}^l$ and $\mathbf{Z}_h^l$ are calculated as shown in equation 3.25. According to the proven Eckart–Young–Mirsky theorem, $\mathbf{P}^l\mathbf{Z}_h^l$ is the best rank r_l approximation of the matrix $\mathbf{W}_h^l$ [35, 36].

In other words the output $\mathbf{h}_t^l \in \mathbb{R}^{1 \times N_l}$ is projected from the N_l-dimensional space onto a smaller r_l-dimensional space. This projected output $\widetilde{\mathbf{h}_t^l} = \mathbf{h}_t^l\mathbf{P}^l \in \mathbb{R}^{1 \times r_l}$ is not only used for the recurrent part of the layer l but also passed to the next layer $l+1$. To preserve the learned functionality of the layer $l+1$, the kernel $\mathbf{W}_x^{l+1}$ from 3.21 is replaced with a back-projection matrix $\mathbf{Z}_x^{l+1} \in \mathbb{R}^{r_l \times N_{l+1}}$ in equation 3.23, which is calculated as

$$\mathbf{Z}_x^{l+1} = \underset{\mathbf{Z}}{\operatorname{argmin}} \|\mathbf{P}^l\mathbf{Z} - \mathbf{W}_x^{l+1}\|_{\mathcal{F}}^2 \tag{3.26}$$

Now $\mathbf{P}^l\mathbf{Z}_x^{l+1}$ is the rank r_l approximation of $\mathbf{W}_x^{l+1}$ with minimal distance in the Frobenius norm $\|\cdot\|_{\mathcal{F}}$.

Prabhavalkar et al. [4] propose to select the rank r_l of layer l as

$$r_l = \underset{1 \le k \le N_l}{\operatorname{argmax}} \left\{ \frac{\sum_{j=1}^{k} {\sigma_j^l}^2}{\sum_{j=1}^{N_l} {\sigma_j^l}^2} \le \lambda \right\} \tag{3.27}$$

based on the singular values σ_j^l with $\sigma_1^l \le \sigma_2^l \le \ldots \le \sigma_{N_l}^l$ from the SVD of $\mathbf{W}_h^l$ (equation 3.24). Thereby, the rank r_l is selected so that λ-percent of the energy of the singular values in $\mathbf{\Sigma}_h^l$ are kept in the truncation. We call the threshold λ the compression threshold. Prabhavalkar et al. found out through experiments that the compression threshold $\lambda = 0.6$ works best for their application of large vocabulary conversational speech recognition (LVCSR) acoustic modeling. With this threshold they are able to compress the LVCSR acoustic model to one third of its original size with only 5% increase in the word error rate (WER).

To apply this compression method to LSTMs Prabhavalker et al. [4] propose to combine the kernel and the recurrent kernel weight matrices of the input gate, output gate, forget gate, and the cell to a single kernel $\bar{\mathbf{W}}_x^l \in \mathbb{R}^{N_{l-1} \times 4N_l}$ and a single recurrent kernel $\bar{\mathbf{W}}_h^l \in \mathbb{R}^{N_l \times 4N_l}$ as

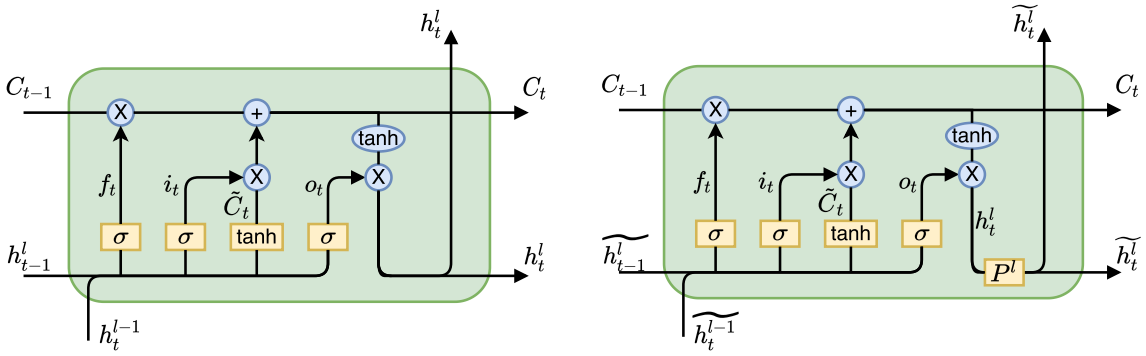
$$\bar{\mathbf{W}}_x^l = [\mathbf{W}_{ix}^l, \mathbf{W}_{ox}^l, \mathbf{W}_{fx}^l, \mathbf{W}_{cx}^l] \tag{3.28}$$

$$\bar{\mathbf{W}}_h^l = [\mathbf{W}_{ih}^l, \mathbf{W}_{oh}^l, \mathbf{W}_{fh}^l, \mathbf{W}_{ch}^l]. \tag{3.29}$$

Now the projection matrix $\mathbf{P}^l$ and the back-projection kernel $\bar{\mathbf{Z}}_x^{l+1} \in \mathbb{R}^{r_l \times 4N_{l+1}}$ and back-projection recurrent kernel $\bar{\mathbf{Z}}_h^l \in \mathbb{R}^{r_l \times 4N_l}$ are calculated in the same way as previously

described. The back-projection matrices $\bar{\mathbf{Z}}_x^l$ and $\bar{\mathbf{Z}}_h^l$ can be decomposed analogously to the composition in equation 3.28 and 3.29.

Figure 3.7b shows where the projection matrix $\mathbf{P}^l$ is added to the standard LSTM cell shown in figure 3.7a. Appendix A.1 provides all equation for the standard LSTM cell as well as the compressed version of these equations.



(a) LSTM Cell [37] (b) Compressed LSTM Cell

Figure 3.7.: The LSTM Cell (a) and its compressed version (b)

We have chosen this compression method because it reduces the number of operations and thus also the run-time of ODANet. This is of decisive importance for the real-time optimization of ODANet on an embedded system. In section 3.2.2 we examine in detail how many operations can be saved when applying the compression method by Prabhavalkar et al. to the ODANet.

In the following, we explain how the compression method effects the number of operations of a RNN in general. For simplicity let us assume the number of input units is equal to the number of output units $N_{l-1} = N_l = n$. Then the size of the weight matrices are $n \times n$ for the kernel and recurrent kernel. Furthermore, the bias vector has size $1 \times n$. The number of operations o is proportional $\sim$ to the size of the weight matrices. To further simplify the estimation we neglect adding the bias vector because the computational expense of the matrix multiplication is much higher than the computational expense of adding the bias vector. This way the number of operations of a general RNN o with n input units and n output units is proportional to

$$o \sim 2 \cdot n \cdot n. \tag{3.30}$$

For the compressed version of the general RNN the weight matrices are $\mathbf{Z}_x^l$, $\mathbf{P}^l$, and $\mathbf{Z}_h^l$, which have the sizes $r_{l-1} \times n$, $n \times r_l$, and $r_l \times n$ respectively. Let us assume $r_{l-1} = r_l = r$. Then the number of operations of the compressed RNN $\tilde{o}$ is proportional to

$$\tilde{o} \sim 3 \cdot n \cdot r. \tag{3.31}$$

For an actual reduction of the number of operations $\tilde{o}$ has to be smaller than o. This is the case if r is smaller than $\frac{2}{3}n$

$$\tilde{o} < o \tag{3.32}$$

$$3 \cdot n \cdot r < 2 \cdot n \cdot n \tag{3.33}$$

$$r < \frac{2 \cdot n}{3}. \tag{3.34}$$

Making the same simplification for an LSTM, which we made previously for the RNN, the number of operation of a LSTM o^* is proportional to

$$o^* \sim 8 \cdot n \cdot n \tag{3.35}$$

because each LSTM has 4 kernels and 4 recurrent kernels of size $n \times n$, one for each the input gate, output gate, forget get, and the cell of the LSTM. In the CLSTM the kernels are replaced with matrices of size $r_{l-1} \times n$ and the recurrent kernels with matrices of size $n \times r_l$. Again assuming that $r_{l-1} = r_l = r$ for simplicity, this results in the following proportion for the number of operations of the CLSTM

$$\tilde{o}^* \sim 9 \cdot n \cdot r. \tag{3.36}$$

where the 9th $n \times r$ matrix is the projection matrix. Therefore, there is an actual reduction of the number of operation $\tilde{o}^*$ in the CLSTMs if r is smaller than $\frac{8}{9}n$

$$\tilde{o}^* < o^* \tag{3.37}$$

$$9 \cdot n \cdot r < 8 \cdot n \cdot n \tag{3.38}$$

$$r < \frac{8 \cdot n}{9}. \tag{3.39}$$

For the actual size reduction in the number of weights from the ODANet to the CODANet with CLSTMs see table 4.6 on page 41.

3.2. Proposed Methods

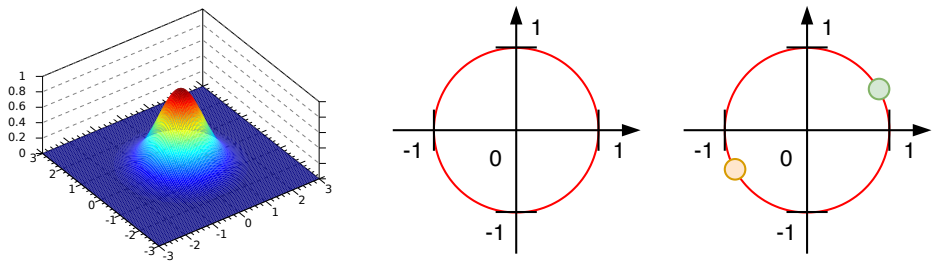
In this section, we present an optimized way to initialize the anchor points and how we apply the previously described compression method to the ODANet resulting in our CODANet.

3.2.1. Sphere Anchor Point Initialization

For the first ADANet experiments we plot the movement of the anchor points in the embedding from epoch to epoch during training in figure 3.9. Based on the observations

on the anchor point movement, which we describe in the following, we thought of two alternative ways to initialize the anchor points for the training of the ADANet.

In the first experiments we initialize the anchor points randomly, which leads to a Gaussian distribution as shown in figure 3.8a for the two dimensional case. In this distribution the initial values of the anchor points have a high probability to be close to the origin and have only a small length. As described in section 3.1.2 the anchor point combination that leads to the maximal distance between the two closest attractor points is selected. For simplicity let us assume that the attractor points are relatively close to the anchor points. We want the ADANet to use all N anchor points and use them to select the features (directions) that are best for separating the sources.



(a) Random (b) Sphere (c) Sphere Pair

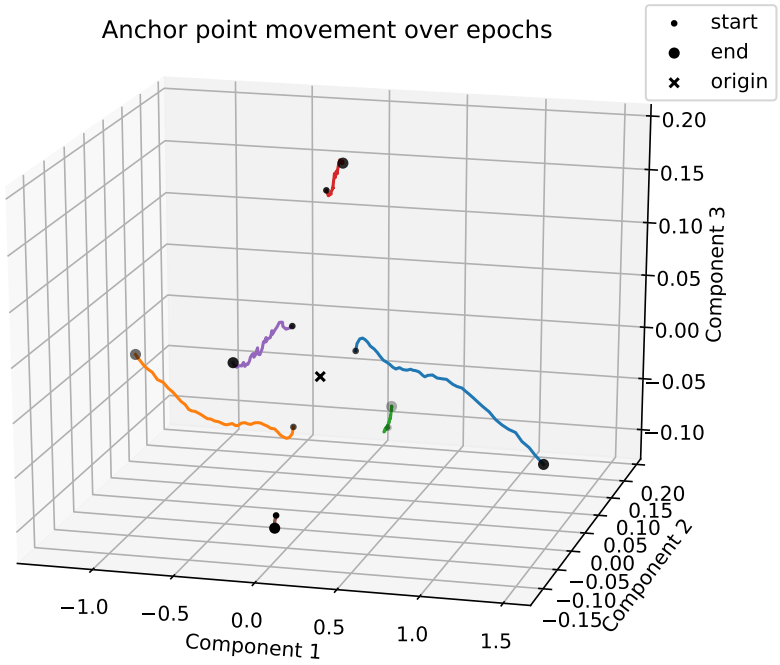
Figure 3.8.: Proposed anchor point initialization methods

A worst case for the anchor point initialization and the learning of good anchor points in the two speaker scenario ($C = 2$) would be that all anchor points are initialized close to the origin except for two anchor points. In this case the two anchor points that are not close to the origin are selected most often as the best anchor points for the separation. In the following back propagation step only the selected anchor points are optimized and therefore only these two anchor points that are not close to the origin will be optimized most often. In this case the other anchor points are neglected during the training. This behavior can be observed in anchor point movement figure 3.9.

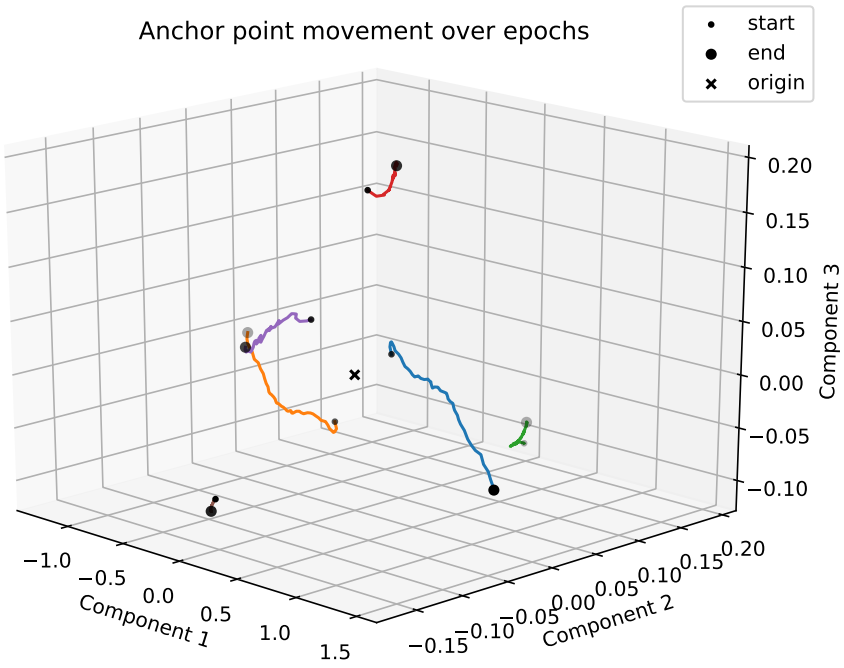
In order to avoid this problem we propose the Sphere anchor point initialization method. In the Sphere anchor point initialization method the anchor points are initialized randomly followed by a normalization step, which normalizes the anchor points to have length 1 in the Euclidean norm. The pseudo code for this method is given in algorithm 1. The first line loops over the number of anchor points N. In line 2 the anchor point $\boldsymbol{\psi}_p \in \mathbb{R}^{1\times K}$ is initialized using the K-dimensional normal distribution $\mathcal{N}_K$ with the mean $\boldsymbol{\mu} \in \mathbb{R}^K$ and the covariance matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{K\times K}$. In line 3 the anchor point is normalized by dividing through its Euclidean norm $\|\cdot\|_2$.

With the Sphere anchor point initialization method the anchor points are randomly distributed on the Sphere with radius one around the origin as shown in figure 3.8b for $K = 2$.

The second anchor point initialization method we propose goes one step further. We



(a) Perspective 1



(b) Perspective 2

Figure 3.9.: PCA of anchor point movement during training from epoch to epoch with random anchor point initialization for ADANet with BLSTMs. **Please note** that the axis of principal component 1 has a much larger span than the axis of the principal components 2 and 3.

Algorithm 1 Proposed Sphere Anchor Point Initialization Method

1: **for** $p \in \{0, 1, \ldots, N\}$ **do**
2: $\quad \boldsymbol{\psi}_p := \mathcal{N}_K(\boldsymbol{\mu}, \boldsymbol{\Sigma})$
3: $\quad \boldsymbol{\psi}_p := \boldsymbol{\psi}_p / \|\boldsymbol{\psi}_p\|_2$
4: **end for**

call this method the Sphere Pair anchor point initialization method because we initialize pairs of anchor points randomly on the sphere with radius one around the origin. This is done by initializing the first $N/2$ anchor points using the previously described Sphere anchor point initialization method. The second half of the anchor points are then initialized as the point-symmetric partners of the first half of the anchor points. This is done by multiplying each of the anchor points in the first half with -1 as shown in line 6 of algorithm 2.

Algorithm 2 Proposed Sphere Pair Anchor Point Initialization Method

1: **for** $p \in \{0, 1, \ldots, N/2\}$ **do**
2: $\quad \boldsymbol{\psi}_p := \mathcal{N}_K(\boldsymbol{\mu}, \boldsymbol{\Sigma})$
3: $\quad \boldsymbol{\psi}_p := \boldsymbol{\psi}_p / \|\boldsymbol{\psi}_p\|_2$
4: **end for**
5: **for** $p \in \{N/2, N/2+1, \ldots, N\}$ **do**
6: $\quad \boldsymbol{\psi}_p := -1 * \boldsymbol{\psi}_{p-N/2}$
7: **end for**

3.2.2. Compressed Online Deep Attractor Network (CODANet)

In this section, we describe how we applied the compression method by Prabhavalkar et al. [4] described in section 3.1.4 to the ODANet to reduce the size of the network and the number of operations, which should result in a smaller processing time for a single frame.

To compress the ODANet we replace the LSTM layers with CLSTM layers. Prabhavalkar et al. do not explicitly describe what to do with the first and the last layer. For the first layer we use the CLSTM with out compressing the kernel weights of the LSTM. If we use a CLSTM for the last LSTM layer in the ODANet the output of this layer will be the projected output $\widetilde{\mathbf{h}}_t^l$, which is not compatible with the expected input to the fully-connected layer following the LSTM layers.

We see the following options to deal with this problem:

1. Do not compress the output of the last LSTM layer.
2. Add an additional layer between the last LSTM layer and the fully-connected layer that does the back-projection of $\widetilde{\mathbf{h}}_t^l$ with the inverse of $\mathbf{P}^{l=4}$.
3. Use equation 3.26 to calculate a back-projection kernel based on the kernel of the fully-connected layer.

Option 3 is selected because it continues the idea of the compression method of Prabhavalkar et al. and thus leads to a further compression of the network. The second option is not selected because adding an additional layer between the last LSTM and the fully-connected layer would increase the number of operations in the network again. The first option of not compressing the output of the last LSTM layer is not selected because to implement this we would need to insert the projection matrix P^l at two different places in the LSTM architecture.

In the following we calculate the number of weights for the ODANet and the CODANet. The number of weights for the ODANet is the sum of

- the number of weights in the LSTM layers

$$\text{size(LSTMs)} = \sum_{l=1}^{4} (\underbrace{4 \cdot N_{l-1} \cdot N_l}_{\text{kernel}} + \underbrace{4 \cdot N_l \cdot N_l}_{\text{recurrent kernel}} + \underbrace{4 \cdot N_l}_{\text{bias}}) \tag{3.40}$$

  where $N^0 = F$ is the input dimension, which is the number of frequency bins F and N_l is the number of units in the l-th LSTM layer,

- the number of weights in the dense layer

$$\text{size(dense)} = N_4 \cdot K \cdot F \tag{3.41}$$

  with K the embedding dimension,

- and the number of weights for the anchor points

$$\text{size(anchors)} = K \cdot N \tag{3.42}$$

  where N is the number of anchor points.

$$\text{size(ODANet)} = \sum_{l=1}^{4} (4 \cdot N_{l-1} \cdot N_l + 4 \cdot N_l \cdot N_l + 4 \cdot N_l) + N_4 \cdot K \cdot F + K \cdot N \tag{3.43}$$

For example with $N_l = 600$ for $l = 1, \ldots, 4$, $F = 129$, $K = 20$, and $N = 4$ this results in

$$\text{size(ODANet)} = \underbrace{10,399,200}_{\text{size(LSTMs)}} + \underbrace{1,550,580}_{\text{size(dense)}} + \underbrace{80}_{\text{size(anchors)}} = 11,949,860 \tag{3.44}$$

For the CODANet we replace all LSTM layers with CLSTM layers, which changes the equations 3.40 in the following way. The number of weights in the CLSTMs are

$$\text{size(CLSTMs)} = \sum_{l=1}^{4} (\underbrace{4 \cdot r_{l-1} \cdot N_l}_{\text{comp. kernel}} + \underbrace{N_l \cdot r_l}_{\text{projection matrix}} + \underbrace{4 \cdot r_l \cdot N_l}_{\text{comp. recurrent kernel}} + \underbrace{8 \cdot N_l}_{\text{bias}}) \tag{3.45}$$

with r_l the rank of layer l and $r_0 = F$ the input dimension. The dense layer now takes the

projected output of the fourth CLSTM layer, which also reduces its size to

$$\text{size(comp. dense)} = r_4 \cdot K \cdot F \tag{3.46}$$

The rank r_l for each layer is determined using equation 3.27 and depends on the singular values of the recurrent kernel $\mathbf{W}_h^l$. For example with a compression threshold $\tau = 0.7$ the compressed weights of the previous example with $N_l = 600$ for $l = 1, \ldots, 4$, $F = 129$, $K = 20$, and $N = 4$ could have the following ranks:

$$r_1 = 251, \quad r_2 = 234, \quad r_3 = 205, \quad r_4 = 177 \tag{3.47}$$

This results in a total number of weights for the CODANet of

$$\text{size(CODANet)} = \underbrace{4,576,200}_{\text{size(CLSTMs)}} + \underbrace{459,240}_{\text{size(comp. dense)}} + \underbrace{80}_{\text{size(anchors)}} = 5,035,520 \tag{3.48}$$

which is a reduction of more than 50% compared to the size of the ODANet. In the results section 4.3.4 we see how this reduction in the size of the network effects the processing time and the performance of the CODANet.

4. Evaluation

In this chapter, we first explain the experimental setup, which includes the used dataset and its preprocessing for the training, as well as the hardware we use in our experiments. Second, we describe how we measure the speech separation performance and the processing time per frame. Finally, we present the results of our experiments.

4.1. Experimental Setup

In this section, we present the used WSJ0-2mix dataset, its preprocessing for the training, the postprocessing of network output, and the hardware we use to train our NNs and to evaluate the processing time, which are a desktop PC and the NVIDIA Jetson Nano.

Dataset

For the training and evaluation of our NNs we use the WSJ0-2mix dataset introduced in [5] and also used in the mentioned related work on speech separation with NNs [2, 3, 5–7, 12–20, 22–24]. Generated from the Wall Street Journal (WSJ0) corpus the WSJ0-2mix dataset contains time domain mixture utterances of averagely 6 seconds as well as the ground truth speech signal of the two speakers from which the mixture utterances are generated. The WSJ0-2mix is divided into the following three sets:

1. A 30 hour training set generated by randomly selecting utterances of different speakers from the WSJ0 training set `si_tr_s`, and mixing them at various signal-to-noise ratios between 0 dB and 10 dB.

2. A 10 hour validation set generated just like the training set.

3. A 5 hour evaluation set generated using utterances from 16 speakers from the WSJ0 development set `si_dt_05` and evaluation set `si_et_05` while using different speakers from those in the previous training and validations sets.

While this dataset has become a current standard in the speech separation research, its disadvantage is that the WSJ0 contains professional recordings without background noise of professional speakers reading from the Wall Street Journal newspaper, which are very different from everyday audio recordings at cocktail parties.

Preprocessing and Postprocessing for Training and Evaluation

For training our DANet implementation versions, which operate in the time-frequency domain, we preprocess the waveform audio data in the WSJ0-2mix dataset once by downsampling it and applying an STFT with the parameters shown in table 4.1. The resulting complex spectrograms are saved into a h5py-file, which allows fast access to the spectrogram data for the training of the networks [38].

Preprocessing Parameter	Value
Sampling rate	8 kHz
Window length	32 ms
Hop size	8 ms
Window	Root Hann Window

Table 4.1.: Preprocessing parameters

For the training of our implementations we split each utterance spectrogram into meta-frames of 400 consecutive time frames. Because many utterances start with silent frames, we apply a random offset at the beginning. This offset is a random number between zero and the remainder of the division of the total number of time frames per utterance with the meta-frame size.

For the speech separation performance evaluation each utterance is zero padded at the end, so that the utterance is exactly divisible by the meta-frame size. After the processing of the network, the meta-frames for each utterance are concatenated again. Finally a we apply an inverse short-time Fourier transform (ISTFT) and evaluate the speech separation performance of the predicted time domain signals using the SI-SNR and SDR presented in section 4.2.

Hardware

We train our implementations on a desktop PC, with a NVIDIA Corporation GP102 TITAN X GPU and an Intel® Core™ i7-5930K Prozessor central processing unit (CPU), trying optimize it for real-time speech separation on the NVIDIA Jetson Nano. The NVIDIA Jetson Nano is an embedded computer board equipped with

- a 128-core Maxwell GPU with a performance of up to 512 GFLOPS (FP16) at 921 MHz,
- a quad-core ARM A57 CPU operating at 1.43 GHz,
- 4 GB on-board memory with 4ch x 16-bit LPDDR4 at 1600 MHz with a peak bandwidth of 25.6 GB/s [11].

Especially the GPU should make this embedded computer suitable for running NN implementations. Because the NVIDIA Jetson Nano GPU processes FP16 operations, we also evaluate our best ODANet experiment number 3.2 with FP16 weights in the NN instead of FP32 weights, which shows that the SI-SNR performance is almost the same for FP16 and FP32.

4.2. Evaluation Methods

To measure the speech separation performance of the implementations we use the SI-SNR and the SDR, which are calculated in the time domain. The processing time for a single frame is measured on the hardware presented in section 4.1 using the system time in Python and includes the time for the fast Fourier transform (FFT) and the inverse fast Fourier transform (IFFT) of the frame.

4.2.1. Scale-invariant Source to Noise Ratio (SI-SNR)

Following the notation of Luo et al. [16] the SI-SNR is defined as

$$\text{SI-SNR} := 10 \log_{10} \frac{\|\mathbf{s}_{\text{target}}\|^2}{\|\mathbf{e}_{\text{noise}}\|^2} \tag{4.1}$$

with

$$\mathbf{s}_{\text{target}} = \frac{\langle \hat{\mathbf{s}}, \mathbf{s} \rangle \cdot \mathbf{s}}{\|\mathbf{s}\|^2} \tag{4.2}$$

as the orthogonal projection of $\hat{\mathbf{s}}$, the estimated audio signal, onto $\mathbf{s}$, the true audio signal, and

$$\mathbf{e}_{\text{noise}} = \hat{\mathbf{s}} - \mathbf{s}_{\text{target}} \tag{4.3}$$

as the difference between $\hat{\mathbf{s}}$ and $\mathbf{s}_{\text{target}}$ as shown in figure 4.1. To ensure scale-invariance both $\mathbf{s}$ and $\hat{\mathbf{s}}$ are normalized to have a mean of zero [16].

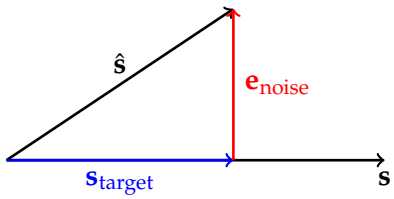


Figure 4.1.: Illustration of $\mathbf{s}_{\text{target}}$ as the orthogonal projection of $\hat{\mathbf{s}}$ onto $\mathbf{s}$ and $\mathbf{e}_{\text{noise}}$ as the difference between $\hat{\mathbf{s}}$ and $\mathbf{s}_{\text{target}}$.

4.2.2. Source-to-Distortion Ratio (SDR)

A similar measure, which is used in the literature, is the SDR, which is defined by Vincent et al. [39] and considers the following error sources.

- $\mathbf{e}_{\text{interf}}$ the interference error caused by the interference of the other signal sources e.g. other speakers,
- $\mathbf{e}^*_{\text{noise}}$ the noise error caused by sensor noise,
- $\mathbf{e}_{\text{artif}}$ the artifacts error, which is the remaining error not caused by interference or noise.

Based on these errors Vincent et. al. [39] define the following quality measures

- the source-to-distortion ratio (SDR)

$$\text{SDR} := 10\log_{10}\frac{\|\mathbf{s}_{\text{target}}\|^2}{\|\mathbf{e}_{\text{interf}} + \mathbf{e}^*_{\text{noise}} + \mathbf{e}_{\text{artif}}\|^2}\,, \tag{4.4}$$

- the source-to-interferences ratio (SIR)

$$\text{SIR} := 10\log_{10}\frac{\|\mathbf{s}_{\text{target}}\|^2}{\|\mathbf{e}_{\text{interf}}\|^2}\,, \tag{4.5}$$

- the sources-to-noise ratio (SNR)

$$\text{SNR} := 10\log_{10}\frac{\|\mathbf{s}_{\text{target}} + \mathbf{e}_{\text{interf}}\|^2}{\|\mathbf{e}^*_{\text{noise}}\|^2}\,, \tag{4.6}$$

- the sources-to-artifacts ratio (SAR)

$$\text{SAR} := 10\log_{10}\frac{\|\mathbf{s}_{\text{target}} + \mathbf{e}_{\text{interf}} + \mathbf{e}^*_{\text{noise}}\|^2}{\|\mathbf{e}_{\text{artif}}\|^2}\,. \tag{4.7}$$

We only report the SDR in our evaluation for specific experiments, because calculating the quality measures by Vincent et al. takes more than two times as long as calculating the SI-SNR.

4.2.3. Measurement of Processing Time per Frame

An important criterion for real-time processing is the time it takes to process a single frame using the ODANet or the CODANet. For real-time processing it is a necessary requirement that the processing time for each frame is less than the hop size of 8 *ms*. However, for real-time processing this is not a sufficient requirement. To measure the processing time of a single frame on the PC or the NVIDIA Jetson Nano, which are presented in section 4.1, with the ODANet or the CODANet we use a script that splits the time domain data into frames. For each of these frames we measure the processing time by saving the system time into a `start_time` variable. Followed by the processing of the frame, which includes the steps of a FFT, the prediction of the speech sources using

the ODANet or the CODANet, and the IFFT of these predicted sources. After the processing, the system time is subtracted from the `start_time` resulting in the processing time, which is saved for each frame. In the results section we evaluate this processing time per frame with different statistical methods like the mean and variance. An important measurement is also the number of frames exceeding the output buffer size (e.g. 8 *ms*), because if the output buffer for the sound card is not filled in time, then click sound could be heard in the audio output.

Our implementations of ODANet and CODANet use a sampling rate of 8 *kHz*, as do the implementations in [2, 6, 7], in order to keep the input data quantity and thus the computational complexity small. In our evaluation of the processing time, we do not take into account the possible effort required to downsample the audio signal to 8 *kHz*, which is necessary if the used sound card does not support this sampling rate. We do not take this processing time into account because we expect it to be relatively small.

4.3. Results

In this section, we present our findings on the proposed anchor point initialization methods, the impact of the optimizer and the number of LSTM units on the ADANet, as well as the hyper-parameter tuning for the ODANet, and the resulting run-time optimization of the CODANet. Furthermore, we present the run-time and real-time capability of our implementations based on the hardware. At the end of this section we present additional findings.

4.3.1. Anchor Point Initialization Methods

In this section, we present our results for the anchor point initialization methods proposed in section 3.2.1, which we evaluate in a first series of experiments. This section is split into preliminary and final results, which lead to different conclusion. Both results are presented because based on the preliminary results we decided that Sphere is the best anchor point initialization method for the following experiments, however the final results, which were generated with an improved method to select the best epoch of a training, lead to the conclusion the Sphere Pair is the best anchor point initialization method. But we were not able to integrate this final conclusion into our later experiments.

Preliminary Results on Anchor Point Initialization Methods

For the evaluation of the anchor point initialization method we conduct two experiments for each initialization method while the other parameters shown in table 4.2 are fixed. We use BLSTMs with 600 units per direction in each of the four BLSTM layers, 6 anchor points, and an embedding dimension of 20 to compare the results of our implementation to the results in the paper [7] by Luo et al., in which they also used the same set of

parameters for the ADANet. Furthermore, we use the Root Mean Square Propagation (RMSprop) optimizer with a learning rate of 0.0001 and early stopping if the validation loss does not decrease for 10 epochs. For each experiment we selected the 3 epochs with the lowest validation loss for each experiment and evaluated the ADANet weights after these epochs on the test set. [1]

Parameter	**Value**
Meta-frame size	400
LSTM directonality	bidirectional
Number of units per direction in LSTM Layers	600
Number of anchor points	6
Embedding dimension	20
Batch size	32
Loss function	equation 3.6
Optimizer	RMSprop
Learning rate	0.0001
Patience for early stopping (on val. loss)	10
Reduce learining rate on plateau	not used

Table 4.2.: Fixed parameters of first experiment series evaluating the anchor point initialization methods.

This setup leads to the results shown in table 4.3. The first and second column specifies an ID and a label for each training run, respectively. For each experiment the table 4.3 only shows the results for the epoch with the best mean SDR on the test set. For each initialization method we train the ADANet two times to see the effect of the randomness in the weight initialization, not only in the anchor points, but also in the weights of the LSTM layers and the dense layer, which generate an initially random mapping of the time-frequency bins into the embedding space.

Discussing the preliminary results of these experiments, which we call the first experiment series, we see that the experiment with the label Random1 has the best SDR and SI-SNR results on the test set. But we also see that the second experiment with the label Random2 has the worst SDR and SI-SNR value on the test set in the whole experiment series. In comparison, the experiments Sphere1 and Sphere2 have almost as good SDR and SI-SNR values as experiment Random1. In contrast, the experiments SpherePair1 and SpherePair2 are only slightly better than the worst experiment Random2. However, these experiments with the Sphere Pair anchor point initialization method reach their optimum based on the validation loss already after 29 and 28 epochs. So they need less

[1] In later experiments we see that there is a discrepancy between the validation loss and the SI-SNR and SDR on the validation set, which leads us to the conclusion that the validation loss is not a good criterion for early stopping or selecting the best epoch in a training run. For the final results we therefore continued all training runs till epoch 50 without early stopping and used the validation SI-SNR for selecting the best epoch.

ID	Label	Epoch	SDR on test set	SI-SNR on test set	Validation loss	Training loss
1.1	Random1	50	7.5	6.3	0.239	0.068
1.2	Random2	39	7.0	5.8	0.240	0.078
1.3	Sphere1	44	7.4	6.1	0.235	0.068
1.4	Sphere2	36	7.4	6.2	0.246	0.073
1.5	SpherePair1	29	7.2	5.9	0.246	0.080
1.6	SpherePair2	28	7.2	5.9	0.241	0.085

Table 4.3.: Preliminary experimental results for the different anchor point initialization methods Random, Sphere and Sphere Pair with two experiments for each initialization method with early stopping and selection of the best epoch based on the validation loss.

time to reach their optimum than the other experiments. In comparison to this, the experiments with the Sphere anchor point initialization method are in the middle between the experiments with the random initialization of anchor points and the experiments with the Sphere Pair initialization method regarding the number of epochs it takes for them to reach their optimum.

The results with early stopping and selection of the best epoch based on the validation loss can be summarized as follows. If we reduce the randomness of the initialization of the anchor points, the variance in the results is reduced and the optimum based on the validation loss is reached earlier.

Final Results on Anchor Point Initialization Methods

We continue all training runs reported previously till epoch 50 without early stopping and used the validation SI-SNR for selecting the best epoch, because in an later experiment we see that the validation loss is not a good criterion for early stopping or selecting the best epoch in a training run, as there is a discrepancy between the validation loss and the SI-SNR and SDR on the validation set as shown in figure 4.2. In this figure we see that even though the validation loss does not go far below 0.24 after epoch 10, the training loss and validation SI-SNR still decrease. We have no direct explanation for this discrepancy and assume that there is a programming error that we have not found. That is way we use the SI-SNR on the validation set as a workaround.

In table 4.4 we see the final experimental results for the evaluating of the different anchor point initialization methods, which show that the Sphere Pair initialization method leads to the best SI-SNR results on the test set in average. However, as mentioned previously we conducted the experiment series three and four with Sphere instead of Sphere Pair as the initialization method based on our preliminary results of this first experiment series.

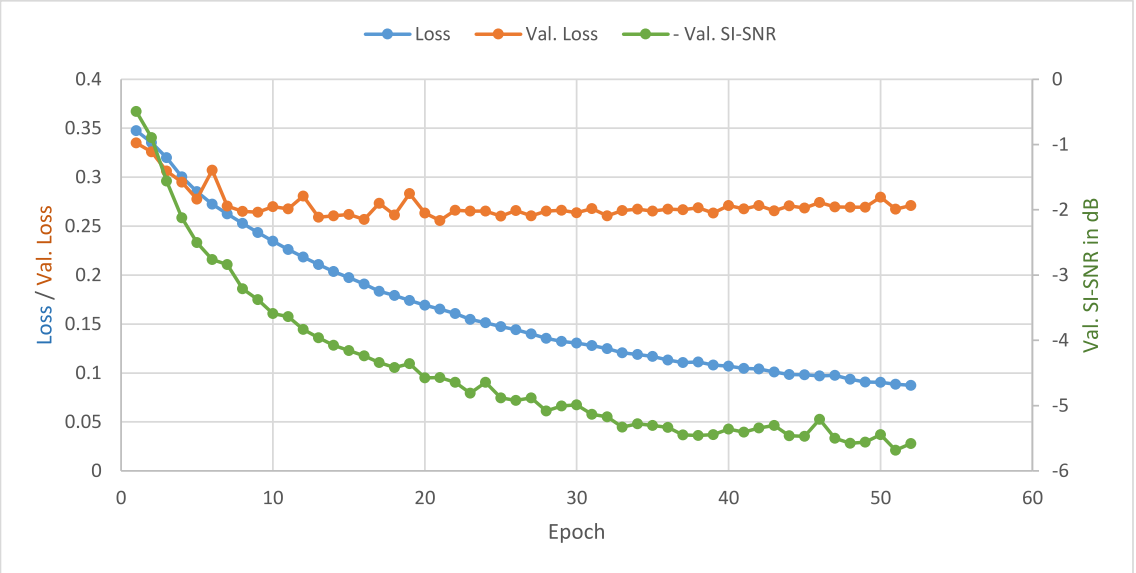


Figure 4.2.: Discrepancy between validation loss and validation SI-SNR shown for one example ADANet experiment with unidirectional LSTMs with 600 units in each layer, 6 anchor points and embedding dimension 20.

ID	Label	Epoch	SDR on test set	SI-SNR on test set
1.1	Random1	50	7.5	6.3
1.2	Random2	50	7.4	6.1
1.3	Sphere1	44	7.4	6.1
1.4	Sphere2	50	7.9	6.7
1.5	SpherePair1	49	7.8	6.7
1.6	SpherePair2	50	7.9	6.8

Table 4.4.: Final experimental results for the different anchor point initialization methods. Training each experiment for 50 epochs and selecting the best epoch based on SI-SNR on the validation set.

ID	Units per direction in LSTMs	Initialization method of anchor points	Optimizer	Epoch	SDR on test set	SI-SNR on test set
2.1	300	Sphere	RMSprop	50	6.5	5.1
2.2	300	Sphere	ADAM	29	6.4	5.0
2.3	300	Sphere	ADAM*	15	5.8	4.1
2.4	600	Sphere	ADAM	28	6.6	5.2
2.5	600	Random	ADAM	21	6.3	4.8
2.6	600	Random	ADAM	31	6.8	5.5

Table 4.5.: Experimental results of evaluation of different optimizer settings and number of units per direction in LSTMs of the ADANet. * with reduce learning rate on plateau. Best epoch is selected base on validation loss with early stopping similar to preliminary results of the first test series.

4.3.2. Impact of Optimizer and Number of LSTM Units on ADANet

Because there is a difference between our first results presented previously and the results reported in the ADANet paper by Luo et al. [7], we conducted as second series of experiments analyzing if the number of units per direction in the BLSTMs or different optimizers could cause this differences. However, the results of this second experiment series, which we present in detail in this section, validate the previously presented results of our first experiment series. In the chapter 5 we discuss possible other reasons for the observed difference.

The ADANet with 6 anchor points by Luo et al. achieves a SI-SNR of 9.6 dB. Our best result of 6.8 dB SI-SNR is far below. Looking for differences in the implementation we see that Luo et al. use the Adaptive Moment Estimation (ADAM) optimizer and we use the RMSprop optimizer. Furthermore, Luo et al. do not directly specify if the number of 600 units in the BLSTM layers are per direction or in total. That is why we perform experiments with 300 units per direction in the BLSTMs, which results in a total of 600 units per BLSTM layer. We only report results with early stopping based on the validation loss for the second experiment series. Due to time constrains we are not able to continue this second experiment series similar to the first experiment series without early stopping based on the validation loss and selecting the best epoch based on the SI-SNR instead.

Tabel 4.5 shows the results of the second experiment series. The first 3 experiments are conducted with 300 units pre direction in all four BLSTM layers. For the first experiment of this series we continued to use RMSprop as the optimizer to investigate the influence of the units per direction in the LSTMs independently of the optimizer. For this first experiment with 300 units per direction in the BLSTM layers the SI-SNR is 1 dB less compared to the results of the experiments Sphere1 and Sphere2 of table 4.3, in which the BLSTMs have 600 units per direction. We perform two more experiments (2.2 and 2.3) with 300 units per direction in the BLSTM and the Sphere anchor point initialization method using ADAM as the optimizer to exclude that the first experiment of this test

series randomly got a bad result. However, experiments 2.2 and 2.3 do not achieve better results than experiment 2.1. Experiment 2.3 is conducted with a reduction of the learning rate with the factor 0.5 if the validation loss does not improve for 5 epochs similar to [7]. But this is not constructive as shown in figure 4.3, which shows the validation loss for this experiment.

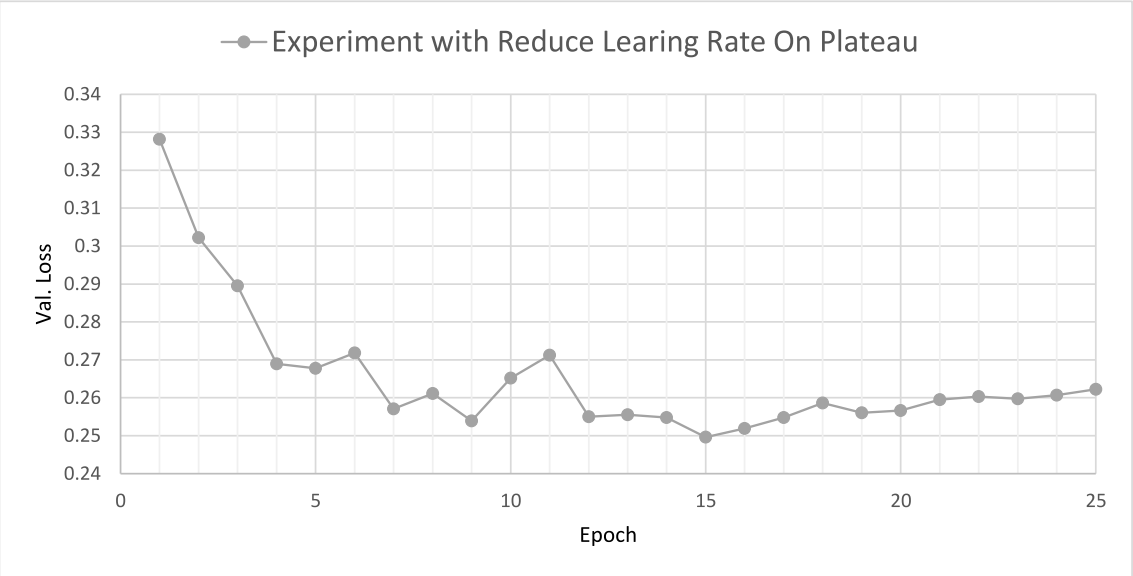


Figure 4.3.: Validation loss of experiment 2.3 with learning rate reduction after epoch 14 from 0.0001 to 0.00005 and after epoch 20 to 0.000025.

Second, we evaluate the influence of the optimizer with experiments 2.4, 2.5, and 2.6. Compared to the preliminary results of experiments 1.1, 1.2, and 1.4 in table 4.3, which use the RMSprop optimizer, experiments 2.4, 2.5, and 2.6, which use the ADAM optimizer, have about 10% lower SDR and SI-SNR (comparing 1.1 with 2.6, 1.2 with 2.5, and 1.4 with 2.4). These results suggest that RMSprop is better suited than ADAM for training the ADANet. This could be because ADAM is a combines the ideas of RMSprop and Momentum [40, 41] and the Momentum component of ADAM does not seem to be helpful for training the ADANet.

Summarizing the results of the second series of experiments presented in this section, we conclude that the number of 600 units in the BLSTM layers described by Luo et al. [7] are probably per direction and that RMSprop is better suited than ADAM for training the ADANet.

4.3.3. Hyper-parameter Tuning of ODANet

The impact of hyper-parameter tuning on the performance and run-time of the ODANet is presented in this section. As suggested in [2] we train the ADANet with unidirectional LSTMs with the different hyper-parameters and use the trained weights due to time constrains directly in the ODANet without curriculum training.

The hyper-parameters we evaluate are the number of units in the LSTM layers, the embedding dimension, and the number of anchor points. The other parameters remain fixed and are shown in Table A.1 in appendix. Based on our preliminary results presented in section 4.3.1 we use Sphere as the anchor point initialization method. Additionally, the results presented in the following are based on training each set of hyper-parameters for 50 epochs and select the best epoch based on the SI-SNR on the validation set.

ID	No. of Units in LSTMs	Embedding Dimension	No. of Anchor Points	No. of weights in million	Processing Time mean in *ms*	SI-SNR in *dB*
3.1	600	20	6	12	7.0	5.3
3.2	600	20	4	12	6.9	5.5
3.3	600	15	6	12	6.9	5.1
3.4	600	15	4	12	6.9	5.1
3.5	500	20	6	9	5.6	4.8
3.6	500	20	4	9	5.5	4.9
3.7	400	20	6	6	4.6	4.4
3.8	400	20	4	6	4.3	4.6

Table 4.6.: Results of hyper-parameter tuning. Processing time per frame measured on the PC with only CPU[2]. SI-SNR results are for the ODANet on the test set.

The results of the hyper-parameter tuning are shown in Table 4.6. As expected, if we decrees the hyper-parameter values, then the number of weights in the ADANet will also decrease, which is equal to the number of weights in the ODANet. Likewise, the average processing time per frame decreases with a reduction in number of weights in the ODANet. However, if we reduce the network size the speech separation performance measured with the SI-SNR also decreases.

It can be seen that the number of units in the LSTMs has a great direct influence on the number of weights and the processing time. In comparison, the influence of the embedding dimension and the number of anchor points on the processing time and the number of weights is negligible.

We observe that the SI-SNR value for the ODANet with 4 anchor points is always better than for the ODANet with 6 anchor points, in case of the same number of units in the LSTMs and the same embedding dimension. For experiment 3.4 and 3.3 this cannot be seen in the rounded SI-SNR values. But if we look at the next decimal places we see that the SI-SNR for experiment 3.4 with 4 anchor points is 5.063 dB and the SI-SNR for experiment 3.3 with 6 anchor points is 5.052 dB.

In terms of the processing time saved, we see that from experiment No. 3.2 to No. 3.8 we can reduce the processing time by 37% from 6.9 *ms* to 4.3 *ms* while the SI-SNR is reduced by 16% from 5.5 dB to 4.6 dB.

[2]In section 4.3.5 we show that our network is faster with only the CPU compared to running it on the GPU.

In the following section we see an alternative way to reduce the processing time while trying to maintain the speech separation performance.

4.3.4. Run-time Optimization with Compression

To analyze the influence of compression on the performance and run-time we conduct a fourth experiment series. In these experiments we compress the weights of the best previously trained ADANet with unidirectional LSTMs in particular the weights from experiment number 3.2 and use them in the CODANet to evaluate the performance of the CODANet and its run-time.

ID	Compression Threshold	No. of weights in million	Processing Time mean in ms	SI-SNR in dB
4.1	1.0	13	7.6	5.5
4.2	0.9	10	6.1	5.1
4.3	0.8	7	4.9	4.9
4.4	0.7	5	4.2	4.7
4.5	0.6	3	3.8	4.5
4.6	0.5	2	3.7	3.9
3.2	ODANet	12	6.9	5.5

Table 4.7.: Results of CODANet on the PC with weights compressed from experiment 3.2 the ODANet with 600 LSTM units, embedding dimension 20 and 4 anchor points.

Table 4.7 shows the experimental results of compressing the weights from experiment 3.2 with different compression thresholds. The first experiment 4.1 uses the compression threshold 1.0, which is equivalent to no compression. This experiments verifies that that the CODANet does exactly the same prediction as the ODANet. The SI-SNR for the ODANet with 600 LSTM Units, the embedding dimension 20, and 4 anchor points and for the CODANet of this network with a compression threshold of 1.0 are 5.53633 dB. This SI-SNR value is exactly the same for the ODANet and the CODANet up to the previously specified decimal places.

As explained in section 3.2.2 the number of weights in the CODANet goes down if we reduce the compression threshold, which influences the rank of the CLSTM layers in the CODANet. In Table 4.8 we show how changing the compression threshold τ effects the rank r_l of each LSTM layer l when compressing the weights of experiment 3.2. The number of weights in the CODANet are given in column 4 of table 4.7 and are calculated based on the ranks r_l given in table 4.8.

Similar to the previous test series, if we reduce the number of weights, then the processing time per frame and the SI-SNR decreases. Reducing the processing time per frame is desired while we want to maintain the SI-SNR. If we compare the results of the CO-

Compression Threshold	r_1	r_2	r_3	r_4
1.0	600	600	600	600
0.9	443	444	428	403
0.8	336	328	304	273
0.7	251	234	205	177
0.6	182	157	127	107
0.5	124	95	69	61

Table 4.8.: Rank for each layer of the CODANet with different compression threshold.

DANet from experiment 4.4 with a compression threshold of 0.7 and the ODANet from experiment 3.2, then we see that we can reduce the processing time by 39% from 6.9 *ms* to 4.2 *ms* while the SI-SNR is reduced by 15% from 5.5 *dB* to 4.7 *dB*. This is better than the result we achieved with the hyper-parameter tuning.

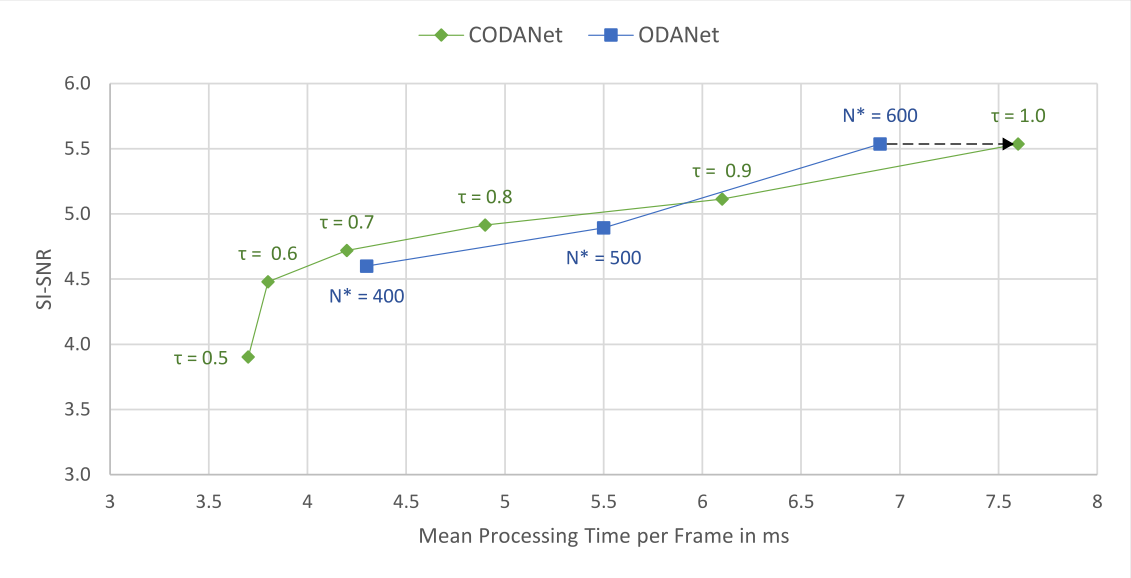


Figure 4.4.: SI-SNR over processing time per frame on the PC for ODANet with 4 anchor points (blue) and CODANet (green). We aim for a high SI-SNR with a low processing time. The labels indicate the compression threshold τ for the CODANet and the number of LSTM units N^* in each layer of the ODANet. The dashed arrow indicates that the weights from this ODANet experiment are used in all CODANet experiments.

In Figure 4.4 we show the SI-SNR performance over the processing time per frame for the hyper-parameter tuning and CODANet experiments. The dashed arrow from the ODANet with 4 anchor points and 600 units in the LSTMs to the CODANet with the compression threshold 1.0 represents that the weights from this ODANet experiment (3.2) are used in all CODANet experiments. As expected, the CODANet with compression threshold 1.0 has the same SI-SNR performance as the ODANet on which it is based. However, the processing time per frame is higher for the CODANet with compression threshold 1.0 because it additionally contains the projection matrices in the CLSTMs compared to

the ODANet.

For the compression thresholds 1.0 to 0.7 we see a nearly linear decrease of the SI-SNR with a decreasing mean processing time per frame of $0.24\frac{dB}{ms}$ with an decreasing compression threshold. If we compare this to the decrease of the SI-SNR over the mean processing time for the ODANet with 4 anchor points from the experiment with 600 units in the LSTM layers to experiment 400 units in the LSTM layers, then we see that the SI-SNR decreases with $0.35\frac{dB}{ms}$, which is faster than for the compression experiments. However, if the compression threshold is decreased below 0.7, then the SI-SNR decreases much faster than the processing time per frame.

In general, if we compare the results of of the hyper-parameter tuning (in blue) to the results of the compression method (in green) in figure 4.4, then we see that for similar processing times the SI-SNR for the CODANet is slightly better than the SI-SNR of the ODANet, except for the ODANet with 4 anchor points and 600 LSTM units in each layer. Furthermore, for each experiment in the hyper-parameter tuning we have to train the network for 19 to 25 hours on the PC to get the network weights for a specific set of hyper-parameters. In comparison, we get the weights for the CODANet much faster with the compression method. It only takes view seconds to compress the weights of a previously trained ADANet. In Appendix A.4 on page 64 we provide a similar plot to figure 4.4, which also includes the ODANet experiments with 6 anchor points.

To summarize, compressing a network is much faster than the hyper-parameter tuning and with the CODANet we get even slightly better results then with the hyper-parameter tuning in terms of the SI-SNR speech separation performance over the processing time per frame.

4.3.5. Hardware Related Run-time and Real-time Capability

We evaluate the processing time per frame of the CODANet on the PC and the NVIDIA Jetson Nano with and without using the GPU. Furthermore, we test the influence of the different LSTM implementations in the Keras Version 2.2.4 [42]. The LSTM implemenation 1 performce the matrix multiplications of the input gate, the forget get, the output gate and the cell of the LSTM individually. The LSTM implemenation 2 combines the individual kernels of the input gate, the forget get, the output gate and the cell to one singular kernel similar to equation 3.28 on page 23. This composed kernel is then multiplied with the input and the output of this multiplication is decomposed again afterwards. The same is done for the recurrent kernel. This implementation 2 is supposed to work better with the NVIDIA CUDA® Deep Neural Network library (cuDNN) for GPUs [43].

If we compare the processing time in Table 4.9 of a specific set of parameter on the PC to the same setup on NVIDIA Jetson Nano, then we see that the processing of a single frame takes 3 to 4 times longer on the NVIDIA Jetson Nano compared to the PC. This results are a little better than the results of the benchmark analysis by Bianco et al. [28], in which they compared the run-time between the NVIDIA Titan X and the NVIDIA Jetson

Hardware	Processing Unit	LSTM Impl.	Compression Threshold 0.5	0.6	0.7	0.8	0.9	1.0
PC	CPU	1	3.7	3.8	4.2	4.9	6.1	7.6
		2	3.8	4.5	5.1	5.9	6.9	8.1
	GPU	1	8.6	8.6	8.5	8.6	8.5	8.7
		2	7.7	7.7	7.7	7.6	7.8	7.7
Jetson	CPU	1	10.5	12.9	15.6	19.6	24.4	31.5
		2	10.2	12.8	15.6	19.8	25.1	32.4
	GPU	1	32.9	34.5	37.3	35.8	36.3	36.4
		2	28.0	27.4	27.5	27.7	28.3	28.9

Table 4.9.: Mean processing time per frame in *ms* of CODANet on NVIDIA Jetson Nano and PC with and without GPU and for different Keras LSTM implementations.

TX1 for different DNN architectures for image recognition. Bianco et al. observe that on the NVIDIA Jetson TX1 it takes 5 to 35 times longer to process a single image than on the NVIDIA Titan X.

Looking at the influence of using the GPU or only using the CPU we see that when the GPU is used the processing times per frame is almost independent of the compression threshold and depends more on the type of Keras LSTM implementation that is used. As expected, with the LSTM implementation 2 the network is faster then with the LSTM implementaion 1 when using the GPU. On the PC the Keras LSTM implementation 2 is about 10% faster than the implementation 1. On the NVIDIA Jetson Nano the Keras LSTM implementation 2 is even about 20% faster than the implementation 1. Nevertheless, for all setups the processing time per frame of the CODANet is lower when using only the CPU and not the GPU. Furthermore, when using only the CPU we see a direct impact of the compression threshold of the CODANet on the mean processing time per frame. On the NVIDIA Jetson Nano the processing time per frame for the compression threshold 0.5 is almost 70% less then the processing time per frame for the compression threshold 1.0 when using only the CPU. On the PC the processing time per frame for the compression threshold 0.5 is about 50% less then the processing time per frame for the compression threshold 1.0 when using only the CPU.

Looking at the histogram of the processing time per frame of the CODANet on the PC (figure 4.5), we see that for the compression thresholds 0.9 and below, the majority of the frames have a processing time smaller than 8 *ms*. However, for all experiments there is a small percentage of frames that has a processing time larger than 8 *ms* (see table 4.10). This makes non of the implementations truly real-time capable in the current setup. However, the desktop PC on which we perform our experiments is connected

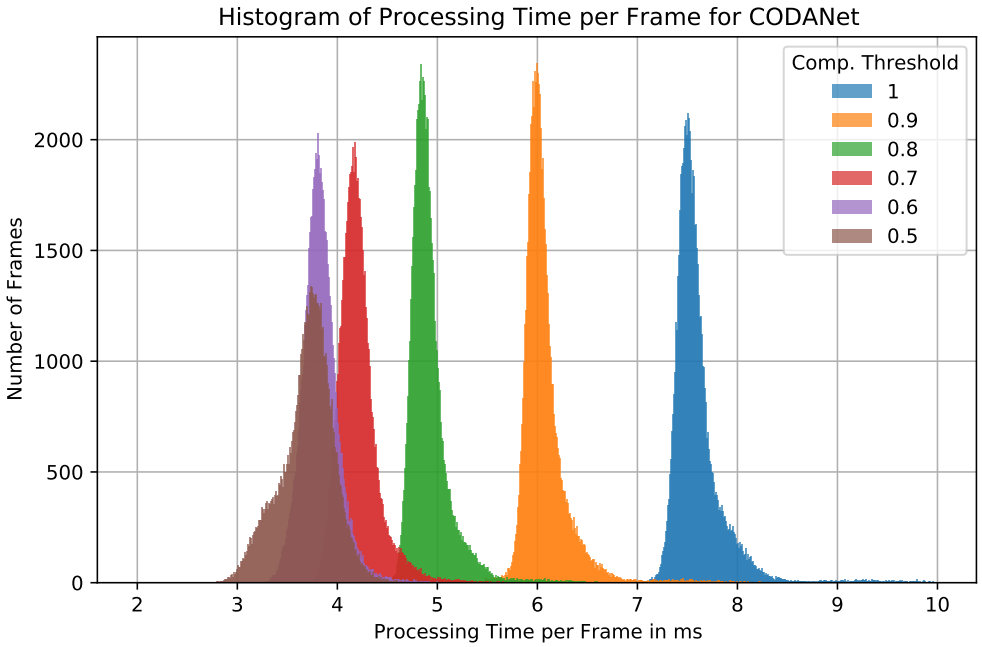


Figure 4.5.: Histogram of processing time per frame for CODANet on the PC with only CPU and Keras LSTM Implementation 1.

Compression Threshold	Percentage of frames over 8 *ms*
1.0	6.88%
0.9	0.41%
0.8	0.12%
0.7	0.08%
0.6	0.03%
0.5	0.03%

Table 4.10.: Percentage of frames of the CODANet for which the processing time is larger than the hop size of 8 *ms*.

to the internet and the university network. It runs a standard Ubuntu 18.04 and is accessed through Secure Shell (ssh). This leaves room for optimization and we think that for the compression threshold 0.7 acceptable real-time processing could be achieved on a desktop PC.

In Figure 4.6, which shows the histogram of the processing time per frame for the CODANet on the NVIDIA Jetson Nano, we see that for all compression thresholds the CODANet is not real-time capable on the NVIDIA Jetson Nano.

All the conclusions we make in this section about the processing time per frame as a function of the hardware and the Keras LSTM implementation for the CODANet, we see in the same way for the hyper-parameter tuning of the ODANet. Appendix A.5 contains all corresponding tables and histograms for the ODANet.

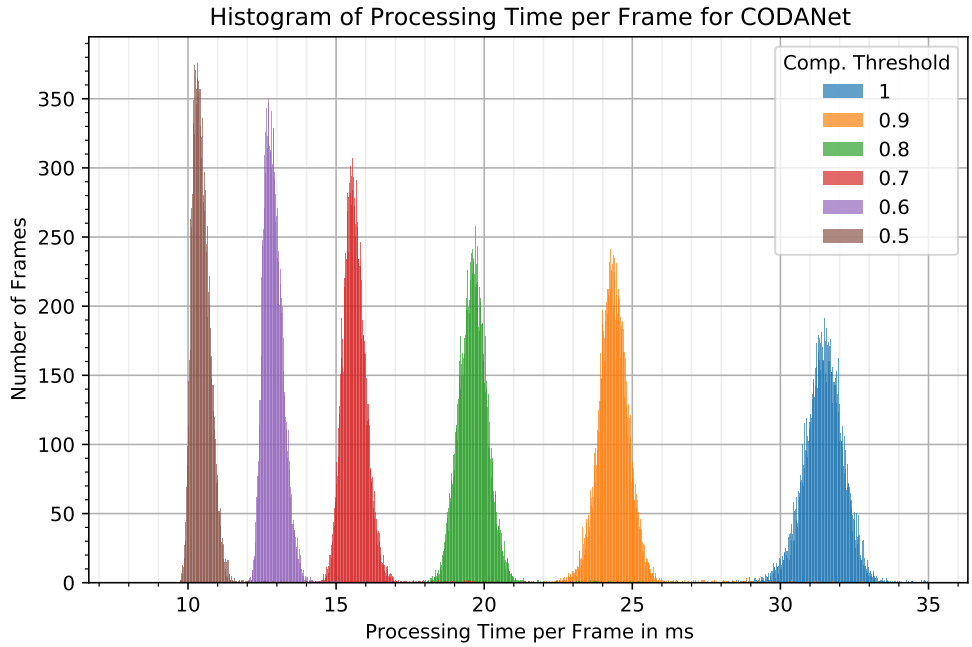


Figure 4.6.: Histogram of processing time per frame for CODANet on the NVIDIA Jetson Nano with only CPU and Keras LSTM Implementation 1.

4.3.6. Additional Findings

In this section, we present additional results that are not part of the core research of this work but may be of interest to other researchers.

Meta-Frame Size

We conducted one experiment with meta-frame size 100 and all other parameters like shown in table 4.2 on page 36. However, for this experiment the validation loss does not decrease further after epoch 8 but starts increasing as shown in figure 4.7. In general this could be a sign of over-fitting. Nevertheless, over-fitting is unusual so early in the training. One possible explanation is that the network is not able to learn good anchors and embeddings from such short spectrogram segments, which contain information of only 0.8 seconds of the audio signal. That is why we use the meta-frame size of 400 for all experiments presented in the previous sections.

Curriculum Training of CODANet

Figure 4.8 shows the SI-SNR of the CODANet over the number of epochs when it is trained with the initial weights (epoch 0) taken from the ADANet of experiment number 3.2 at a compression threshold of 0.7. The black horizontal line shows the SI-SNR without curriculum training, which is exceeded after 15 epochs of curriculum training. We assume that the SI-SNR goes down from epoch 0 to epoch 1 because the optimizer in our

case RMSprop is still learning how the weights can be updated in an optimized way. In general, this shows that curriculum training can further improve the performance of the CODANet.

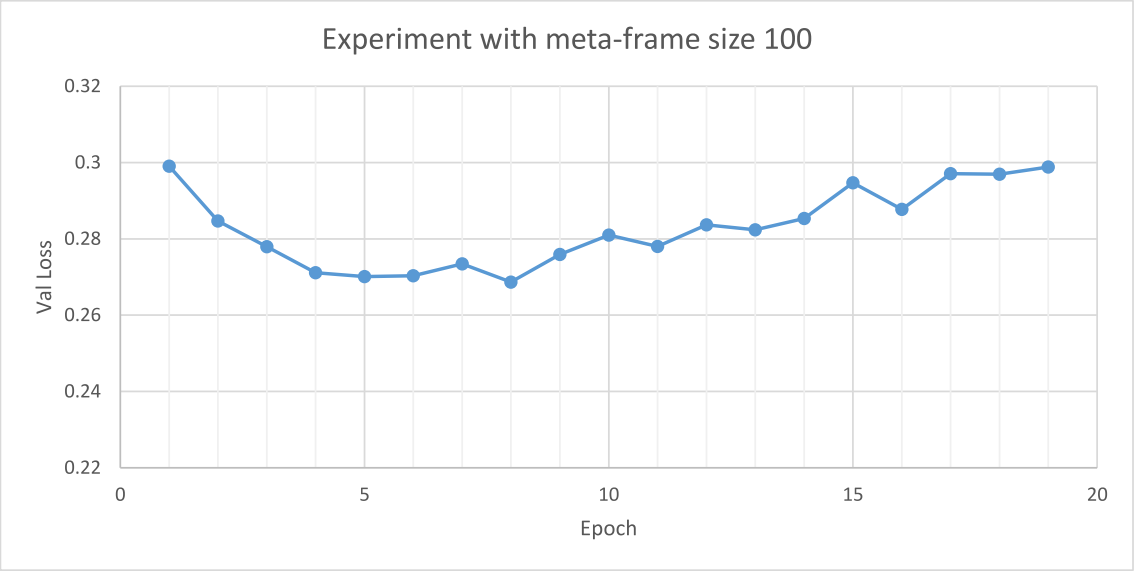


Figure 4.7.: Validation loss of ADANet experiment with meta-frame size 100

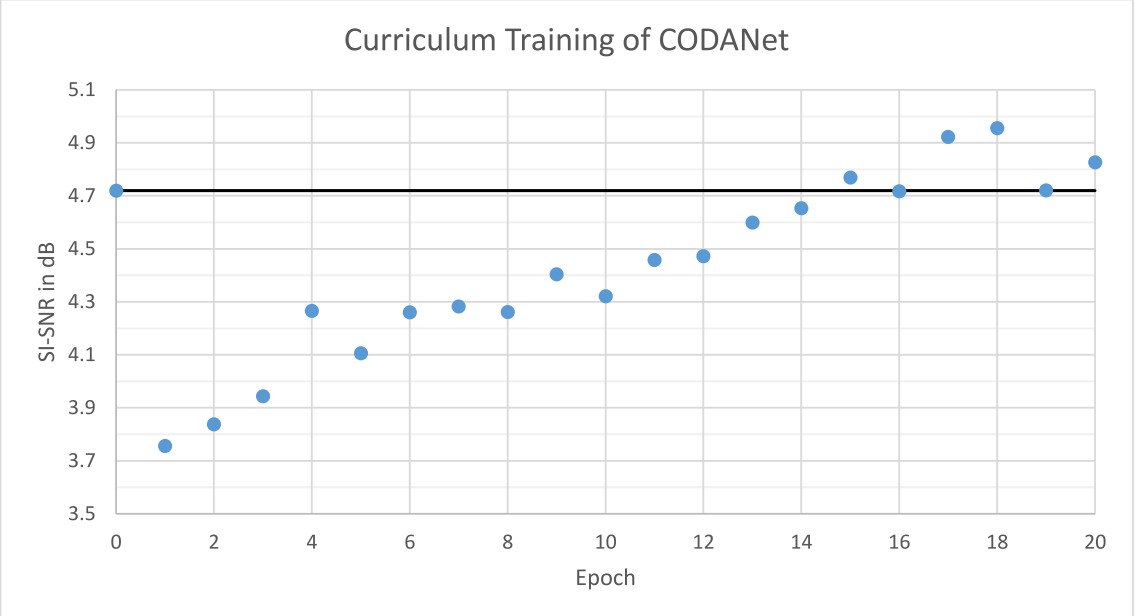


Figure 4.8.: SI-SNR on test set for curriculum training of CODANet with compression threshold 0.7. The black horizontal line indicates the SI-SNR without curriculum training.

ODANet with 3 LSTM Layers

Training the ADANet with 3 LSTM layers instead of 4 LSTM layers and using the resulting weights in the ODANet leads to good results as shown in figure 4.9. The SI-SNR for the ODANet with 3 LSTM layers is 5.2 dB at an average processing time per frame of 5.9 ms on PC with only CPU and LSTM Implementation 1. However, due to time constrains we did not further investigate on this option. In future work, it would be interesting to analyze the effect of compressing the ODANet with 3 LSTM layers and the effect of reducing the number of LSTM units on the performance and processing time per frame.

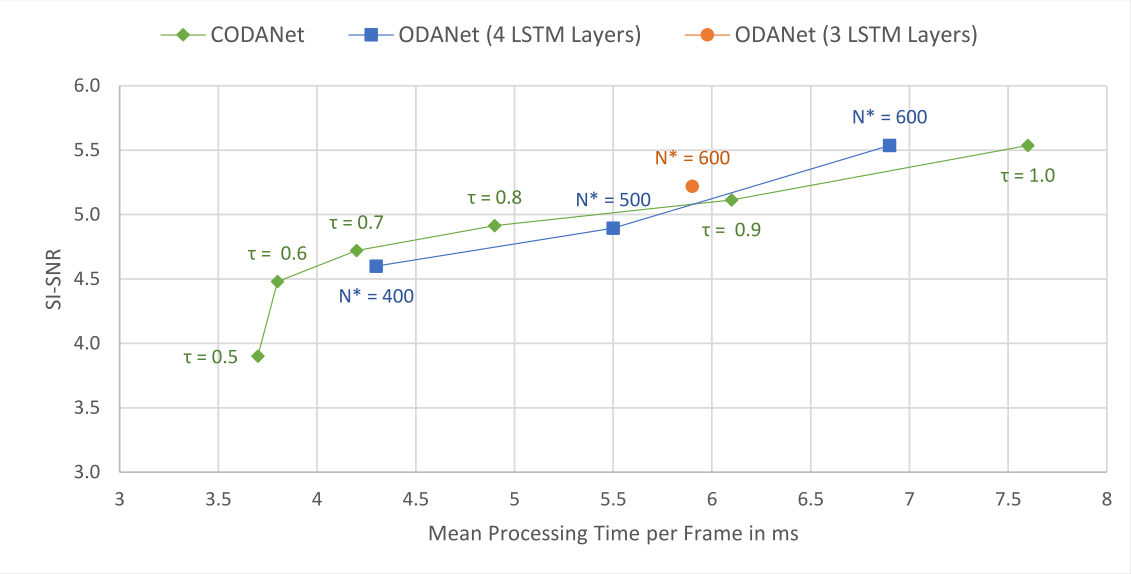


Figure 4.9.: SI-SNR over processing time per frame on the PC for CODANet (green) and ODANet with 4 anchor points and 4 LSTM layers (blue) and 3 LSTM layers (orange). We aim for a high SI-SNR with a low processing time. The labels indicate the compression threshold τ for the CODANet and the number of LSTM units N^* in each layer of the ODANet. The dashed arrow indicates that the weights from this ODANet experiment are used in all CODANet experiments.

5. Discussion

In this chapter, we summarize our results and compare them to the related work.

One key finding of our experiments is that the compression method by Prabhavalka et al. [4] is faster and better in finding a good trade-off between low processing time and speech separation performance than hyper-parameter tuning. The compression method is faster in finding a good trade-off, because once we have a trained ODANet the weights for the CLSTMs can be calculated in less than a minute using the SVD as described in section 3.1.4. That the resulting CODANet has a better speech separation performance than the ODANet for similar processing times can easily be seen in figure 4.4 on page 43.

While real-time speech separation seems possible on the desktop PC, it does not seem possible on the NVIDIA Jetson Nano, because the average processing time of the smallest tested CODANet with a compression threshold of 0.5 is above the hop size of 8 ms with an SI-SNR of 3.9 dB. We could further decrease the compression threshold, but figure 4.4 suggest that this would cause a strong decrease in the speech separation performance with only a minor decrease in the processing time.

When comparing the performance loss of 15% in the SI-SNR due to the reduction of the network size in our work to the performance loss of 1% in the SI-SNR in [7] and the performance loss of 5% in the SDR in [27], we find that the performance loss in our work is relatively large in relation to the compression achieved. In [1] and [27] the networks are even more reduced in proportion than in our work. Nevertheless, as shown in the additional findings (section 4.3.6) with curriculum training of the CODANet the gap in the performance between the ODANet and the CODANet could possibly be narrowed.

If we compare our results to related speech separation systems, we see that our ADANet, ODANet and the CODANet with reduced processing time are far below the state of the art speech separation performance as shown in table 5.1. Nevertheless, we believe that the observations we make on the relative improvement in processing time of CODANets in relation to the performance degradation can be transferred to other speech signal processing algorithms using RNNs in particular LSTMs.

The differences in the SI-SNR performance between our ADANet-BLSTM implementation and the ADANet-BLSTM implementation by Luo et al. [7] could be caused by the difference in the implementation, which we detailed in section 3.1.1 and 3.1.2. In contrast to our implementation, Luo et al. implemented a salient weight threshold, used a slightly different loss function, performed curriculum training with a meta-frame size of 100 followed by a meta-frame size of 400, and used dropout in the LSTM layers during training.

ID	Method	No. of weights in million	Causal	SI-SNR (dB)	SDR (dB)
-	DPCL++ [12]	8	×	10.3	-
-	ADANet-BLSTM [7]	33	×	9.6	-
-	ADANet-BLSTM† [7]	33	×	10.8	10.8
-	ADANet-LSTM [2]	12	×	9.1	9.5
-	Conv-TasNet-gLN [17]	5	×	15.3	15.6
-	DPTNet [19]	3	×	20.2	20.6
-	ODANet [2]	12	✓	9.0	9.4
-	uPIT-LSTM [15]	77	✓	-	7.0
-	LSTM-TasNet [16]	32	✓	10.8	11.2
-	Conv-TasNet-cLN [17]	5	✓	10.6	11.0
1.6	ADANet-BLSTM	33	×	6.8	7.9
1.6	ADANet-BLSTM*	33	×	8.4	9.1
3.2	ADANet-LSTM*	12	×	6.6	7.7
3.2	ODANet*	12	✓	5.5	6.8
4.4	CODANet*	5	✓	4.7	6.1

Table 5.1.: Comparison with other methods on WSJ0-2mix dataset. For the methods with references we report the results from the respective publications. The † indicates the results with curriculum training and dropout. The * indicates evaluation with meta-frame size 800.

Furthermore, we only trained our models for 50 epochs while Lue et al. trained their models for up to 100 epochs. Plots of the SI-SNR on the validation set over the epochs of one training suggest that we possible could continue training our networks without over-fitting (see figure 4.2 on page 38).

For better comparison between the ADANet, the ODANet and CODANet the we also evaluate our non-causal experiments with a meta-frame size of 800 instead of 400 as indicated by the * in table 5.1. Because we evaluate the ODANet and CODANet with a meta-frame size of 800 by default to ensure that an entire utterance is processed before the states in the LSTMs respectively CLSTMs are reset. Because of this change in the evaluation setup, the SI-SNR of experiment number 1.6 is about 25% better for the ADANet-BLSTM* evaluated with a meta-frame size of 800 than the SI-SNR for the ADANet-BLSTM evaluated with a meta-frame size of 400. This is because with meta-frame size 400 speaker permutation in the output can occur between the meta-frames of one utterance.

If we compare the CODANet* with a compression threshold of 0.7, which has an average processing time of 4.2 *ms* on the PC, to our ADANet and ODANet implementations and the related work, we can make the following observations. The SI-SNR of our CODANet* is 15% less than our uncompressed version the ODANet* while the average processing time per frame is decrease from 6.9 *ms* to 4.2 *ms* by 39%. If we compare the relative decrease of the SI-SNR in our implementation from the ADANet-LSTM* to the

ODANet* it is slightly larger than the relative decrease in [2]. However, Han et al. [2] trained the ODANet after loading the weights from the ADANet, which we do not. This could be an explanation for this difference especially since our additional experiments on the CODANet show that curriculum training can further improve the performance while maintaining the processing time.

Looking at the other causal methods we see that in the other LSTM-based approaches uPIT-LSTM [15] and LSTM-TasNet [16] are much larger than the ODANet [2] in terms of the number of weights while having almost similar performances. This shows that the ODANet is a good choice in terms of the baseline model size. In contrast, Conv-TasNet-cLN [17] has only as many weights as our CODANet and better performance than the time-frequency domain approaches. However, though the algorithmic latency for TASNet is as low as 5 *ms* an actual processing time per frame has not been reported to our knowledge.

Finally, if we look a the non-causal time domain approaches like Conv-TasNet-gLN [17] and DPTNet [19] they outperform the time-frequency domain approaches and use non-recurrent DNN. Low-rank matrix factorization like in [33] could also be applied to compress these approaches, but the compression method investigated in this work based on [4], which jointly optimizes the recurrent and non-recurrent kernel of the following layer is specific to RNNs like LSTMs.

In the next chapter we conclude our work.

6. Conclusion

In this work, we optimized the Online Deep Attractor Network (ODANet) [2] for real-time speech separation on an embedded system like the NVIDIA Jetson Nano by applying hyper-parameter tuning and the compression method by Prabhavalkar et al. [4]. We introduced the Compressed Online Deep Attractor Network (CODANet), which is a compressed version of the ODANet, in which the long short-term memory (LSTM) layers are replaced with compressed long short-term memory (CLSTM) layers. For these CLSTMs we are the first to our knowledge to provide a detailed explanation in combination with an open-source implementation[1]. In addition, we presented Sphere Pair: an optimized way to initialize the anchors of the Anchored Deep Attractor Network (ADANet) [7], which is the non-causal version of the ODANet and required for curriculum training of the ODANet.

With the CODANet we improved the average processing time per frame on the NVIDIA Jetson Nano, the selected embedded system, from 26.8 ms per frame for the ODANet to 15.6 ms per frame for the corresponding CODANet with a compression threshold of 0.7, while the speech separation performance decreased by only 15% from 5.5 dB to 4.7 dB in the SI-SNR on the WSJ0-2mix test set. Yet we have not achieved our goal of bringing the processing time below the hop size of the STFT of 8 ms, which is a necessary requirement for real-time application on the embedded system. On the PC in contrast the average processing time per frame is smaller than the hop size, 6.9 ms for the ODANet and 4.2 ms for the CODANet and thus allowing for real-time speech separation on such systems.

In general, our experiments showed that the CODANet has a better speech separation performance compared to a hyper-parameter tuned ODANet for similar processing times. On this basis, we conclude that the applied compression method with CLSTMs is better suited for finding a good trade-off between low latency and speech separation performance than hyper-parameter tuning. Future research could validate whether CLSTMs yield similar results when they are applied to other signal processing algorithms using LSTMs.

[1] We provide our open-source CLSTM implementation at `https://github.com/sp-uhh/compressed-lstm`

Bibliography

[1] Y. Luo, C. Han, and N. Mesgarani, "Ultra-Lightweight Speech Separation via Group Communication," *arXiv preprint arXiv:2011.08397*, 2020.

[2] C. Han, Y. Luo, and N. Mesgarani, "Online Deep Attractor Network for Real-time Single-channel Speech Separation," in *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2019, pp. 361–365, iSSN: 1520-6149.

[3] D. Yu, M. Kolbæk, Z.-H. Tan, and J. Jensen, "Permutation invariant training of deep models for speaker-independent multi-talker speech separation," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Mar. 2017, pp. 241–245, iSSN: 2379-190X.

[4] R. Prabhavalkar, O. Alsharif, A. Bruguier, and L. McGraw, "On the compression of recurrent neural networks with an application to LVCSR acoustic modeling for embedded speech recognition," in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2016, pp. 5970–5974.

[5] J. R. Hershey, Z. Chen, J. Le Roux, and S. Watanabe, "Deep clustering: Discriminative embeddings for segmentation and separation," in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Mar. 2016, pp. 31–35, iSSN: 2379-190X.

[6] Z. Chen, Y. Luo, and N. Mesgarani, "Deep attractor network for single-microphone speaker separation," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Mar. 2017, pp. 246–250, iSSN: 2379-190X.

[7] Y. Luo, Z. Chen, and N. Mesgarani, "Speaker-Independent Speech Separation With Deep Attractor Network," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 26, no. 4, pp. 787–796, Apr. 2018, conference Name: IEEE/ACM Transactions on Audio, Speech, and Language Processing.

[8] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[9] S. M. Kuo, B. H. Lee, and W. Tian, *Real-Time Digital Signal Processing: Fundamentals, Implementations and Applications*. John Wiley & Sons, Aug. 2013.

[10] M. A. Stone and B. C. Moore, "Tolerable hearing aid delays. I. Estimation of limits imposed by the auditory path alone using simulated hearing losses," *Ear and Hearing*, vol. 20, no. 3, pp. 182–192, 1999, publisher: LWW.

[11] NVIDIA Corporation, "Jetson Nano Data Sheet," Feb. 2020. [Online]. Available: https://developer.nvidia.com/embedded/dlc/jetson-nano-system-module-datasheet

[12] Y. Isik, J. L. Roux, Z. Chen, S. Watanabe, and J. R. Hershey, "Single-Channel Multi-Speaker Separation using Deep Clustering," *arXiv:1607.02173 [cs, stat]*, Jul. 2016, arXiv: 1607.02173. [Online]. Available: http://arxiv.org/abs/1607.02173

[13] Z.-Q. Wang, J. L. Roux, and J. R. Hershey, "Alternative Objective Functions for Deep Clustering," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Apr. 2018, pp. 686–690, iSSN: 2379-190X.

[14] S. Wang, G. Naithani, and T. Virtanen, "Low-latency deep clustering for speech separation," in *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2019, pp. 76–80.

[15] M. Kolbaek, D. Yu, Z.-H. Tan, J. Jensen, M. Kolbaek, D. Yu, Z.-H. Tan, and J. Jensen, "Multitalker Speech Separation With Utterance-Level Permutation Invariant Training of Deep Recurrent Neural Networks," *IEEE/ACM Trans. Audio, Speech and Lang. Proc.*, vol. 25, no. 10, pp. 1901–1913, Oct. 2017. [Online]. Available: https://doi.org/10.1109/TASLP.2017.2726762

[16] Y. Luo and N. Mesgarani, "TaSNet: Time-Domain Audio Separation Network for Real-Time, Single-Channel Speech Separation," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Apr. 2018, pp. 696–700, iSSN: 2379-190X.

[17] Y. Luo and N. Mesgarani, "Conv-TasNet: Surpassing Ideal Time–Frequency Magnitude Masking for Speech Separation," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 27, no. 8, pp. 1256–1266, Aug. 2019.

[18] Y. Luo, Z. Chen, and T. Yoshioka, "Dual-Path RNN: Efficient Long Sequence Modeling for Time-Domain Single-Channel Speech Separation," in *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. Barcelona, Spain: IEEE, May 2020, pp. 46–50. [Online]. Available: https://ieeexplore.ieee.org/document/9054266/

[19] J. Chen, Q. Mao, and D. Liu, "Dual-Path Transformer Network: Direct Context-Aware Modeling for End-to-End Monaural Speech Separation," *arXiv preprint arXiv:2007.13975*, 2020.

[20] D. Ditter and T. Gerkmann, "A Multi-Phase Gammatone Filterbank for Speech Separation via TasNet," *arXiv:1910.11615 [cs, eess]*, Oct. 2019, arXiv: 1910.11615. [Online]. Available: http://arxiv.org/abs/1910.11615

[21] D. Stoller, S. Ewert, and S. Dixon, "Wave-u-net: A multi-scale neural network for end-to-end audio source separation," *arXiv preprint arXiv:1806.03185*, 2018.

[22] Z. Shi, H. Lin, L. Liu, R. Liu, J. Han, and A. Shi, "FurcaNeXt: End-to-end monaural speech separation with dynamic gated dilated temporal convolutional networks," *arXiv preprint arXiv:1902.04891*, 2019.

[23] C. Subakan, M. Ravanelli, S. Cornell, M. Bronzi, and J. Zhong, "Attention is All You Need in Speech Separation," *arXiv preprint arXiv:2010.13154*, 2020.

[24] N. Zeghidour and D. Grangier, "Wavesplit: End-to-end speech separation by speaker clustering," *arXiv preprint arXiv:2002.08933*, 2020.

[25] T. Choudhary, V. Mishra, A. Goswami, and J. Sarangapani, "A comprehensive survey on model compression and acceleration," *Artificial Intelligence Review*, vol. 53, no. 7, pp. 5113–5155, Oct. 2020. [Online]. Available: https://doi.org/10.1007/s10462-020-09816-7

[26] F. Drakopoulos, D. Baby, and S. Verhulst, "Real-Time Audio Processing on a Raspberry Pi using Deep Neural Networks," in *23rd International Congress on Acoustics (ICA 2019)*. Deutsche Gesellschaft für Akustik, 2019, pp. 2827–2834.

[27] I. Fedorov, M. Stamenovic, C. Jensen, L.-C. Yang, A. Mandell, Y. Gan, M. Mattina, and P. N. Whatmough, "TinyLSTMs: Efficient Neural Speech Enhancement for Hearing Aids," *arXiv preprint arXiv:2005.11138*, 2020.

[28] S. Bianco, R. Cadene, L. Celona, and P. Napoletano, "Benchmark analysis of representative deep neural network architectures," *IEEE Access*, vol. 6, pp. 64 270–64 277, 2018, publisher: IEEE.

[29] A. W. Rix, J. G. Beerends, M. P. Hollier, and A. P. Hekstra, "Perceptual evaluation of speech quality (PESQ)-a new method for speech quality assessment of telephone networks and codecs," in *2001 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No. 01CH37221)*, vol. 2. IEEE, 2001, pp. 749–752.

[30] E. Vincent, J. Barker, S. Watanabe, J. Le Roux, F. Nesta, and M. Matassoni, "The second 'CHiME'speech separation and recognition challenge: Datasets, tasks and baselines," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2013, pp. 126–130.

[31] NVIDIA Corporation, "Jetson TX1 Module Data Sheet," Nov. 2016. [Online]. Available: http://developer.nvidia.com/embedded/dlc/jetson-tx1-module-data-sheet

[32] P. K. Kuhl, "Human adults and human infants show a "perceptual magnet effect" for the prototypes of speech categories, monkeys do not," *Perception & psychophysics*, vol. 50, no. 2, pp. 93–107, 1991, publisher: Springer.

[33] J. Xue, J. Li, and Y. Gong, "Restructuring of deep neural network acoustic models with singular value decomposition." in *Interspeech*, 2013, pp. 2365–2369.

[34] Keras-team, "SimpleRNNCell," Sep. 2018. [Online]. Available: https://github.com/keras-team/keras/blob/2.2.4/keras/layers/recurrent.py#L779

[35] C. Eckart and G. Young, "The approximation of one matrix by another of lower rank," *Psychometrika*, vol. 1, no. 3, pp. 211–218, 1936, publisher: Springer.

[36] L. Mirsky, "Symmetric gauge functions and unitarily invariant norms," *The quarterly journal of mathematics*, vol. 11, no. 1, pp. 50–59, 1960, publisher: Oxford University Press.

[37] C. Olah, "Understanding LSTM Networks – colah's blog," Aug. 2015. [Online]. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[38] A. Collette, *Python and HDF5: unlocking scientific data*. " O'Reilly Media, Inc.", 2013.

[39] E. Vincent, R. Gribonval, and C. Fevotte, "Performance measurement in blind audio source separation," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 14, no. 4, pp. 1462–1469, Jul. 2006.

[40] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[41] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.

[42] Keras-team, "LSTMCell," Sep. 2018. [Online]. Available: https://github.com/keras-team/keras/blob/2.2.4/keras/layers/recurrent.py#L1756

[43] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient Primitives for Deep Learning," *arXiv:1410.0759 [cs]*, Dec. 2014, arXiv: 1410.0759. [Online]. Available: http://arxiv.org/abs/1410.0759

A. Appendix

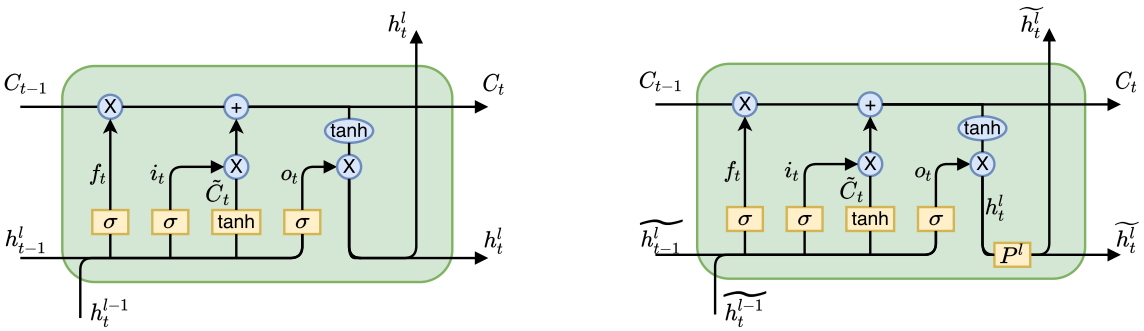
A.1. Compressed LSTM Equations

LSTM Equations

$$\mathbf{f}_t = \sigma_g(\mathbf{h}_t^{l-1}\mathbf{W}_{fx}^l + \mathbf{h}_{t-1}^l\mathbf{W}_{fh}^l + \mathbf{b}_f) \tag{A.1}$$
$$\mathbf{i}_t = \sigma_g(\mathbf{h}_t^{l-1}\mathbf{W}_{ix}^l + \mathbf{h}_{t-1}^l\mathbf{W}_{ih}^l + \mathbf{b}_i) \tag{A.2}$$
$$\mathbf{o}_t = \sigma_g(\mathbf{h}_t^{l-1}\mathbf{W}_{ox}^l + \mathbf{h}_{t-1}^l\mathbf{W}_{oh}^l + \mathbf{b}_o) \tag{A.3}$$
$$\tilde{\mathbf{C}}_t = \sigma_c(\mathbf{h}_t^{l-1}\mathbf{W}_{cx}^l + \mathbf{h}_{t-1}^l\mathbf{W}_{ch}^l + \mathbf{b}_c) \tag{A.4}$$
$$\mathbf{C}_t = \mathbf{f}_t \odot \mathbf{C}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{C}}_t \tag{A.5}$$
$$\mathbf{h}_t^l = \mathbf{o}_t \odot \sigma_h(\mathbf{C}_t) \tag{A.6}$$

Compressed LSTM Equations

$$\mathbf{f}_t = \sigma_g(\widetilde{\mathbf{h}_t^{l-1}}\mathbf{Z}_{fx}^l + \widetilde{\mathbf{h}_{t-1}^l}\mathbf{Z}_{fh}^l + \mathbf{b}_f) \tag{A.7}$$
$$\mathbf{i}_t = \sigma_g(\widetilde{\mathbf{h}_t^{l-1}}\mathbf{Z}_{ix}^l + \widetilde{\mathbf{h}_{t-1}^l}\mathbf{Z}_{ih}^l + \mathbf{b}_i) \tag{A.8}$$
$$\mathbf{o}_t = \sigma_g(\widetilde{\mathbf{h}_t^{l-1}}\mathbf{Z}_{ox}^l + \widetilde{\mathbf{h}_{t-1}^l}\mathbf{Z}_{oh}^l + \mathbf{b}_o) \tag{A.9}$$
$$\tilde{\mathbf{C}}_t = \sigma_c(\widetilde{\mathbf{h}_t^{l-1}}\mathbf{Z}_{cx}^l + \widetilde{\mathbf{h}_{t-1}^l}\mathbf{Z}_{ch}^l + \mathbf{b}_c) \tag{A.10}$$
$$\mathbf{C}_t = \mathbf{f}_t \odot \mathbf{C}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{C}}_t \tag{A.11}$$
$$\mathbf{h}_t^l = \mathbf{o}_t \odot \sigma_h(\mathbf{C}_t) \tag{A.12}$$
$$\widetilde{\mathbf{h}_t^l} = \mathbf{h}_t^l\mathbf{P}^l \tag{A.13}$$



(a) LSTM Cell [37]

(b) Compressed LSTM Cell

Figure A.1.: The LSTM Cell (a) and its compressed version (b)

A.2. ADANet Architecture Figure Adapted for the First Frame of the ODANet

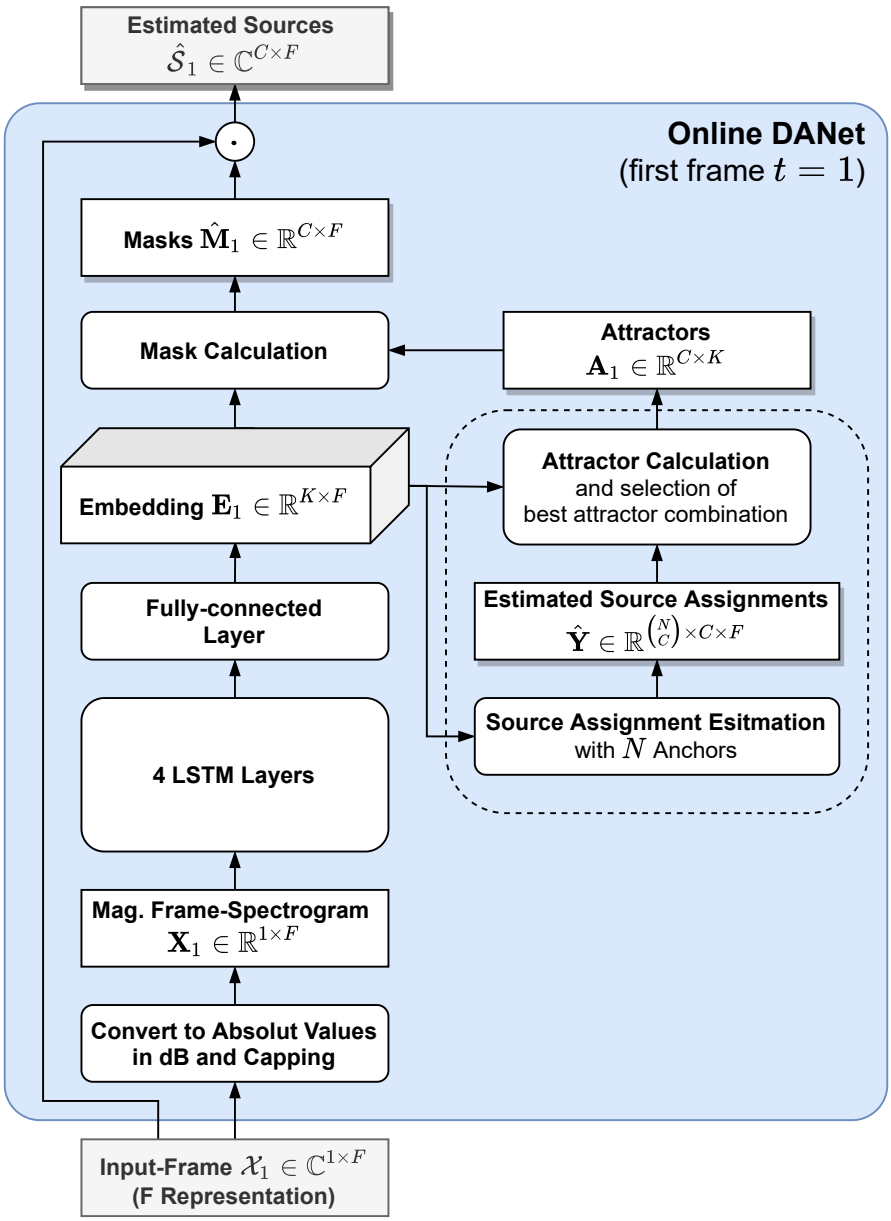


Figure A.2.: ODANet architecture (first frame $t = 1$). The dashed box with rounded edges represents a super-process which combines the entire attractor calculation for the first-frame and the attractor calculation for the for the following frames which is shown in figure 3.5 on page 20. Graphic adapted from [7, p. 789].

A.3. Fixed Parameters for Hyper-parameter Tuning

Parameter	Value
Meta-frame size	400
LSTM directonality	unidirectional
Batch size	32
Loss function	equation 3.6
Optimizer	RMSprop
Learning rate	0.0001
Early stopping (on val. loss)	not used
Reduce learining rate on plateau	not used

Table A.1.: Fixed parameters for hyper-parameter tuning experiments.

A.4. SI-SNR over Processing Time per Frame Including ODANet with 6 Anchors

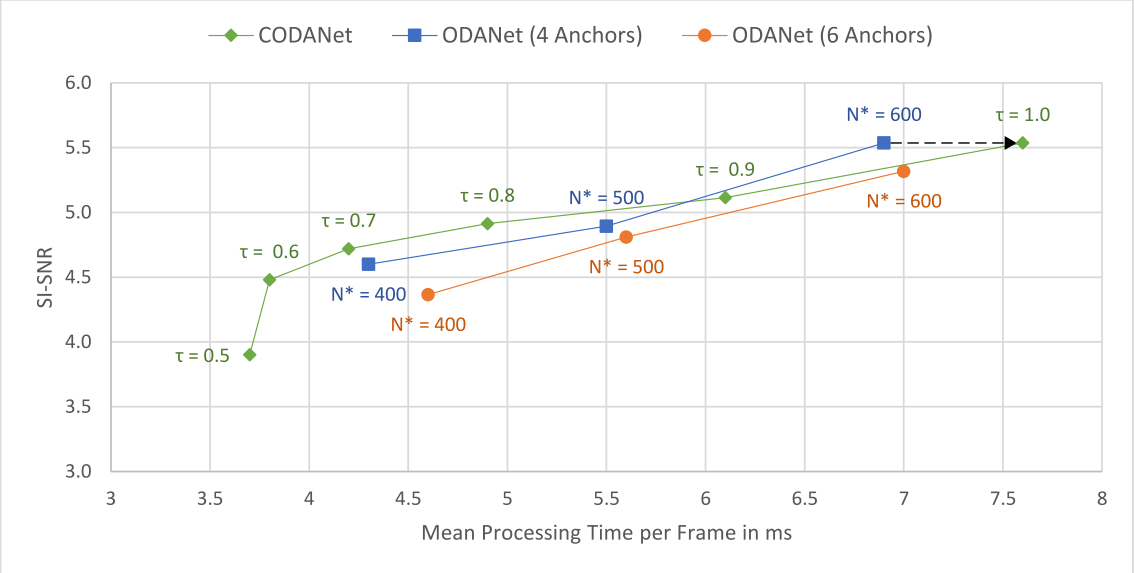


Figure A.3.: SI-SNR over processing time per frame on the PC for ODANet with 4 anchor points (blue) and ODANet with 6 anchor points (orange) and CODANet (green). We aim for a high SI-SNR with a low processing time. The labels indicate the compression threshold τ for the CODANet and the number of LSTM units N^* in each layer of the ODANet. The dashed arrow indicates that the weights from this ODANet experiment are used in all CODANet experiments.

A.5. Processing Time per Frame of ODANet

On the PC using only CPU

No. of Units in LSTMs	Embedding Dimension	No. of Anchor Points	mean in *ms*	var	max in *ms*
400	20	6	4.6	0.15	79
500	20	4	5.5	0.15	79
		6	5.6	0.16	78
600	15	4	6.9	0.18	79
		6	6.9	0.20	79
	20	4	6.9	0.23	90
		6	7.0	0.21	80

Table A.2.: Processing time per frame of ODANet on **PC** using only the **CPU** with the Keras LSTM implementation **1**

No. of Units in LSTMs	Embedding Dimension	No. of Anchor Points	mean in *ms*	var	max in *ms*
400	20	6	5.5	0.32	84
500	20	4	6.1	0.40	87
		6	6.4	0.41	81
600	15	4	6.9	0.60	88
		6	7.2	0.66	88
	20	4	7.1	0.60	89
		6	7.3	0.57	82

Table A.3.: Processing time per frame of ODANet on **PC** using only the **CPU** with the Keras LSTM implementation **2**

On the PC using GPU

No. of Units in LSTMs	Embedding Dimension	No. of Anchor Points	mean in *ms*	var	max in *ms*
400	20	6	8.7	0.86	104
500	20	4	8.6	0.80	92
		6	8.6	0.83	97
600	15	4	8.5	0.80	89
		6	8.7	0.82	90
	20	4	8.7	0.82	93
		6	8.6	0.84	97

Table A.4.: Processing time per frame of ODANet on **PC** using the **GPU** with the Keras LSTM implementation **1**

No. of Units in LSTMs	Embedding Dimension	No. of Anchor Points	mean in *ms*	var	max in *ms*
400	20	6	7.8	0.73	90
500	20	4	7.8	0.72	91
		6	7.8	0.77	106
600	15	4	7.7	0.73	96
		6	7.8	0.73	92
	20	4	7.8	0.71	95
		6	7.9	0.75	94

Table A.5.: Processing time per frame of ODANet on **PC** using the **GPU** with the Keras LSTM implementation **2**

On the NVIDIA Jetson Nano using only CPU

No. of Units in LSTMs	Embedding Dimension	No. of Anchor Points	mean in *ms*	var	max in *ms*
400	20	6	17.8	0.25	38
500	20	4	21.2	0.26	41
		6	21.3	0.28	32
600	15	4	26.6	0.41	49
		6	27.1	0.39	44
	20	4	26.8	0.42	43
		6	27.7	0.42	41

Table A.6.: Processing time per frame of ODANet on NVIDIA **Jetson Nano** using only the **CPU** with the Keras LSTM implementation **1**

No. of Units in LSTMs	Embedding Dimension	No. of Anchor Points	mean in *ms*	var	max in *ms*
400	20	6	17.5	0.25	35
500	20	4	21.1	0.66	44
		6	21.7	0.69	38
600	15	4	27.7	0.23	47
		6	28.4	0.18	38
	20	4	28.5	0.29	44
		6	29.1	0.23	47

Table A.7.: Processing time per frame of ODANet on NVIDIA **Jetson Nano** using only the **CPU** with the Keras LSTM implementation **2**

On the NVIDIA Jetson Nano using GPU

No. of Units in LSTMs	Embedding Dimension	No. of Anchor Points	mean in *ms*	var	max in *ms*
400	20	6	34.7	2.45	53
500	20	4	37.5	1.54	55
		6	38.0	1.83	55
600	15	4	38.6	1.65	57
		6	38.7	2.16	57
	20	4	38.7	2.10	71
		6	38.2	2.42	63

Table A.8.: Processing time per frame of ODANet on NVIDIA **Jetson Nano** using the **GPU** with the Keras LSTM implementation **1**

No. of Units in LSTMs	Embedding Dimension	No. of Anchor Points	mean in *ms*	var	max in *ms*
400	20	6	28.9	3.09	52
500	20	4	28.2	3.29	47
		6	29.7	3.33	50
600	15	4	28.1	2.37	56
		6	29.7	2.58	53
	20	4	28.3	2.63	63
		6	30.1	2.61	53

Table A.9.: Processing time per frame of ODANet on NVIDIA **Jetson Nano** using the **GPU** with the Keras LSTM implementation **2**

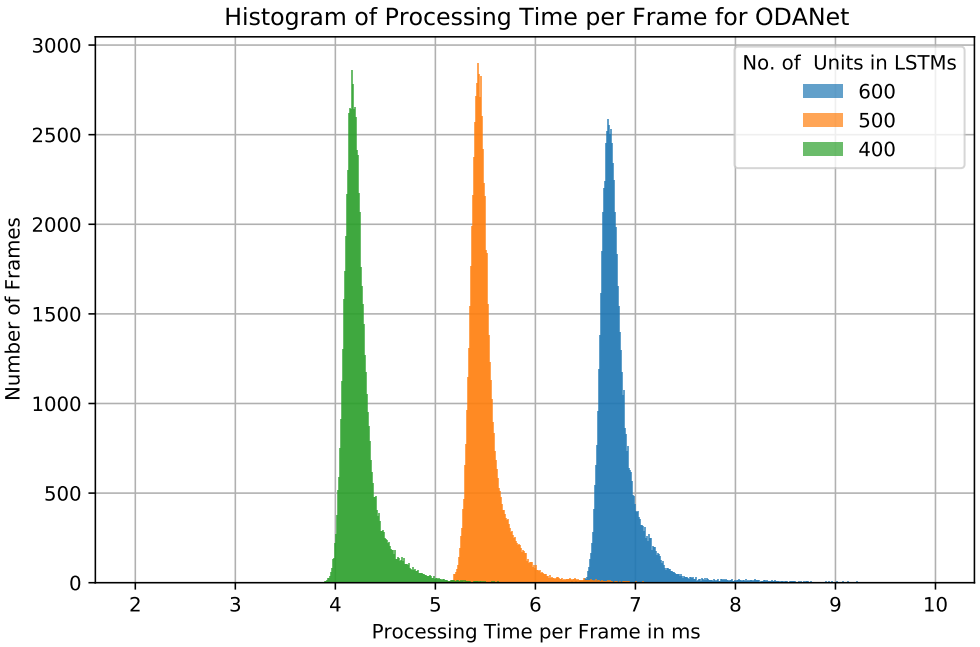


Figure A.4.: Histogram of processing time per frame for ODANet with 4 anchor points and embedding dimension 20 on the PC with only CPU and Keras LSTM Implementation 1.

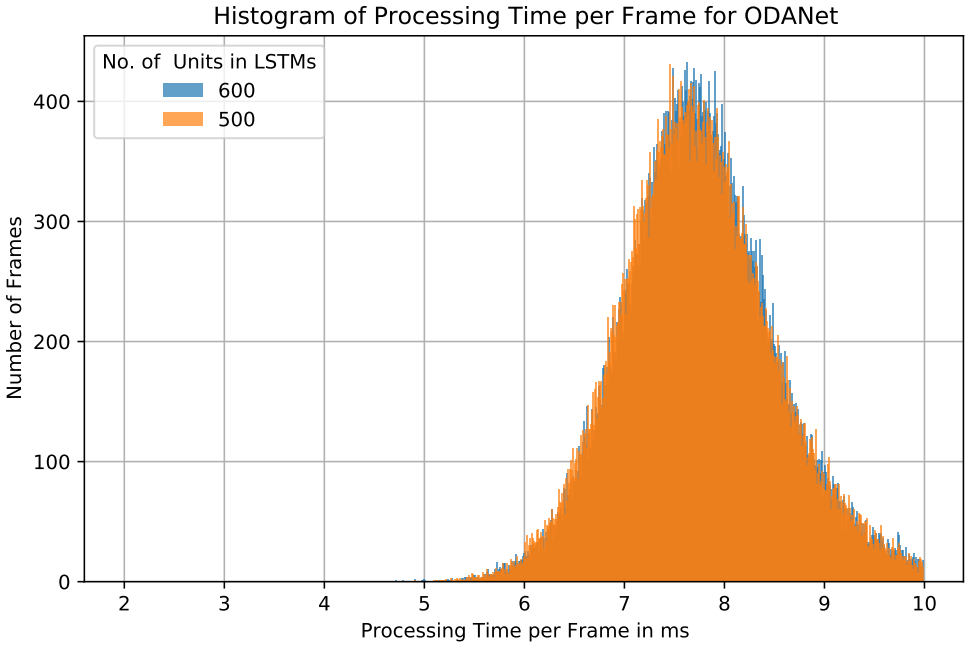


Figure A.5.: Histogram of processing time per frame for ODANet with 4 anchor points and embedding dimension 20 on the PC with GPU and Keras LSTM Implementation 2.

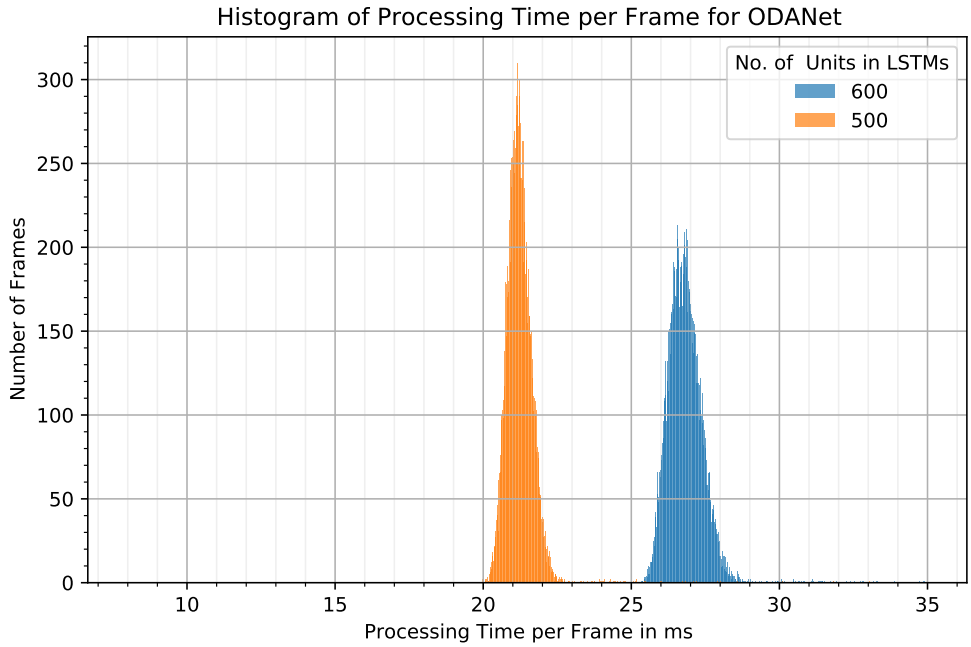


Figure A.6.: Histogram of processing time per frame for ODANet with 4 anchor points and embedding dimension 20 on the NVIDIA Jetson Nano with only CPU and Keras LSTM Implementation 1.

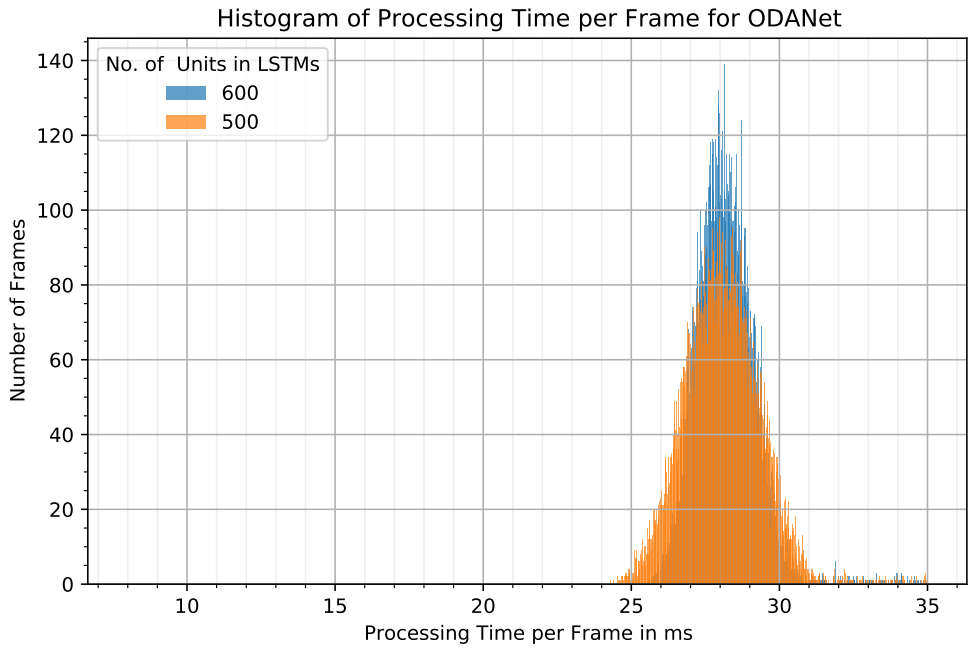


Figure A.7.: Histogram of processing time per frame for ODANet with 4 anchor points and embedding dimension 20 on the NVIDIA Jetson Nano with GPU and Keras LSTM Implementation 2.

A.6. Processing Time per Frame of CODANet

Hardware	Processing Unit	LSTM Impl.	Compression Threshold 0.5	0.6	0.7	0.8	0.9	1.0
Jetson	CPU	1	10.5	12.9	15.6	19.6	24.4	31.5
		2	10.2	12.8	15.6	19.8	25.1	32.4
	GPU	1	32.9	34.5	37.3	35.8	36.3	36.4
		2	28.0	27.4	27.5	27.7	28.3	28.9
PC	CPU	1	3.7	3.8	4.2	4.9	6.1	7.6
		2	3.8	4.5	5.1	5.9	6.9	8.1
	GPU	1	8.6	8.6	8.5	8.6	8.5	8.7
		2	7.7	7.7	7.7	7.6	7.8	7.7

Table A.10.: **Mean** processing time per frame in *ms* of CODANet

Hardware	Processing Unit	LSTM Impl.	Compression Threshold 0.5	0.6	0.7	0.8	0.9	1.0
Jetson	CPU	1	0.2	0.1	0.2	0.3	0.5	0.6
		2	0.2	0.4	0.2	0.3	0.4	0.5
	GPU	1	3.3	3.7	2.8	3.7	2.9	2.3
		2	2.1	2.3	1.9	2.5	2.1	2.8
PC	CPU	1	0.2	0.1	0.2	0.2	0.2	0.3
		2	0.3	0.3	0.3	0.4	0.5	0.7
	GPU	1	0.8	0.8	0.8	0.8	0.8	0.8
		2	0.7	0.7	0.7	0.7	0.7	0.7

Table A.11.: **Variance** of processing time per frame of CODANet

Hardware	Processing Unit	LSTM Impl.	Compression Threshold 0.5	0.6	0.7	0.8	0.9	1.0
Jetson	CPU	1	25	26	29	33	42	56
		2	28	30	34	36	42	55
	GPU	1	61	72	55	55	55	77
		2	54	56	46	46	59	54
PC	CPU	1	78	79	78	81	79	101
		2	90	90	82	88	88	90
	GPU	1	94	92	92	96	88	91
		2	92	87	94	105	93	94

Table A.12.: **Maximum** processing time per frame in *ms* of CODANet

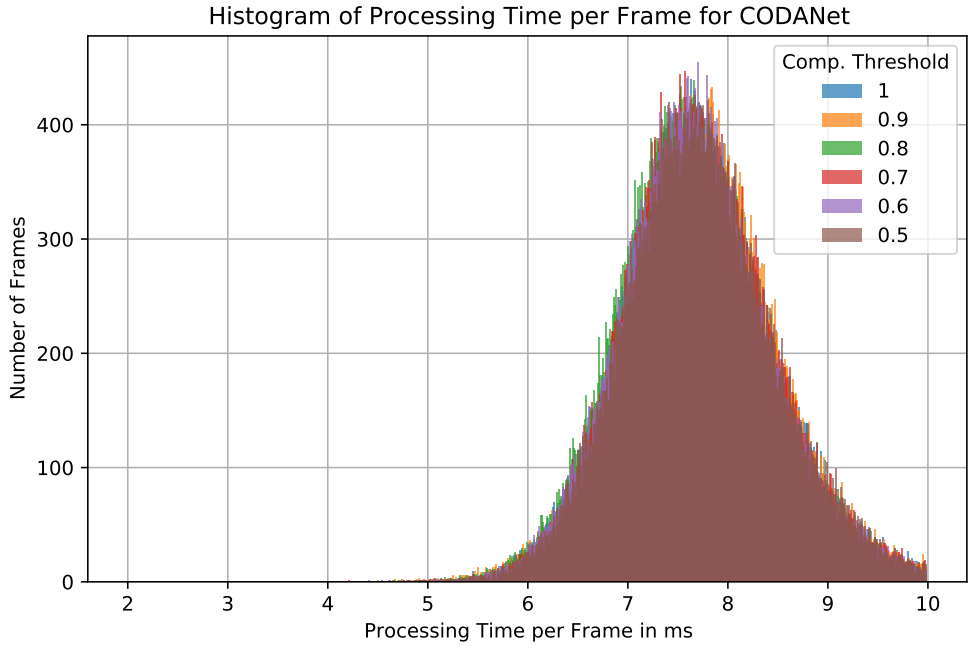


Figure A.8.: Histogram of processing time per frame for CODANet on the PC with GPU and Keras LSTM Implementation 2.

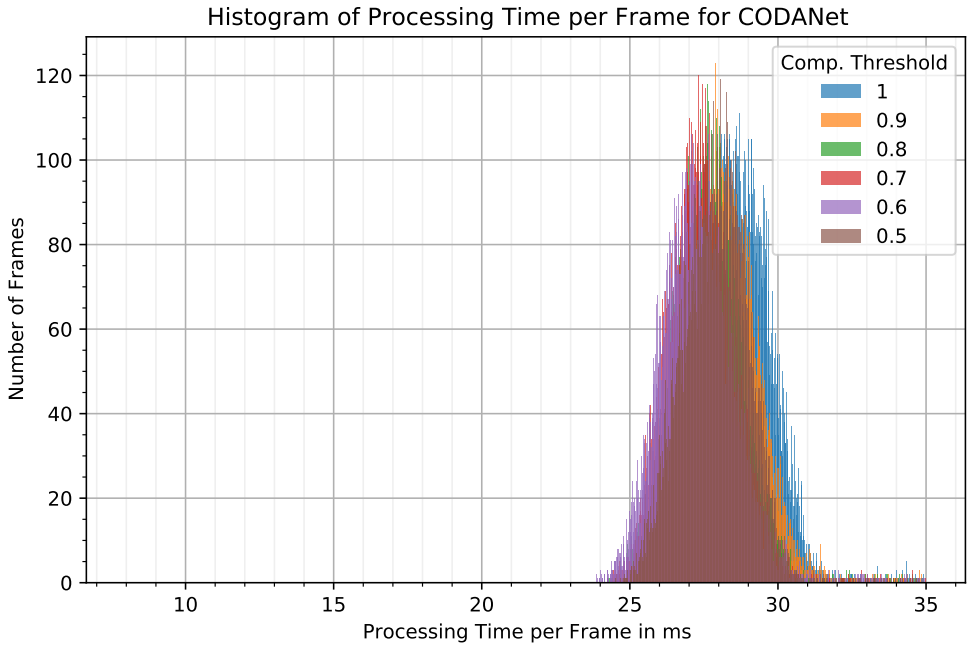


Figure A.9.: Histogram of processing time per frame for CODANet on the NVIDIA Jetson Nano with GPU and Keras LSTM Implementation 2.

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Masterstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel — insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen — benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Hamburg, den 27.11.2020 Marc Siemering

Veröffentlichung

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

Hamburg, den 27.11.2020 Marc Siemering