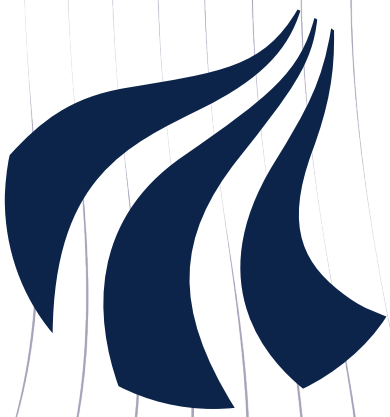
Real-time implementation considerations of a deep learning based Voice Activity Detection



AALBORG UNIVERSITY

STUDENT REPORT

Master thesis (50 ECTS)

Group 976

Signal Processing and Acoustics

with Specialisation in Signal Processing and computing

Aalborg University - June 2022

AALBORG UNIVERSITY
STUDENT REPORT

Department of Electronic Systems
Fredrik Bajers Vej 7B
9220 Aalborg Øst
https://www.es.aau.dk/

Title:

Real-time implementation considerations of a deep learning based Voice Activity Detection

Project:
Master thesis (50 ECTS)

Project period:

September 2021 - June 2022

Project group:

Group 976

Participants:

Claus Meyer Larsen

AAU supervisors:
Zheng-Hua Tan
Peter Koch

Industry partners:
Sebastian Schiøler (RTX)

No. of pages: 101
Appendix: 99 - 101
Date of completion: June 2022

Abstract:

In this thesis carried out in collaboration with RTX we consider a deep learning based method for Voice Activity Detection. In this work we investigate the potential of this method to be used in a real-time application on an embedded device.
Towards achieving this we work with three research questions that are aiming to increase the performance of the Voice Activity Detection, lower the algorithmic delay and finally consider methods making it more suitable for implementation on a resource constrained device.
As part of this work is submitted a paper to *Interspeech 2022* which proposes a method for increasing the Voice Activity Detection performance and reducing the algorithmic delay. The performance is increased by introducing adversarial multi-task learning during training and the algorithmic delay is lowered by reducing the filter sizes of the network. Reducing the algorithmic delay leads to a small performance degradation.
Afterwards is considered pruning and quantization in the use-case of this project. Finally it is discussed on which hardware architectures this algorithm is best suited for an implementation based on the aforementioned optimisations.

Claus Meyer Larsen

Claus Meyer Larsen

Preface

This thesis is carried out by group 976 at the masters programme of Signal Processing and Acoustics, with specialization in Signal Processing and Computing, at Aalborg University from September 2021 to June 2022.

Reading instructions

This thesis is divided into 4 main parts. Within these parts are a number of chapters numbered in the order which they appear. Sections, subsections etc. are also numbered following this convention.

Figures, equations and tables are numbered as X.Y where X is the chapter in which they appear, and Y is the order of which they appear within this chapter. Thus, the third table of chapter 5 will be numbered 5.3. Unless explicitly noted by a citation, the figures in this report is made by the author

Appendices are in the back of the report.

Sources are cited using the IEEE citation style. *IEEE style is a numbered referencing style that uses citation numbers in the text of the paper, provided in square brackets. A full corresponding reference is listed at the end of the paper, next to the respective citation number.*

References to the paper submitted to Interspeech 2022 is highlighted using bold text. Example: section **2.1**

Throughout the report, commas are used as thousand separators while dots are used as decimal separators.

Acknowledgements

The author would like to thank Associate Professor Peter Koch and Professor Zheng-Hua Tan for their supervision of this project. Additionally the author would like to thank RTX for their support and inputs during the course of this project.

Contents

Acronyms

Adam
. 15

ALU
Arithmetic Logic Unit. 84

ASIC
Application-Specific Integrated Circuit. 83, 86, 91, 92, 94

AUC
Area Under the Curve. 21, 24, 27, 28, 29, 30, 47, 50, 54, 67, 68, 69, 94, 95

Aurora-2
. 5, 22, 24, 38, 40, 42, 47, 48, 49, 52, 90, 93, 94

BCE LF
Binary Cross Entropy Loss Function. 11

BN Batch Normalisation. 18

CE LF
Cross Entropy Loss Function. 10, 11

CNN
Convolutional Neural Network. 7, 8, 9, 11, 12, 13, 26, 38, 57, 59

CRS
Compressed Row Storage. 70

DB Decoder Block. 6, 39, 40, 46, 51, 52, 53, 54, 66, 73, 74, 95

DNN
Deep Neural Network. 3

DSP
Digital Signal Processor. 83, 84, 85, 86, 87, 88, 91, 94

EB Encoder Block. 6, 39, 46, 47, 51, 52, 67, 69, 72, 73, 74, 77, 79

FB Framing Block. 6, 39, 40, 46, 51, 52, 73, 74, 75

FCN
Fully Convolutional Network. 5, 26, 45, 58, 61, 93

FIR Finite Impulse Response. 86

FN False Negative. 21, 29

FP False Positive. 21, 29

FPGA
Field Programmable Gate Array. 83, 86, 87, 88, 91, 92, 94

FPR False Positive Rate. 21, 28

GAN
Generative adversarial network. 45

GPU
Graphics processing unit. 25

KKT
Karush Kuhn Tucker. 65

Leaky ReLU
Leaky Rectified Linear Unit. 9, 10, 16, 18, 23, 27, 74

LSB Least Significant Bit. 63

M10K
Memory 10K. 87

MAC
Multiply ACcumulate. 58, 83, 84, 85, 91

MLAB
Memory Logic Array Block. 87

MSB
Most Significant Bit. 85

RAM
Random Access Memory. 58

ReLU
Rectified Linear Unit. 9, 18

RMSprop
Root Mean Squared propagation. 14, 15, 23

ROC
Receiver Operating Characteristic. 20, 21, 28

SGD
Stochastic Gradient Descent. 13, 14, 15

SIMD
Single Instruction/Multiple Data. 83

SNR
Signal to Noise Ratio. 2, 22, 28, 29, 30, 47, 48, 49, 50, 52, 53, 90, 94

SQNR
Signal to Quantization Noise Ratio. 64, 65, 73, 77, 78, 79, 81, 82, 83, 84, 85, 87, 88

SSN
Speech Shaped Noise. 42

TIMIT
. 42, 43, 46, 47, 48, 90, 94

TN True Negative. 21, 29

TP True Positive. 21, 29

TPR True Positive Rate. 21, 28

VAD
Voice Activity Detection. 2, 3, 4, 5, 10, 11, 20, 21, 22, 23, 24, 27, 28, 29, 38, 39, 40, 41, 44, 45, 46, 47, 48, 51, 53, 56, 57, 68, 73, 75, 76, 77, 78, 79, 80, 81, 82, 90, 91, 92, 93, 94, 95

VLIW
Very Long Instruction Word. 83

WVAD
Waveform-based VAD. 16

Part I

Introduction

1 Introduction

Voice Activity Detection

In recent years we have seen a boom in the use of communication devices targeted towards transmitting audio in a wide range of applications and industries. A few examples are devices commonly used for everyday-activities such as smartphones and headsets. Manufacturers in this industry are continuously working towards increasing the capabilities of these devices in terms of metrics as performance and power consumption. Meanwhile the consumers demand these devices to be smaller, thus putting heavy constraints on the physical area available for embedded systems. Increasing performance while decreasing power consumption and the physical area calls for more and more advanced and optimised signal processing algorithms, such as speaker verification, speech recognition and noise cancelling.

These are generally computationally expensive algorithms, thus it is not desired to execute them at all times. Instead a common pre-processing step is Voice Activity Detection (VAD), which is the process of classifying the presence of speech in a audio. The different applications in which VAD can be used obviously entails different sets of requirements for the performance in terms of accuracy, computational complexity, latency etc. In academia a VAD algorithm is usually classifying speech in frames of 10 ms each.

Speech

In general, a noisy speech signal can be modelled using the additive noise signal model, such that:

$$x(t) = s(t) + v(t) \tag{1.1}$$

where $x(n)$ and $s(n)$ represents the noisy and clean speech signal respectively, sampled at time n, while $v(n)$ is the additive noise sampled at time n. This additive noise is what makes the VAD a non-trivial task, as the algorithm is supposed to generalise well under different noise types and Signal to Noise Ratio (SNR) levels. The task of VAD can be carried out by several different algorithms. These different algorithms can broadly by categorised as either a supervised or a non-supervised method.

Unsupervised methods

Unsupervised algorithms is the traditional approach towards the VAD. These algorithms typically calculate some speech characteristics in a pre-processing step, and then exploit these characteristics to classify the frames as either speech or non-speech. An example of this is

the rVAD algorithm presented in [1] which exploits that the energy is typically higher in segments where speech is present, as compared to non-speech segments.

Supervised methods

A supervised method on the other hand lies in the area of machine learning. Machine learning is a very active area of research which has gained increased popularity in recent years. This sudden boom in popularity arises from the increase in computational power available. A rule of thumb is *Moore's law* which states that the number of transistors in integrated circuits are doubled every 18-24 months. This increased computational power has allowed for training of more complex networks leading to better performance. VAD based on machine learning is also an active area of research, which has already proven to outperform more conventional methods for the task of VAD [2] [3]

These algorithms include training on some labeled data, from which the algorithm learns to find the optimal parameters for the classification task at hand. Supervised methods are often based on Deep Neural Network (DNN) [2][3][4][5]. In recent years it has shown beneficial to perform the VAD directly on the time domain waveform, and let the network learn the optimal parameters itself [2] [3].

Collaboration

This thesis is made in collaboration with RTX. RTX specialises mainly in short-range wireless communication and audio signal processing. They operate in a wide range of applications such as healthcare systems, intercom systems, gaming equipment and stage-microphones and equipment. Many of these are battery powered devices which are limited in both computational power, power consumption and physical area. Additionally, most of these devices are subject to hard latency constraints ranging from microseconds to a few milliseconds.

It is of RTX' interest to conduct a feasibility study on a VAD algorithm based on machine learning with focus on increasing the performance without introducing any additional computational cost or latency to the execution-time - as well as observing the performance when the latency is lowered. Additionally, methods for optimising the model with regards to a potential implementation in an embedded system will be considered. In this work the total latency will be considered a results of the algorithmic delay(i.e. how much future knowledge do we need to generate and output) and the computational overhead(i.e. how long time does it take to compute the output once all samples are ready.)

1.1 Scope of this thesis

This introduction of the work leads to the following three research questions:

1. *Can we potentially increase the noise-robustness of a VAD without increasing its computational cost and latency to the execution-time?*
2. *How will it affect the performance of the VAD if we allow it to use less future samples to generate a VAD output and thus decrease the algorithmic delay?*

3. *How can we reduce the computational cost of the VAD in a real time application while maintaining performance?*

These three research questions will act as the foundation for this work. In the first part of the study will be presented an algorithm which will be used as the starting point of this thesis. Then some necessary theory to get a deep understanding of the algorithm will be presented. Following this presentation some early design choices will be made and the results from the original paper will be attempted recreated. This will lead into a discussion on what data sets will be used for training and testing, as well as how these data sets are used to resemble a real world application in the best way possible.

After this early part of the work the focus will switch to the three research questions. First will be introduced a way of potentially increasing the VAD performance without introducing any cost to the execution-time. Secondly the algorithmic delay of the algorithm will be analysed and some experiments carried out trying to reduce this delay. Lastly, the work will focus on aspects related to a real-time implementation. First will be introduced some methods for reducing requirements to the computational power, the power consumption and the physical area needed for an implementation on an embedded system, where after these methods will be further discussed and some experiments carried out in the use-case of this work. Lastly, this discussion will be put in the context of a potential implementation on some different hardware architectures.

1.2 Framework

As this work has a limited timeframe and is mainly focused on improving VAD performance and finding its potential for a real-time application, an existing VAD algorithm will be used as a framework.

In recent years supervised VAD algorithms taking the raw waveform as input has been an active area of research [2][3]. Using the raw waveform as input allows the VAD to find the optimal features on its own, leading to that the VAD performance is not limited by the quality of the features found from a pre-processing step. Another benefit is that the VAD can make predictions based both on the phase and the magnitude of the signal.

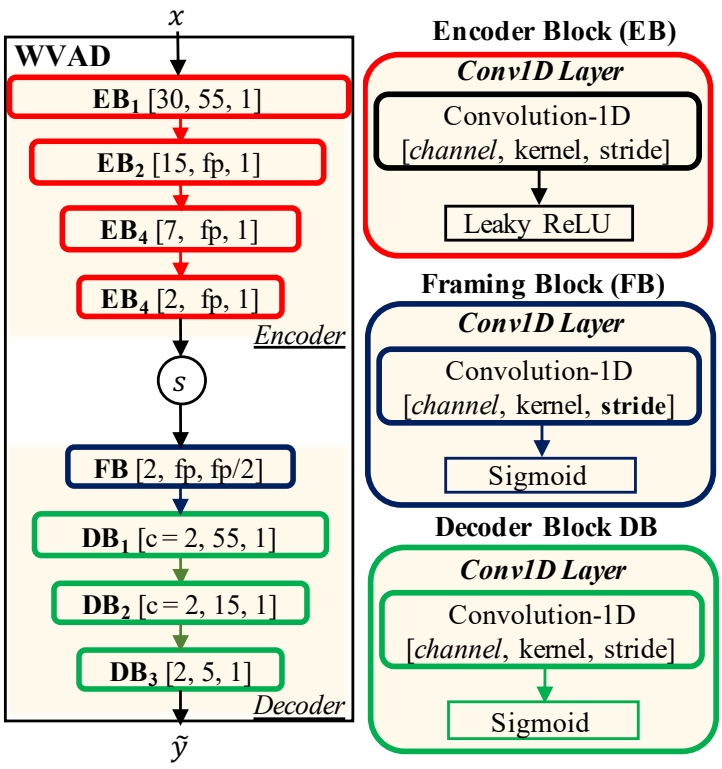
One VAD algorithm that takes as input the waveform is the *Waveform based VAD* presented in [2]. This algorithm has shown state-of-the-art performance on the (Aurora-2) [6] database and will be considered the framework for this work.

The method resolves to a Fully Convolutional Network (FCN) and consist of 3 separate blocks which serves easily distinguishable purposes. The network being FCN means that the only operations involved in the network is convolutional layers and activation functions. Thus there are no pooling layers etc.

1.2.1 Algorithm architecture

The framework from [2] is presented in fig. 1.1.

In fig. 1.1a the structure of the network is shown, while fig. 1.1b outlines the notation used in



(a) WVAD framework **(b)** Detailed view of each block

Figure 1.1: The VAD method presented in [2]

fig. 1.1a as well as the activation function used.

In the following section the purpose of the three blocks of the algorithm will be described. This description will only be on a conceptual level. In the next chapter the relevant theory will be presented and a further analysis of the algorithm will be carried out.

Encoder Block

The Encoder Block (EB) consist of four convolutional layers with filters operating on features sampled at the same rate as the audio input. In the first layer the raw waveform is convolved with 30 different filters to generate 30 new feature maps of same size as the input - one for each filter. These 30 feature maps are then through the next three layers gradually downscaled into 2 feature maps, which will be the input for the next part of the algorithm.

Framing Block

Following the EB is the Framing Block (FB). This block is responsible for downsampling the feature maps along the time-axis, such that each channel now has one feature per 10 ms frame.

Decoder Block

The last block of the algorithm is the Decoder Block (DB). This block takes as input the frame-wise features generated by the FB. Through 3 convolutional layers the feature maps are refined into scores for speech and non-speech by taking into account the past and future features. Thus the DB outputs a vector $[y_{\text{speech}}\ y_{\text{non-speech}}]$ with one feature for each 10 ms frame. The VAD decision is then chosen as the channel with the highest score.

Presentation of theory 2

From the short introduction of the algorithm in section 1.2, further knowledge about the theory behind the algorithm is necessary in order to get a deeper understanding. The relevant theory will be presented in this chapter. Afterwards, this theory will be the foundation of a series of design choices when implementing the algorithm presented in section 1.2.

2.1 Convolutional Neural Networks

The chapter will mainly focus on neural networks, in particular Convolutional Neural Network (CNN)s. The theory will be presented in two steps: the forward step and the backward step. The forward step is where the predictions are made by the CNN and the backward step is where the parameters of the CNN are updated such that even better predictions can be made in the next forward step.

2.2 Forward step

In general, a CNN consist of three types of layers. An input layer, an output layer and one or more hidden layers in-between. These layers are called hidden because their inputs and outputs are masked by activation functions and convolutions in the input and output layers, and thus acts as a black-box system.

Each of these layers in a CNN consist of a linear convolutional layer followed by an activation functions which applies an unlinearity to the layer. The next two sections will explain the concepts of a convolutional layer and activation functions. Given that an audio signal is a 1-dimensional time series, only 1-dimensional convolutions will be considered in the following.

2.2.1 Convolutions

This section on convolutions is based on [7, Chapter 9]. In the first part of a convolutional layer the convolutions take place. A discrete time convolution is usually denoted as:

$$y[n] = x[n] * w[n] \tag{2.1}$$

In CNN terminology x is referred to as the input, w is referred to as the *kernel* or *filter*, while y is the output, also referred to as the *feature map*. It is worth noting that convolving a time domain signal $x(t)$ with a filter $w(t)$ is equivalent to multiplying the frequency representation of a signal $X(f)$ with the frequency response of the filter $W(f)$. The convolution operation is defined as:

$$y[n] = x[n] * w[n] = \sum_{k=-\infty}^{\infty} x[k] \cdot w[n-k] \tag{2.2}$$

Which can in turn be computed efficiently using matrix multiplications:

$$y = Wx \tag{2.3}$$

Additionally, in a CNN it is often beneficial to include a bias b in each layer. Further generalising the matrix multiplication to span several layers, each layer l of a CNN can be computed as:

$$x^{[l]} = W^{[l]}x^{[l-1]} + b^{[l]} \tag{2.4}$$

And as mentioned in section 2.2 the output of a convolutional layer is masked by an activation function. Thus the output of a convolutional layer can finally be expressed as:

$$x^{[l]} = \mathrm{AF}(f^{[l]}x^{[l-1]} + b^{[l]}) \tag{2.5}$$

Where l denotes the layer, w is the weights of the layer, x is the output of the previous layer and b is the bias of the layer.

In terms of the convolutions in CNN, three concepts are particularly important to understand the purpose of, and what happens if they are changed. These are:

- Filters and their size
- Stride
- Zero padding

Filter

In a CNN a filter w is the sequence of filter coefficients with which the input x is convolved to produce the feature map y. The kernel size, i.e. how many filter coefficients it contains, is crucial when it comes to designing CNN. In the case of a 1D convolution, a larger kernel size would mean that the feature generated from the convolution will be less local and rely more on its neighbours, while smaller kernels will extract more fine grained information. Because of this it is often desirable to use small kernels, and instead more layers such that more and more complex features can be extracted from the fine grained features through the layers.

When convolving an input with a kernel, the size of the resulting feature map will decrease. That is because the kernel can only fit a limited number of times when sliding over the input. The feature map will have the size $N - M + 1$, where N is the length of the input and M is the length of the kernel. This also emphasizes the benefit of a smaller kernel size. Additionally, because of this property, it is desired to use an odd-sized kernel to ensure symmetry in the convolutions

Stride

Another important aspect in CNNs is the *stride* of a convolutional layer. The stride is the number of samples with which the kernel is moved in between each convolution. In the discussion of section 2.2.1 a stride of 1 was assumed. This means that by changing the stride, you change the number of times the kernel can fit inside the input, and thus the size of the feature map. Using a stride different from one can be an efficient way to downscale the size of your feature map.

Zero padding

Through section 2.2.1 it was discussed which effect the kernel size and stride has on the size of the feature map. One way to ensure that the size of the feature map remains the same size as the input is to use zero-padding. When you zero-pad in a CNN you pad zeroes to both sides of the input. Recall that assuming a stride of 1, the size of the feature map will be $N - M + 1$. Thus in order to achieve the same size through the input and the feature map, you will need to pad $\frac{M}{2} - 1$ zeroes to both sides of the input.

2.2.2 Activation Functions

This section on activation functions is based on the overview provided in [8]. As seen in eq. (2.2), convolutions are purely linear operations. The idea behind a CNN is that we stack layers of convolutional layers, however since these convolutions are linear, it does not make sense in itself. To apply an unlinearity to the layers, the convolutions are followed by *activation functions*, which themselves are unlinear.

In the figure of the algorithm depicted in section 1.2, it is seen that the activation functions used is Leaky ReLU and Sigmoid. For that reason, this section will be limited to a discussion of these. However, to understand the Leaky Rectified Linear Unit (Leaky ReLU), one must know its more simple variant, the Rectified Linear Unit (ReLU).

ReLU

A very popular activation function for use in CNNs is the ReLU. The ReLU is defined as:

$$f(x) = \max(0,x) \tag{2.6}$$

Thus it is simply setting negative parts of its input to 0 while keeping the positive parts. This has the advantage that it is very simple and easy to compute. It is also invariant to the magnitude of its input and therefore is *scale invariant*, such that:

$$\max(0, a \cdot x) = a \cdot \max(0, x) \tag{2.7}$$

Leaky ReLU

The Leaky ReLU is very similar to the ordinary ReLU function. The difference is that instead of setting negative inputs 0, instead it will apply some small positive gradient to them, such that:

$$f(x) = \max(\alpha \cdot x,x) \tag{2.8}$$

α is usually a small number around the magnitude of 1e$-$2 to 1e$-$4.

The purpose of Leaky ReLU is to avoid the *dying ReLU problem* leading to vanishing gradients, which will be explained further in the section on backpropagation.

Sigmoid

The last activation function to be used in the algorithm shown in section 1.2 is the *Sigmoid* function. The Sigmoid functions is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.9}$$

A plot of this function is seen below:

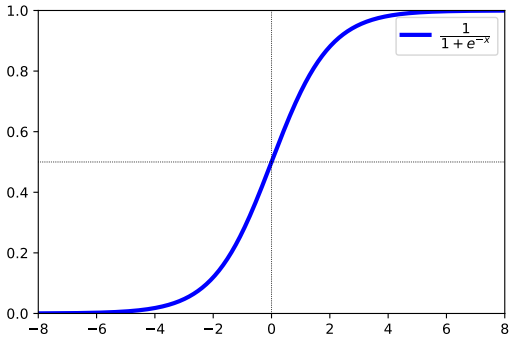


Figure 2.1: The sigmoid function

It is seen that the output is squeezed in between 0 and 1, and while the input is defined for every real value, in practice larger values will have little to no effect on the output. Therefore in order to utilise the Sigmoid activation function it is crucial to make sure the inputs are properly scaled, otherwise the *vanishing gradient* problem will occur. This will be further explained in the section on backpropagation

2.2.3 Loss functions

This section on loss functions is based on [7, Chapter 7]. The two channels of the output layer of the WVAD algorithm will each output vectors of values between 0 and 1, given that the activation function of the outout layer is the Sigmoid. These output values are unnormalised and therefore referred to as *logits*.

The last part of the forward step is to evaluate these logits. This evaluation can be done in several ways, i.e. prediction accuracy or *loss*. The loss is of particular interest as this will be a fundamental part of the backward step where the parameters of the network are updated. VAD is a classification task, and therefore a loss function of interest is the Cross Entropy Loss Function (CE LF).

2.2.3.1 Cross Entropy Loss Function

While training a network the goal is to find the parameters that minimise the loss. In the case of the CE LF the loss is calculated as the entropy between the predicted logits and the true labels. First will be considered the multi class scenario, and this theory will be used to find the Binary Cross Entropy Loss Function (BCE LF) that is of interest in the VAD classification task where only two classes is to be considered; *speech* and *non-speech*.

The CE LF is defined:

$$J_{CE} = -\sum_i t_i \log(p_i) \tag{2.10}$$

Where:

t_i = The true label for the i^{th} class
p_i = The probability for the i^{th} class
J_{CE} = The cross entropy loss

The purpose of taking the log of the logits is to punish predictions that are far off the true label, while the negative sign is to ensure a positive loss given that the probabilities are all in the range of 0 to 1. In the CE LF defined in eq. (2.10) the predicted probability for each class is needed. To get the probabilities the *softmax* function is applied to the logits. The softmax function generates probabilities for each class, which in total sums up to 1.

An example on the use of the CE LF is shown below in fig. 2.2. In this example an arbitrary CNN outputs one logit for each of its 4 classes. These logits are then converted to probabilities by the softmax function, whereafter the cross entropy is found between the predicted probabilities P and the true labels T.

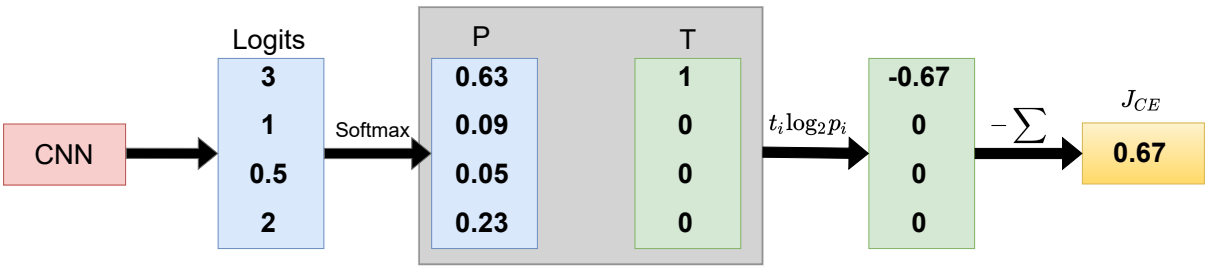


Figure 2.2: An example illustrating the use of the CE LF applied to an arbitrary CNN

2.2.3.2 Binary cross entropy loss function

In the case of WVAD, only two classes is part of the classification; *speech* and *non-speech*. Because of this a simplified version of the CE LF can be used, namely the BCE LF.

The BCE LF is defined as:

$$J_{BCE} = -\sum_{i=1}^{2} t_i \log(p_i) \tag{2.11}$$

$$\tag{2.12}$$

By exploiting the fact that $p_1 + p_2 = 1$ and only one label is 1 while the other is 0, either one of the probabilities with its corresponding label can be used to calculate the loss as:

$$J_{BCE} = -\left[t\log(p) + (1-t)\log(1-p)\right] \tag{2.13}$$

In the case of WVAD each channel outputs a vector of scores each representing a 10 ms segment of the input. To find the loss over the entire input signal, the loss is calculated for each 10 ms segment whereafter the mean of the losses is found:

$$J_{BCE} = = -\frac{1}{N}\left[\sum_{j=1}^{N}\left[t_j\log(p_j) + (1-t_j)\log(1-p_j)\right]\right] \tag{2.14}$$

2.3 Backward step

As briefly mentioned in section 2.1, the second part involved in training a CNN is the backward step. The purpose of the forward step was to calculate the loss between the true labels and the output y generated from the input x. The purpose of the backward step is to update the parameters of the CNN, such that the loss will be smaller in the next forward step. By computing the forward step and the backward step in turns, the loss will decrease until at least a local minimum is found and thereby training the network.

The backward step consists of two main components. First, the gradients are to be calculated based on the loss, such that the parameters can afterwards be updated based on the gradients.

2.3.1 Backpropagation

This section on backpropagation is based on [7, Chapter 6]. The first part of the backward step is to compute the gradients of each of the parameters with respect to the loss function. However, a CNN usually consists of a large amount of parameters making the direct gradient computation numerically infeasible.

To ease the computations in the gradient computation the *backpropagation* algorithm is used. This algorithm works by applying the chain rule to the *computational graph* used in the forward pass. This computational graph stores the order of operations from the forward pass. In the case of a CNN, it will propagate backwards through the layers of the network.

The chain rule is defined as in eq. (2.15) and works by computing the derivative of a function by composing other functions whose derivatives are already known.

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y}\frac{\partial y}{\partial x} \tag{2.15}$$

In the case of a backpropagation the function f is the loss of the network, while the derivative of this loss with respect to y is already known from the previous layer.

The chain rule can be applied in an iterative manner to the parameters of the CNN in the order determined by the computational graph. This way the gradients of a layer can be calculated once the gradient of the previous layer is known, and thus the complexity of the gradient computations can be reduced.

2.3.2 Optimizers and updating parameters

Once the gradients have been calculated using backpropagation, then next part of the backward step is to update the parameters. In doing this an optimisation algorithm capable of optimising multi-variable functions is needed. In [9] is provided an overview of and comparison between different state-of-the-art optimisers, and this will be the foundation of this short overview. In the literature is multiple studies investigating the performance of different optimisers with different results, and this field of research is so extensive that only a short excerpt is included in this work as a short introdution.

2.3.2.1 SGD

Let us first consider a popular optimiser that is used both as an optimiser of its own and as the foundation of more advanced optimisers; that is the Stochastic Gradient Descent (SGD) optimiser.

The SGD is an extension of the gradient descent algorithm which calculates the gradients based on the entire data set, which in case of a large data set will take a long time to compute and thus take long time to converge towards an optimum.

The update rule for gradient descent is defined as:

$$\theta = \theta - \alpha \cdot \nabla_\theta L(\theta; x) \tag{2.16}$$

where

θ = the parameters of the network
L = the loss computed over the full dataset x

In order to overcome this challenge we instead consider the SGD. The SGD solves the optimisation by instead of calculating gradients and updating weights based on the entire dataset, only a single sample from the data set is considered. This sample is randomly chosen, thus the algorithm is named SGD. This however leads to the problem of higher variance in the model parameters, leading to that the computed loss will fluctuate between each sample.

The update rule for stochastic gradient descent is defined as:

$$\theta = \theta - \alpha \cdot \nabla_\theta L(\theta; x^{[i]})) \tag{2.17}$$

where $x^{[i]}$ denotes a single piece of data from dataset x.

Mini-batch gradient descent

One way to get both the lower variance from the gradient descent algorithm and the faster convergence from the SGD algorithm is to use small batches from the data set for each weight update. This method is referred to as mini-batch gradient descent. In each optimisation step using mini-batch gradient descent N samples are chosen from the data set at random. The loss is calculated for all of these, where after this combined loss is backpropagated and parameters are updated.

The update rule for mini-batch gradient descent is defined as:

$$\theta = \theta - \alpha \cdot \nabla_\theta L(\theta; x^{[i:i+n]}) \tag{2.18}$$

where $x^{[i:i+n]}$ is a subset of data from dataset x.

2.3.2.2 RMSprop

Another gradient based method is Root Mean Squared propagation (RMSprop)[10]. It builds on top of the SGD (or mini-batch gradient descent) and is designed specifically to deal with the problem of vanishing and exploding gradients that will be introduced in section 2.3.3. It deals with the problem of vanishing and exploding gradients by using a moving average of squared gradients to normalise said gradient. This results in a balanced step size such that larger gradients are decreased and smaller gradients are increased.

In algorithm 1 the update rule of the RMSprop optimiser is seen. The variables are named as follows:

θ = the parameters
ϵ = very small non-zero value to avoid division by zero
ρ = the moving average decay factor
γ = weight factor for previous update step
η = the learning rate
L = the loss
m = the parameter update step
v = the moving average of the gradient

Algorithm 1: RMSprop update rule

Input: $\theta,\ \epsilon,\ \rho, \gamma,\ \eta$
Output: θ
Set $v_0 = 0,\ m_0 = 0$
for n $= 0$ **to** no. iterations **do**
  $v_{t+1} = \rho v_t + (1-\rho)\nabla L\left(\theta_t\right)^2$
  $m_{t+1} = \gamma m_t + \frac{\eta_t}{\sqrt{v_{t+1}+\epsilon}}\nabla L\left(\theta_t\right)$
  $\theta_{t+1} = \theta_t - m_{t+1}$
end

The learning rate is not as was the case of SGD, where the learning rate was multiplied directly by the gradient. Instead it is normalised by the gradient. Additionally the update

step is added to the previous update step. This means that multiple consecutive gradients of the same direction will lead to larger update steps, and it will not necessarily instantly change direction in case the gradients sign is flipped.

2.3.2.3 ADAM

The last optimiser that will be considered is the (Adam) optimiser [11]. It is an adaptive learning rate method that is computing different learning rates for different parameters. This is achieved by using estimates of the first and second moments of the moving average of the gradients similar to that of the RMSprop.

In algorithm 2 the update rule for the Adam optimiser is shown. The variables are as follows:

θ = the parameters
ϵ = very small non-zero value to avoid division by zero
η = the learning rate
β = the moving average decay factors
L = the loss
m = the moving average of the gradient of the first moment
v = the moving average of the gradient of the first moment

Algorithm 2: ADAM update rule

Input: $\theta,\ \epsilon,\ \eta,\ \beta_1,\ \beta_2$
Output: θ
Set $v_0 = 0,\ m_0 = 0,\ b_0 = 0$
for n $= 0$ **to** no. iterations **do**
  $m_{t+1} = \beta_1 m_t + (1 - \beta_1) \nabla L(\theta_t)$
  $v_{t+1} = \beta_2 v_t + (1 - \beta_2) \nabla L(\theta_t)^2$
  $b_{t+1} = \frac{\sqrt{1-\beta_2^{t+1}}}{1-\beta_1^{t+1}}$
  $\theta_{t+1} = \theta_t - \eta t \frac{m_{t+1}}{\sqrt{v_{t+1}+\epsilon}} b_{t+1}$
end

Due to the normalisation using the first and second moments of the gradient step size of the Adam optimiser is in fact invariant to the magnitude of the gradient. This can prove very helpful when exploring saddle points or ravines where SGD will struggle due to the gradients being close to 0.

Last few words on optimisers

An interesting conclusion of [9] is that the more sophisticated optimisers can always perform at least as well as the optimiser on which it is built on top. Thus as both optimisers are inspired by SGD it is possible to achieve at least the same performance. This however is very dependent on optimal tuning of hyperparameters. And the more hyperparameters is present in an optimiser the more difficult it is to initialise properly. Thus there is not "one optimiser to rule them all", but what performs the best is very application specific.

2.3.3 Exploding and vanishing gradient problems

The last topic to be discussed in the section on backpropagation is the *vanishing and exploding gradient problems* [7, Chaper 8]. First, let us consider the chain rule used in backpropagation, as seen in eq. (2.15). This states that the gradients of a layer is computed as a multiplication using the gradient of the previous layer. This means that the gradient of a network of n hidden layers will be multiplied together n times.

Following this the gradient will increase in magnitude through the layers in the case of large gradients and the gradient is said to be *exploding*, and in the case of small gradients it will decrease through the layers, and the gradient is said to be *vanishing*.

In a network with vanishing gradients the network is unable to learn, or at least learns very slowly due to the small parameter updates that follow from the small gradients. Additionally, it will lead to that the layers closer to the output will be learning more than the layers close to the input, because of how the gradients will shrink through the layers.

In a network with exploding gradients the parameter updates acts adversarily to those of the vanishing gradients; the parameter updates will be large, especially in the layers close to the input. This will lead to instability making the loss of the network fluctuate at each update. This will lead to the parameters overshooting in the optimisation step and it is not guaranteed to ever reach a local minimum.

2.3.3.1 Parameter initialisation

One way to avoid the exploding and vanishing gradient is to initialise the parameters in ways such that the output of each layer will remain similar in magnitude and variance. Considering that the layers with the sigmoid activation function outputs between 0 and 1, it is desirable to have layers with different activation functions also output values in this range. Additionally, it is standard practice to have your inputs scaled such that they fall within an normal distribution of mean 0 and standard deviation 1 [12].

In the Waveform-based VAD (WVAD) algorithm sigmoid and Leaky ReLU activation functions are used. For that reason this section will focus only on parameter initialisation for layers with these activation functions.

Parameter initialisation for sigmoid layers

Currently a go-to approach for initialising parameters in a layer with the sigmoid activation function is to use the *Xavier initialisation*.[13] As mentioned above, it is desired that the variance does not change through the layers, such that:

$$\mathrm{Var}(x^{[l]}) = \mathrm{Var}(x^{[l-1]}) \tag{2.19}$$

In the Xavier initialisation the variance of the weights is chosen such that the following constraints are satisfied:

$$n^{[l-1]} \operatorname{Var}(W^{[l-1]}) = 1 \tag{2.20}$$
$$n^{[l]} \operatorname{Var}(W^{[l]}) = 1 \tag{2.21}$$

Where n is the number of nodes in the layer l. In the general case is referred to the number of nodes in a feed-forward neural network, but this can be rewritten to instead consider convolutional layers as:

$$n^{[l-1]} = w_{size} \cdot c^{[l-1]} \tag{2.22}$$
$$n^{[l]} = w_{size} \cdot c^{[l]} \tag{2.23}$$

- n the number of nodes
- w the kernel
- c the number of channels

Relating this to the problem of exploding and vanishing gradients, three cases can be considered:

$$n^{[l]} \operatorname{Var}\left(W^{[l]}\right) \begin{cases} < 1 & \Longrightarrow \text{Vanishing features} \\ = 1 & \Longrightarrow \operatorname{Var}\left(x^{[l]}\right) = \operatorname{Var}(x^{[l-1]}) \\ > 1 & \Longrightarrow \text{Exploding features} \end{cases} \tag{2.24}$$

Thus in order to satisfy eq. (2.20), the variance of the distribution of the weights have to be initialised as:

$$\operatorname{Var}(W^{[l]}) = \frac{1}{n^{[l]}} \tag{2.25}$$
$$\operatorname{Var}(W^{[l]}) = \frac{1}{n^{[l-1]}} \tag{2.26}$$

The two constraints can be combined to:

$$\frac{2}{n^{[l-1]} + n^{[l]}} \tag{2.27}$$

Having found the variance, the initial weights can now be drawn from the normal distribution:

$$W^{[l]} \sim \mathcal{N}\left(0, \frac{2}{n^{[l-1]} + n^{[l]}}\right) \tag{2.28}$$

2.3.3.2 Parameter initialisation for Leaky ReLU layers

In [14] a new way of initialising weights was proposed. This is known as the *He initialisation* and is the go-to way of initialising weights in a convolutional layer with a ReLU activation function [12]. It is derived based on some of the same thoughts as Xavier initialisation, however it has proven to work very well when initialising ReLU layers. The Xavier initialisation assumes linearity in the activation function, which is not the case for the ReLU. Additionally ReLU is not differentiable, given that it is a combination of two separate linear functions.

Similarly to Xavier initialisation, the key idea is that the variance of the feature maps remains the same through the layers. In He initialisation this is ensured by:

$$\frac{n^{[l]}}{2}\operatorname{Var}\left(W^{[l]}\right) = 1 \Longrightarrow \operatorname{Var}(W^{[l]}) = \frac{2}{n^{[l]}} \tag{2.29}$$

Thus the initial weights of the Leaky ReLU layers will be drawn from the normal distribution;

$$W \sim \mathcal{N}\left(0, \frac{2}{n^{[l]}}\right) \tag{2.30}$$

2.3.3.3 Normalisation

In this section an alternative approach to avoid exploding and vanishing gradients is presented. This method is Batch Normalisation (BN). The section is based on [15]. BN is a method aiming towards making training of neural networks faster and more stable. In short, the first and second statistical moments (mean and variance) of the current batch are used for normalising the feature maps. Given the feature map x, the mean and variance can be found as:

$$\mu = \frac{1}{n}\sum_i x(i) \tag{2.31}$$

$$\sigma^2 = \frac{1}{n}\sum_i (x(i) - \mu)^2 \tag{2.32}$$

The mean and variance are then used for normalising the feature map through eq. (2.33). This way it is ensured that the feature maps follows a normal distribution

$$x(i)_{\text{normalised}} = \frac{x(i) - \mu}{\sqrt{\sigma^2 - \epsilon}} \tag{2.33}$$

The magnitude and standard deviation of this normal distribution can then be modified using the parameters γ and β. These adjust the standard deviation and applies a bias respectively.

$$\hat{x} = \gamma \cdot x(i)_{\text{normalised}} + \beta \tag{2.34}$$

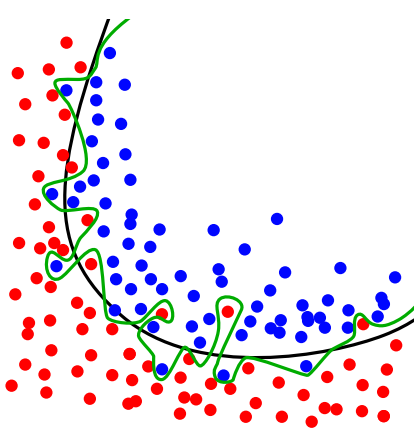
The BN can be considered a layer of its own which is introduced into the network. During training the parameters γ and β is trained alongside the other parameters of the network. Thus the BN layer can be used to keep the feature maps of the same variance and magnitude throughout the network and thereby avoiding exploding/vanishing gradients resulting in faster network training.

2.4 Overfitting and underfitting

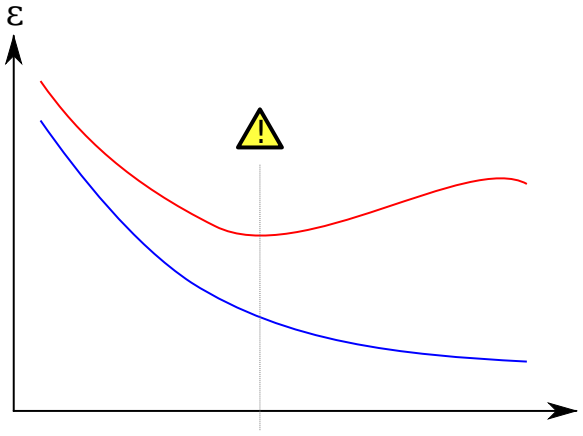
The section is based on [7, Chapter 7]. In this section will be introduced the concepts of overfitting and underfitting. As the essence of machine learning is to approximate a target function to fit some training data iteratively, there is a risk of fitting the function either too well or too bad to the training data. If the target function is fitted too well to the training data we may see that it does not generalize well to unseen testing data. On the other hand, if the target funcion is underfitted we do not see good performance on either the training data or the testing data.

In fig. 2.3a is shown an example of a target function that is overfitted to the training data (green line) and a target function that provides better generalisation to new data (black line).

Thus we want target function that generalises the best to both the unseen data and the training data. In order to obtain this we must stop the training at the right time. in fig. 2.3b is illustrated the loss of the training data (blue line) and the loss of the testing data (red line). This is only an illustration of the concept and is not subject to an actual training set. It is seen that even though the loss on the training data keeps decreasing at some point the loss of the testing data start increasing. In the example of fig. 2.3a this is the expected behavior once we start learning the outliers of the training data and thus obtain the green line. Instead we want to stop the training once we have obtained the black line. This is expected to be just before the testing loss start decreasing as seen in fig. 2.3b



(a) Illustration of overfitting to the training data (green line). Figure by Chabacano[16]



(b) Illustration of the testing loss increasing because of overfitting. Figure by Gringer[16]

Figure 2.3: Illustration of overfitting

2.4.1 Regularization

In the literature there are a variety of different methods to avoid overfitting, which are generally referred to as *regularization* techniques. In this short overview we will present only two of such techniques: early stopping and dropout. Before presenting these techniques will be provided a short overview of how data is usually split up during training.

2.4.2 Training, validation and testing sets

So far in this overview of theory we have only considered the training set and the testing set, however in practice an additional set is usually used. That is the validation set.

Thus the purpose of the three sets are as follows:

Training set:

The set of data that is used for learning. The model is aiming to approximate a function that minimises the loss on this set.

Validation set:

A set of the data that is used for providing unbiased evaluations of the model during training. Evaluation on the validation set is often employed after each training epoch. This is usually taken as a subset of the training data. This subset is not used for training.

Testing set:

A set of data that is used for providing unbiased evaluation of the model after training is finished

Early stopping

The idea of early stopping is very simple. After each epoch the model is evaluated on the validation split. We keep record of this performance and after a predefined number of epochs with no further improvement on the validation set we stop training. Thus we also save training time as we do not necessarily have to train all the planned epochs. In the example of fig. 2.3b using early stopping we will stop training a set distance after the minimum loss is observed and then use the model minimising validation loss for further testing.

Dropout

Another regularization technique is dropout. This procedure aims to reduce overfitting by "turning off" neurons during training. The neurons are randomly dropped out with a probability **P** which is typically P=0.2 for input layers and P=0.5 for hidden layers. Using dropout we perform the forward step, the backpropagation and the parameter updates using the non-dropped-out parameters only.

Due to the lower number of weights we need to scale the outputs by a factor of $\frac{1}{1-P}$.

2.5 Evaluation of VAD

As the last part of this chapter on theory will be presented two ways of evaluating the performance of a VAD algorithm. Given that VAD is a binary classification problem, we consider first the accuracy and thereafter a more sophisticated approach: the Receiver Operating Characteristic (ROC)

2.5.1 Accuracy

A popular way of evaluating a VAD algorithm is by finding its accuracy. That is, simply put, how many of the VAD frames that is classified correctly divided by the total number of frames. In a binary classification problem we consider four scenarios in which the outputs lie. That is:

- True Positive (TP)
- True Negative (TN)
- False Positive (FP)
- False Negative (FN)

The accuracy can then be found as:

$$\text{Accuracy} = \frac{\text{TP+TN}}{\text{TP+TN+FP+FN}} \tag{2.35}$$

The accuracy however may not always be the best metric for evaluating performance. That is for example if the data is heavily biased towards either 0 or 1. In that case a high accuracy may be achieved by simply giving the same output regardless of the input.

2.5.2 Area Under the Curve

Because of this we introduce another way of evaluating a binary classification. That is by considering the ROC. The ROC curve is a plot that illustrates the True Positive Rate (TPR) against the False Positive Rate (FPR) as the classification threshold is varied. These are computed as:

$$\text{TPR} = \frac{\text{TP}}{\text{TP+FN}} \tag{2.36}$$

$$\text{FPR} = \frac{\text{FP}}{\text{FP+TN}} \tag{2.37}$$

Thus the TPR and FPR are normalised to the total occurrences of positives and negatives and thus are not prone to misleading results for biased data. In fig. 2.4 the ROC curve is illustrated

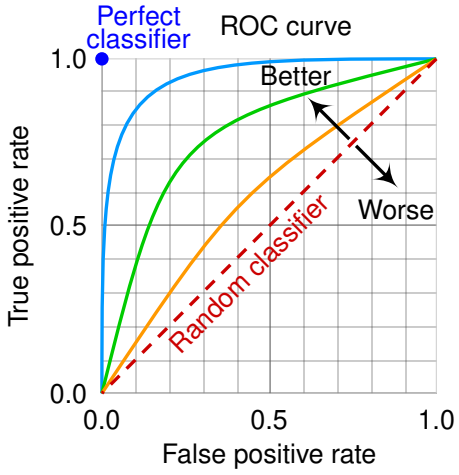


Figure 2.4: Receiver Operating Characteristic curve [17]

As illustrated, the larger the area under the curve is, the better the performance of the binary classification is. Therefore, through numerical integration of the ROC curve can be found a good metric for evaluation of VAD. From now on that is referred to as Area Under the Curve (AUC).

Setting up simulation environment 3

In this chapter it will be described how the simulation environment is set up based on the knowledge from chapter 2. The aim is to have the network perform as good as possible based on the theory, and then reproduce the results presented in [2] before the network will be further modified to accommodate for the three research questions presented in section 1.1.

3.1 Dataset used

First let us consider one of the datasets used in the original work of [2], the Aurora-2 dataset [6]. This dataset contains a traning set and three different test sets. All speechdata are derivatives of the TIdigits database downsampled to 8 kHz and filtered with a G712 characteristic. The spoken audio in this database is digits and letters. The Aurora-2 dataset contains true labels for VAD and both the training sets and the test sets are labelled as containing 73% speech. The labels used can be downloaded from the GitHub repository of [1].

The training set contains both clean speech and noise corrupted speech. The noise corrupted speech is at SNR levels of 20 dB, 15 dB, 10 dB and 5 dB. For the training set the following four noies types are used:

- Recording inside a subway
- Babble
- Car noise
- Recording inside an exhibition hall

The training set contains 8,440 files, whereof each unique combination of noise type and SNR level contains 422 files.

The three test sets all contain different noise types, however only set A and B will be considered here. Each test set contains the same speech which has been corrupted by different noise types at SNR levels of 20 dB, 15 dB, 10 dB, 5 dB 0 dB and -5 dB. In total each set contains 28,028 files, whereof each unique combination of noise type and SNR level contains 1,001 files.

The noise types of test set A are the same as those of the training set, while test set B is corrupted with the following noise types:

- Restaurant
- Street
- Airport

- Train station

Thus test set A can be considered to the model known noise types while test set B is unknown noise types.

3.1.1 Training, validation and test splits

In section 2.4.2 it was explained how the usual approach towards creating a validation set is to split the training set into two parts, such that a small part is reserved for unbiased validation. However, in this work a slightly different approach is taken due to the two different test sets used - set A and B which contains to the model known and unknown noise types respectively. In order to obtain true unbiased validations it has been decided to split both test sets into a validation and a test split. Thus the validation split is 1/3rd of the test set while the test split is the remaining 2/3rd of the files. Additionally, this way more files will be available for training.

3.2 Design choices

Based on the knowledge of the algorithm from section 1.2, the dataset from section 3.1 and the presentation of theory in chapter 2, in this section will be made some initial design choices that is left unanswered by the original work of [2].

3.2.0.1 Optimiser

In section 2.3.2 was introduced different popular algorithms for optimising a neural network based on a comparison study. Because of its fast convergence capabilities, its relatively few hyperparameters and its good performance has been chosen to use the RMSprop optimiser. Additionally, later in this work this particular optimiser will prove useful (chapter 6).

3.2.0.2 Normalisation

In section 2.3.3.3 was introduced a method for normalising the outputs from each layer such that training is faster and more stable. It is decided to use the batch normalisation after each Leaky ReLU layer as the output values of these are unbounded. The Sigmoid layers on the other hand are squeezing the outputs inside the range of [0,1] and for this reason it is not necessary to apply normalisation to these layers.

3.2.0.3 Parameter initialisation

In section 2.3.3.1 was introduced different ways of initialising parameters in a way such that the risk of exploding and vanishing gradients are minimised. In this work will be used Xavier initialisation for the Sigmoid layers and He initialisation for the Leaky ReLU layers.

3.2.0.4 Loss function

In section 2.2.3 was introduced common choices of loss functions for classification. In particular the cross entropy loss function and the binary cross entropy loss function. As the VAD task is a binary classification problem it is decided to use the binary cross entropy loss.

3.2.0.5 Evaluation

In section 2.5 two approaches towards evaluation of the VAD was introduced and as mentioned in section 3.1 the Aurora-2 dataset is labelled as 73% speech. Thus in order to avoid misleading results (i.e. by achieving a "good" performance by labelling too much audio as speech) it is chosen to disregard the accuracy and instead only consider the AUC when evaluating the VAD

3.2.1 Training

The last two design choices considered are directly related to training of the model. It is the value of the learning rate and the way in which the network is regularised in order to avoid overfitting to the training data.

3.2.1.1 Learning rate scheduling

Experimentally it has been found that using a learning rate starting at 0.01 ensures fast and stable convergence towards a solution. Additionally, to ensure more and more fine grained features are found the learning rate is gradually decreased by a factor of 0.7 after each epoch.

3.2.1.2 Regularisation

In section 2.4.1 was introduced two regularisation techniques for reducing overfitting. These were early stopping and dropout. It has been chosen to use the typical values for dropout presented, i.e. P=0.2 for the input layer and P=0.5 for the hidden layers.

Using the dropout a test has been carried out to see whether additional regularisation is needed. Further explanation of how this simulation is carried out can be seen in section 3.3.1:

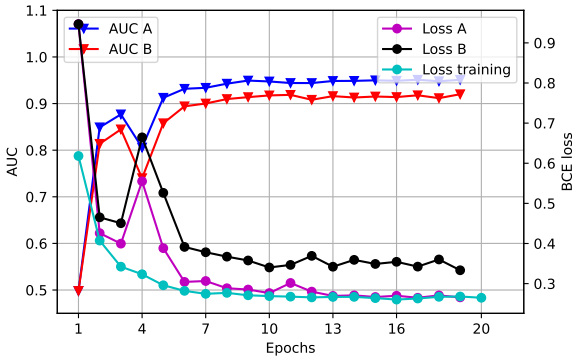


Figure 3.1: Validation curve used for deciding number of epochs to train.

From fig. 3.1 it is seen that even as the loss of the training set saturates the same behaviour as illustrated in fig. 2.3b is not observed. In fact it is seen that the validation loss of both test sets (as explained in section 3.1.1) follows the same pattern as the training loss. Additionally, the AUC has been found on the validation sets after each epoch and the same pattern is observed. Thus as no overfitting is observed it is decided to not use early stopping, but instead a fixed 20 training epochs.

3.3 Design of simulation environment

Now that the initial design choices have been made we will now move on to implementation of the network. First we consider the framework from fig. 1.1 and instead introduce an illustration that includes more information about the algorithm, and also includes both the forward step and the backward step. This illustration also allows for easy modifications later on in this work:

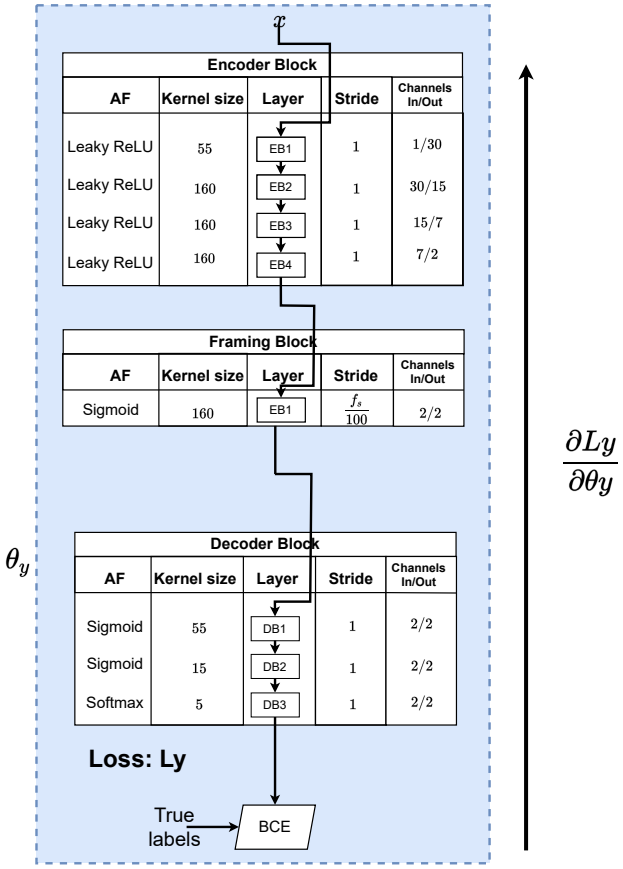


Figure 3.2: Visual representation of the network used as the framework of this project. The network is proposed in [2] while this figure is made by the author of this thesis.

3.3.1 Design of environment

As the framework used does not include publicly available source code, it will have to be built from scratch in this work. Some initial choices on the platforms used is listed below:

- The network will be implemented in Python
- The *PyTorch* [18] module will be used as it offers a lot of built-in functionality in terms of neural network implementation, loss functions, data loading, Graphics processing unit (GPU) kernels etc.
- Operations not supported by PyTorch will be implemented using *Numpy* and *SciPy*. The plots will be made using *Matplotlib* and the result files will be saved using *Pickle*
- The network will be trained on a desktop with an *NVIDIA GTX1080* GPU

As described in section 2.4 a CNN is prone to overfitting. To make sure the network will not overfit to the training set, the main loop of the algorithm will be set up as follows:

Algorithm 3: Main loop

for $n = 0$ **to** *epochs* **do**
  Run training loop
  Run validation loop
end
Run testing loop

The validation after each training will be used to make sure the model is not overfitting to the training data. Later in this chapter the training and validation/testing loops will be further described.

3.3.2 Model initialisation

First however, let us consider how the network is to be initialised in order to stay true to the work presented in [2]. Only little information about the algorithm is described and thus many design choices is left. These design choices will be based on the theory presented in chapter 2. As the network is a FCN all the layers consists of only convolutions and an activation function. Because of this most of the design choices are similar for every layer, and thus it is decided to only consider the first layer in this description. A code snippet showing the initialisation of the first layer of the network implemented in PyTorch is seen in listing 3.1.

The class contains a method for initialising the model and a method for defining the dataflow in the forward step.

Code Listing 3.1: Initialisation of the model in Python

```
import torch
from torch import nn

class VAD_model(nn.Module):
    def __init__(self):
        """Initialisation of the model"""
        super(VAD_model, self).__init__()
        self.EB1 = nn.Conv1d(1,30,55,stride=1,padding='same')
        nn.init.kaiming_uniform_(self.EB1.weight, 0.01, 'fan_in','leaky_relu')
        self.relu = nn.LeakyReLU(negative_slope=0.01)
        self.dropEB1 = nn.Dropout(p=0.2)
        self.bnormEB1 = nn.BatchNorm1d((30))

    def forward(self, x):
        """Define the dataflow in the forward step of the model"""
        x = self.EB1(x)
        x = self.relu(x)
        x = self.bnormEB1(x)
        x = self.dropEB1(x)

        return x
```

The design choices are seen in the initialisation method, and elaborated in the following:

- The channels in, channels out, filter size and stride are defined in [2]. *padding='same'* defines that the length of the output feature map is to remain the same as the length of the input feature map. In order to accomplish this zeroes are padded as introduced in section 2.2.1.
- The weights of the network is initialised using a uniform He (sometimes called Kaiming) initialisation as presented in eq. (2.30). *'fan in'* means that the variance of the weights are preserved in the forward step. 'leaky relu' means the weights are further optimised to retain the variance using the Leaky ReLU activation function.
- The Leaky ReLU layer is initialised using a negative slope of 0.01, which was found to be a commonly used value in section 2.2.2.
- During training 20% of the weights in the layer is randomly dropped out to improve regularization of the network and avoid overfitting. This was presented in section 2.4.1.
- In order to speed up the training a batch normalisation is introduced after each layer as presented in section 2.3.3.3

Once the network is initialised the forward pass can be called. In this case the input is called x, and after propagating the input through the entire model the output is returned. This output is then used for further processing such as computing the loss, making VAD predictions and computing the AUC.

3.3.3 Training loop

Having initialised the model, this section will elaborate the thoughts about the training loop as presented in algorithm 3. The key tasks of the training loop is the following:

- Fetch an audio file and perform the forward step
- Compute the loss between true labels and predicted VAD labels
- Backpropagate through the network and update the parameters

The following code snippet shows the core functionality of one epoch of the training loop implemented using PyTorch. First the optimiser is initialised. As mentioned in section 2.3.2.2 the RMS optimiser will be used in this work. Afterwards the forward step is computed, and the loss is calculated and accumulated. Once *batch size* number of files have forward propagated through the network, the backward step will be executed. In PyTorch the propagation graph is saved when performing a forward pass. This propagation graph is then used for efficiently calculating the new gradients using backpropagation as described in section 2.3.1. Lastly, the parameters are updated based on their gradients. It is important to note that the order in which the files are shown to the network is randomly shuffled in each epoch.

Code Listing 3.2: The training loop implemented in Python.

```
import torch
from torch import nn

optimizer_EB1 = torch.optim.RMSprop(VAD.EB1.parameters(), lr=learning_rate)

for file_number, (audio, true_labels) in enumerate(data_set):
    VAD_predictions = VAD_model(audio) #Forward step
    loss += nn.BCELoss(VAD_predictions, true_labels) #binary cross entropy loss

    if file_number % batch size == 0:
        optimizer_EB1.zero_grad()  # Zeroing gradients
        loss.backward() # Calculating new gradients
        optimizer_EB1.step() # Performing parameter updates
```

3.3.4 Validation loop and testing loop

After each training loop a validation loop can be carried out for two reasons: to ensure we are not overfitting the network and to keep track of the performance of the network. The testing loop operates similarly to the validation loop in this work, and thus they will be introduced together. The only difference between the validation and the testing loop is the data they are operating on. In section 2.4.2 was described how a small part of the training set is usually left for validation. Even though this is standard procedure in the field of machine learning, a slightly different approach has been taken in this work. That is because in this work we are interested in the network learning the characteristics of speech, and not those of noise. For this reason both known (test set A) and unknown noise types (test set B) will be used for validation to ensure we are not overfitting to the noise types.

Thus a validation and a test split is made from each test set. The validation split contains 1/3rd of the files while the testing split contains the remaining 2/3rds.

Additionally, while validating/testing we are interested in the network performance at each individual noise type and SNR level. The testing loop will therefore loop over these. Following this, the key functionality of the validation and testing loops are as follows:

- Fetch an audio file from the appropriate split and compute the forward step
- Compute the loss between true labels and predicted VAD labels
- Compute the AUC

In the following code snippet is shown how the validation/testing loop is constructed. The end goal is to get the average loss and AUC over each unique combination of noise type and SNR level, using the entire split. As presented in section 2.5 the AUC is the area under curve of the ROC. Therefore this ROC curve is constructed and the AUC is computed using trapezoidal integration. In order to find the TPR and FPR of the network a sweep of different threshold values is carried out. As the last layer of the network is a sigmoid the outputs are limited to the interval [0,1] and thus full range can be covered by varying the threshold within the range [-1,1].

Code Listing 3.3: The testing/validation loop implemented in Python

```
import torch
from torch import nn
import numpy as np

def compute_AUC(TP, TN, FP, FN):
    TPR = TP/(TP+FN)
    FPR = FP/(FP+TN)
    AUC = -np.trapz(TPR, FPR)
    return AUC

for SNR in SNR_levels:
    for noise in noise_types:
        for file_number, (audio, true_labels) in enumerate(data_set):
            VAD_predictions = VAD_model(audio) #Forward step
            loss += nn.BCELoss(VAD_predictions, true_labels)

            for idx, threshold in enumerate(np.linspace(-1,1,samples)):
                TP, TN, FP, FN = hits(VAD_predictions, true_labels, threshold)

                TP_acc[idx] += TP # Accumulate for each threshold
                TN_acc[idx] += TN # -,-
                FP_acc[idx] += FP # -,-
                FN_acc[idx] += FN # -,-

    compute_AUC(TP_acc, TN_acc, FP_acc, FN_acc)
```

The two outer loops are looping over combinations of noise types and SNR levels. The third loop is looping over all the files in this exact dataset and computing VAD predictions and the loss. The inner loop is where the foundation of the AUC is computed. This threshold is swept over aforementioned range and using this threshold the number of TP, TN, FP, FN in this file is calculated. The number of these are then accumulated for each threshold value. This way we can compute the AUC over the entire data set. Lastly the AUC is computed.

The source code can be found on https://github.com/aau-es-ml/VAD-with-adversarial-multi-task-learning.

3.4 Reproducing the results

Now that the model network has been implemented, this section will focus on reproducing the results originally published in [2]. The results are published as both accuracy and AUC, thus both of these will be investigated, even though it was decided in section 3.2.0.5 that only the AUC will be considered for the further work.

Accuracy

In table 3.1 the accuracy of the 4 noise types of test set B from both the original paper and this work is presented. It is seen that the implementation in this work closely resembles the original results at lower SNR levels, however at higher SNR levels we see a difference in accuracy on 1-2.5%. This difference is not significant and thus we will carry on with the next results.

Test set B	SNR	Original results	This work
Restaurant	-5 dB	86.99	86.64
	0 dB	91.04	90.71
	5 dB	94.31	93.17
	10 dB	95.72	93.86
Street	-5 dB	86.89	87.26
	0 dB	91.17	90.56
	5 dB	94.27	92.87
	10 dB	95.81	94.11
Airport	-5 dB	87.96	86.62
	0 dB	91.95	90.56
	5 dB	94.54	92.94
	10 dB	95.89	93.96
Train	-5 dB	88.27	86.69
	0 dB	92.37	90.43
	5 dB	95.21	93.10
	10 dB	96.14	93.75

Table 3.1: Results from original WVAD paper

AUC

Next up we compare the computed AUC of the original paper to this work. The AUC is found as the average over 7 noise types at 6 SNR levels. However, it is not noted what these 7 noise types are. Instead the reproduced results will be averaged over the 8 noise types of both test set A and B. The comparison is seen in table 3.2.

AUC Aurora-2	Clean	20 dB	15 dB	10 dB	5 dB	0 dB	-5 dB	**Mean**
Original paper	-	99.45	99.40	99.27	98.80	97.04	92.29	*97.71*
This work	-	98.65	98.61	98.44	98.47	97.96	97.00	*97.68*

Table 3.2: AUC values when training and testing on the Aurora-2 database

It is seen that the AUC of the original results has a higher variance over the SNR levels by performing worse at low SNR levels and performing better at higher SNR levels. The difference is not significant. The mean however, is very similar. It has not been possible to achieve a closer resemblence, so this model will be used for further experimenting in this work.

Part II

Algorithm

Paper submitted to Interspeech 2022

4

As a part of this thesis has been submitted a paper to the *Interspeech 2022* conference. This paper covers a wide range of the work done in this thesis, however as the paper was limited to only 4 pages of content and 1 page of references, it has not been possible to describe the work and the thoughts behind thoroughly. For this reason, this part of the report will be based on the paper and provide some more elaborate theory, discussions and analysis of results.

First, on the next pages the paper is presented. The paper will give an overview of the work, whereafter the different aspects will be more thoroughly described, as mentioned above.

The paper use the algorithm presented in [2] as a baseline. This work then focuses on answering the first 2 of the research questions mentioned in section 1.1:

1. *Can we potentially increase the noise-robustness of a VAD without increasing its computational cost and latency to the execution-time?*
2. *How will it affect the performance of the VAD if we allow it to use less future samples to generate a VAD output and thus decrease the algorithmic delay?*

Adversarial Multi-Task Deep Learning for Noise-Robust Voice Activity Detection with Low Algorithmic Delay

Claus M. Larsen, Peter Koch, Zheng-Hua Tan

Department of Electronic Systems, Aalborg University, Denmark

cmla17@student.aau.dk, pk@es.aau.dk, zt@es.aau.dk

Abstract

Voice Activity Detection (VAD) is an important pre-processing step in a wide variety of speech processing systems. VAD should in a practical application be able to detect speech in both noisy and noise-free environments, while not introducing significant latency. In this work we propose to introduce an adversarial multi-task learning method when training a supervised VAD. The method has been applied to the state-of-the-art VAD *Waveform-based Voice Activity Detection*. Additionally the performance of the VAD is investigated under different algorithmic delays, which is an important factor in latency. Introducing adversarial multi-task learning to the model is observed to increase performance in terms of Area Under Curve (AUC), particularly in noisy environments, while the performance is not degraded at higher SNR levels. The adversarial multi-task learning is only applied in the training phase and thus introduces no additional cost in testing. Furthermore the correlation between performance and algorithmic delays is investigated, and it is observed that the VAD performance degradation is only moderate when lowering the algorithmic delay from 398 ms to 23 ms.

Index Terms: Voice Activity Detection, adversarial multi-task learning, algorithmic delay, deep learning, noise robustness

1. Introduction

Voice Activity Detection (VAD) aims to detect which segments of an audio stream contains speech and the segments are typically 10 ms each [1], [2], [3]. It is widely used as a pre-processing step in more complex audio signal processing tasks such as speech recognition [4], speaker verification [5] or speech enhancement [6], but can also be applied on its own to reduce the computational cost of downstream processing. While detecting speech in a noise-free environment is a trivial task, the difficulty in classification arises in noisy environments [7].

Algorithms for performing VAD can generally be categorized into classes of supervised and unsupervised methods. The unsupervised methods can be energy based [8], however, this approach is very sensitive to noisy conditions. More complex unsupervised methods are generally based on assumptions of speech and noise characteristics, e.g. using Mel Frequency Cepstral Coefficients (MFCCs) [9], [10], perceptural spectral flux [11] or pitch detection and spectral flatness [1].

In recent years supervised methods for VAD has gained increased popularity within the field of research [12], [13]. The supervised methods require large amounts of labelled speech data and their performance are highly dependent on the quality of the labelled data used for training and testing. Some supervised methods contains a pre-processing step which aims to extract useful features from the audio such as MFCCs [14], while other methods resolves to the raw waveform as their input [3], [15]. A benefit of using the raw waveform as input to the supervised VAD is that the method will potentially find the most optimal features to be used for classification on its own, and is therefore able to utilise both the magnitude and the phase of the audio [3]. Using the raw waveform as features to the VAD is an active area of research and has shown appealing performance in terms of noise-robustness.

Two important factors in a VAD algorithm are how noise-robust it is and how much latency it introduces. Adversarial multi-task learning has proven to be effective to make applications invariant to noise and thereby more noise robust, e.g. for speech recognition in [16] and speech enhancement in [17]. Noise robustness of speaker verification is largely boosted by adversarial training in [18]. When applying VAD in a real-world application, often the latency is of great concern. Even though VAD is an active area of research, the algorithmic delay it introduces is rarely explored.

In this work we aim to investigate if the noise-robustness can be increased even further by introducing an additional sub-network which aims to classify the noise types to a supervised VAD, and train the VAD adversarially to these. The supervised VAD method presented in [3], which is based on fully convolutional neural networks (CNN) and shows state-of-the-art performance on the AURORA2 dataset [19], will be used as the framework in this work. An additional discriminative sub-network for adversarial multi-task training, inspired by the work in [16] will be introduced. The additional network used for adversarial multi-task learning is only introduced in the training phase and thus introduces no additional cost in testing. Furthermore, we investigate the impact of different algorithmic delays on VAD performance and realise this by varying CNN kernel sizes. The source code for this work is publicly available on GitHub [1].

2. Proposed method

In this work we propose to introduce adversarial multi-task learning to enhance the robustness of deep model based VAD. Specifically, an additional sub-network for adversarial training is introduced to a state-of-the-art waveform-based VAD using a CNN model. The entire framework is illustrated in Figure 1, in which the adversarial-training sub-network is shown by the green box, the algorithmic delay is investigated by modifying the blue block, and the waveform-based VAD [3] consists of the blue and red blocks.

2.1. Framework

In realising the adversarial multi-task learning for VAD, we consider the model presented in [3] with our own implementation in Pytorch [20]. The method resorts to a fully convolu-

[1] https://github.com/aau-es-ml/VAD-with-adversarial-multi-task-learning

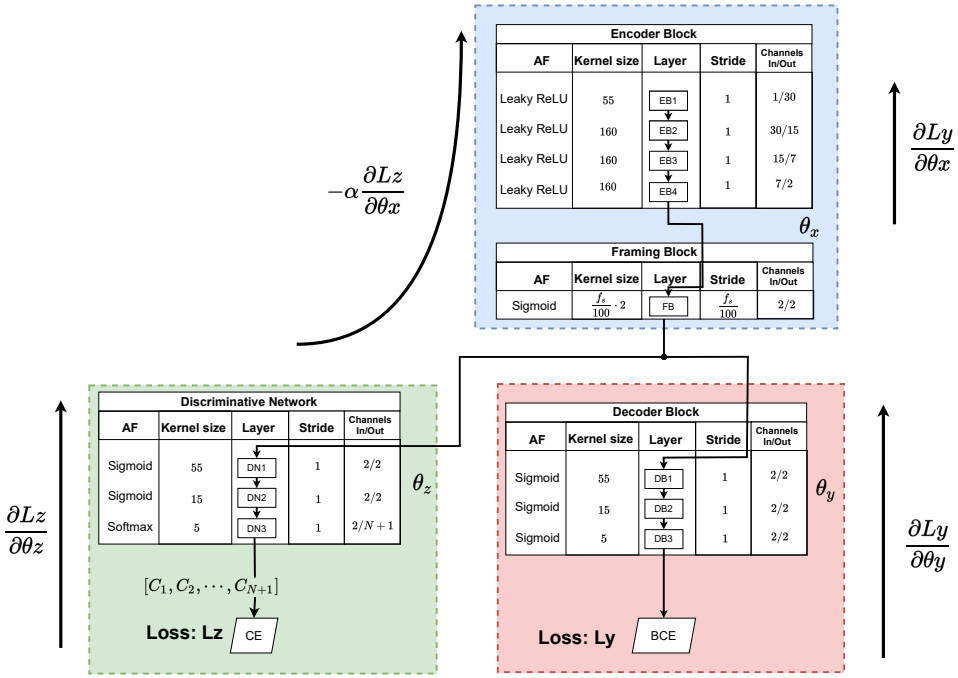


Figure 1: *Overview of the proposed discriminative network (the green block) applied to the waveform-based VAD (the blue and red blocks) proposed in [3].*

tional neural network. It consists of an Encoder Block (EB), a Framing Block (FB) and a Decoder Block (DB). The EB is taking as input the raw waveform of audio and the FB generates features on a 10 ms basis. The DB further refines these features into the VAD output, where the larger value of the two channels, *speech and non-speech*, determines the VAD label as speech or non-speech on a 10 ms basis.

In this work we introduce the additional Discriminative Network (DN). It takes as input the output from the FB and aims to learn to correctly classify the different noise types as well as to adversarially train the EB and FB part to make it noise robust.

2.2. Adversarial multi-task learning

The first part of this work is to introduce adversarial learning. The proposed DN is implemented in a similar way to the DB, where each layer resolves time-frequency representations in the channels of feature maps and the shrinking kernel sizes (55, 15, 5) reflect the decreasing modulation frequency (1.83 Hz, 6.66 Hz, 20 Hz) with the segment rate being 100 Hz [3]. The discriminative network generates softmax probabilities for each of the $N+1$ channels (i.e. outputs), where N is the number of different noise types in the training set, while the remaining channel is for clean speech. The labels used for the DN is the noise types on a 10 ms basis, similarly to the VAD labels.

Following the DN, the cross-entropy loss is calculated between the noise types predicted by the DN and the true noise types. Following the DB is the binary-cross-entropy loss calculated between the VAD labels and the truth labels. The losses are expressed as:

$$L_z = -\sum_i t_i \log(p_i) \tag{1}$$

$$L_y = -\left[t\log(p) + (1-t)\log(1-p)\right] \tag{2}$$

where t is the true labels and p is the scores output by the networks on a 10 ms basis.

When backpropagating the error through the model, the gradients of the DN are updated based only on the loss Lz, the gradients of the DB are updated based only on the loss Ly while the gradients of the EB and the FB are updated based on both losses. However, the sign of the gradients calculated from Lz is flipped such that the EB and FB are trained adversarially to the DN and friendly to the DB. Additionally the magnitude of this gradient is multiplied by a scalar α that determines the contribution from this sub-network. The key idea behind the method is that the FB will then output features that are invariant to the noise type which in turn will lead to a more noise-robust VAD and hence better VAD performance. The DN is illustrated in Figure 1. The gradients are noted as the partial derivatives of the loss function L with respect to the parameters θ.

The convolutional operations of the DN can be expressed as:

$$y_{[c]}^{[l]}(\tau) = \mathrm{AF}\left(\left(\mathbf{F}_{[c]}^{[l]} * y_{[c]}^{[l-1]}(\tau)\right) + \mathbf{b}_{[c]}^{[l]}\right) \tag{3}$$

where c denotes the channel, l denotes the layer, $\mathbf{F}_{[c]}^{[l]} \in \mathbb{R}^{C\times k}$ is the convolutional kernel, $\mathbf{b}_{[c]}^{[l]} \in \mathbb{R}^{C\times 1}$ is the bias, $y_{[c]}^{[l]} \in \mathbb{R}^{C\times \max(\tau)}$ is the feature map and AF is the activation function.

2.3. Algorithmic delay

Second part of this work focuses on reducing the latency of the VAD. Only the algorithmic delay is considered, i.e. it is investigated how the VAD performance is affected based on how many future samples is used in the predictions, from here on referred to as *future context*. The amount of future temporal context used for a given classification is calculated based on the fact that the feature map through a 1-dimensional convolutional layer will shrink as stated by Eq. 4.

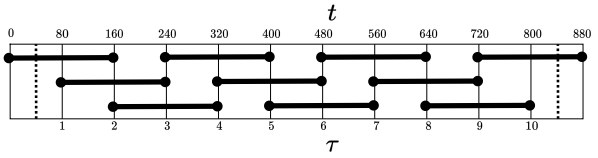


Figure 2: *Illustration of the context needed to generate an output from the FB. In this example 10 outputs are generated from an 8 kHz signal, but because of the 50% overlap between frames, an additional* $\frac{fs}{100} - 1$ *samples of context is needed to generate an output.*

$$n^{[l]} - k + 1 = n^{[l+1]} \tag{4}$$

where $n^{[l]}$ is the size of the feature map generated by the l^{th} layer and k is the kernel size. The number of samples by which the feature map is shrinking will have to be considered as context where half of it is past and the other half is future. The algorithmic delay introducd by the network is found by calculating how much context is needed in each layer and finally summing them together. The context introduced in the EB layers is simply found as in Eq. (4), while the context introduced in the framing block is more complex and best illustrated by Figure 2. The context introduced by the DB is dependent on the stride of the FB and once again calculated using Eq. (4). The total algorithmic delay in seconds is found by dividing the context with the two times the sampling rate and is found as Eq. (5).

$$AD = \frac{\sum_{n=1}^{4}(k_{EBn} - 1) + \frac{fs}{100} - 1 + \sum_{i=1}^{3}(k_{DBi} - 1) \cdot \frac{fs}{100}}{2fs} \tag{5}$$

3. Speech corpora

In this work two speech corpora are used. The AURORA2 [19] database and the TIMIT [21] database.

3.1. AURORA2

First is the AURORA2 [19] database which is used for multi-condition training at a sampling frequency of 8 kHz. The training set consists of 8440 utterances with four noise types artificially added at SNR levels of 5 dB, 10 dB, 15 dB, 20 dB and *clean*. The four noise types used are *subway, babble, car* and *exhibition hall*. For each combination of noise type and SNR level 422 utterances are used.

AURORA2 contains three test sets. Of these two are used in this work. Test set A uses the same noise types as the training set, and test set B uses four noise types unknown to the training set. These are *restuarant, street, airport* and *train station*. In the test sets the following SNR levels are used: -5 dB, 0 dB, -5 dB, 10 dB, 15 dB, 20 dB and *clean*. The test sets each consists of 4004 utterances which are evenly distributed on the four noise types and repeated for each SNR level. The true VAD labels are from the open-source rVAD repository [1].

3.2. TIMIT

Secondly, the TIMIT [21] speech corpus is used. The training set and test set, respectively, consists of 4620 and 1680 spoken sentences. For use in this work 6 noise types are artifically added by the authors at SNR similar to those of the AURORA2 test sets. Different instances of the same noise type are used for training and testing sets such that no instance of noise is ever repeated. Furthermore, in this work the test set is split into validation and test sets using a 1/3, 2/3 split. The purpose of the validation split is to find the optimal hyperparameter α shown in Figure 1, while the test split is used to obtain the rest of the results in this work. The noise types used for training and testing are the same, meaning the noise types will be known under testing. The noise types *babble, bus, caf* and *pedestrian* are from the CHiME3 dataset [22] while *babble* and *speech shaped noise* are generated by the authors of [23]. The noise is artifically added as described in section 2.A of [24]. The true labels for the TIMIT database is generated using the .WRD files which states at which time stamps speech are present. These labels are shared along with the source code of this work which are publicly available on GitHub.

4. Experimental setup and results

In order to evaluate the performance of the proposed method for adversarial training, the model is trained and tested on both the TIMIT database and the AURORA database. When evaluating Eq. (5) with the kernel sizes shown in Figure 1, the context to generate a VAD label spans 398 ms to both sides. This means that to generate the first/last VAD label of each file, 398 ms of context is missing. This in combination with the short duration of the files of AURORA2 (typically 0.8-2 seconds) and TIMIT (typically 2-4 seconds) leads to that a large part of the VAD outputs of each file will be generated without sufficient context. Additionally, the files in these data sets are all following the same structure. That is a short duration of silence in the beginning and the end, while the middle part contains speech. This leads to that the VAD learns to recognize this structure based on the missing context in the beginning and the end. To deal with this problem, during training 10 files are randomly chosen and concatenated leading to longer inputs to the VAD and therefore a smaller part of the VAD outputs will be generated based on insufficient context. The reason for using 10 files is that the computer on which the model is trained runs into memory problems when using more files. This forward CNN calculation step is performed three times before each backward step leading to a mini-batch size of 30 audio files. For each three forward steps a single backward step is performed using the RMSprop optimiser. The model is trained over 30 epochs, the learning rate is initialised as 0.01 and the learning rate is multiplied by 0.7 after each epoch.

Table 1: *AUC values on the validation sets using different values of* α

α	**0**	**0.01**	**0.1**	**1**	**10**	**100**
AURORA2 A	93.90	93.93	94.44	94.58	**95.18**	92.97
AURORA2 B	91.15	91.22	**92.10**	91.99	91.85	90.99
TIMIT	88.78	89.98	**90.32**	89.3	88.91	87.94

4.1. Adversarial multi-task learning

First the optimum value of the scalar α is found experimentally on both data sets by using validation data. Given that the AURORA2 is labelled as 73% speech and TIMIT is labelled as 85% speech, the results will be given by calculating the Area Under Curve (AUC) of their respective Receiver Operating Characteristics (ROC) curves while the accuracy will be disregarded as it can be misleading. For finding the optimum value of α the average AUC over the noise types of the validation split at each SNR

level is calculated. In each experiment the model is initialised using the same set of parameters to remove the randomness that can potentially be introduced by different initialisations. The Leaky ReLU layers are initialised using He [25] while the parameters of the sigmoid layers are initialised using Xavier [26]. The Leaky ReLU layers use a slope coefficient of 0.01. The average AUC at $\alpha = 10^n, n \in [-2, -1, 0, 1, 2]$ on the validation sets are presented in Table 1. It is seen that the performance of the VAD is increased by a wide range of values of α and in general the addition of a discriminative network for adversarial multi-task learning outperforms the baseline model in terms of AUC. In the case of all 3 test sets, and thereby also to the model both known and unknown noise types, it is found that a wide range of values of α results in an increase in performance. In two of three cases a value of 0.1 is found to be optimal, thus this will be the value used for further experiments in this work.

The performance of the model using an α value of 0.1, which was found optimal on the validation splits, is further investigated using the test splits of each data set. The results are seen in Table 2. Once again the models are trained from the same initial values using the same simulation settings as described earlier and it is clearly seen that while the performance at high SNR levels is approximately the same with and without adversarial multi-task learning, at lower SNR levels the models trained using adversarial multi-task learning performs better and proves to be more noise-robust. This is the case both when it comes to noise types known to the model (TIMIT and AURORA2 test set A) and noise types unknown to the model (AURORA2 test set B).

Table 2: *AUC values on the test sets of AURORA2 and TIMIT with **(W)** and without **(W/O)** adversarial multi-task learning. When adversarial multi task learning is used, an α value of 0.1 is used*

	Clean	20 dB	15 dB	10 dB	5 dB	0 dB	-5 dB	**Mean**
AURORA2 A - W/O	**98.46**	**98.46**	**98.37**	98.19	97.05	91.19	75.58	*93.90*
AURORA2 A - W	98.42	98.40	98.32	98.19	**97.19**	**92.81**	**77.78**	***94.44***
AURORA2 B - W/O	**98.46**	**98.44**	**98.22**	**97.58**	94.36	84.90	66.08	*91.15*
AURORA2 B - W	98.42	98.37	98.13	97.39	**94.67**	**87.51**	**70.24**	***92.11***
TIMIT - W/O	95.44	95.41	94.63	92.64	88.60	82.67	71.96	*88.76*
TIMIT - W	**95.62**	**95.49**	**94.92**	**93.91**	**90.75**	**84.94**	**74.26**	***89.99***

4.2. Algorithmic delay

The second part of this work is to evaluate the performance of the proposed method at lower algorithmic delays. The model is trained with the optimum value of $\alpha = 0.1$ while the kernel sizes of the decoder block is reduced. The algorithmic delay is calculated as a function of the sampling frequency and the kernel sizes as in Eq. (5). The majority of the algorithmic delay is introduced by the DB, thus only these kernel sizes will be varied. The kernel sizes and their corresponding algorithmic delays as used in the experiments are presented in Table 3. The AUC is calculated as the average over the 7 SNR levels and the noise types of each test set.

It is seen that the performance of the VAD decreases as the algorithmic delay is lowered, however this performance decrease is not drastic. It is in particular interesting to note that even when completely disregarding the decoder block with an algorithmic delay of 23 ms the VAD still performs well. When decreasing the algorithmic delay from 398 ms to 23 ms, only a performance decrease of 7% AUC is seen. The performance of different algorithmic delays at each SNR level for AURORA2 test set B is presented in Figure 3. In particular the performance at clean speech seems to be unaffected by the low algorithmic delay.

Table 3: *VAD performance in terms of AUC at different kernel sizes and algorithmic delays tested on AURORA2 test sets A and B*

DB1	**DB1**	**DB1**	**AD [ms]**	**AURORA2 B**	**AURORA2 A**
55	15	5	*398*	92.11	94.44
45	15	5	*348*	91.91	94.38
35	15	5	*298*	90.81	93.44
25	15	5	*248*	91.04	93.32
15	10	5	*173*	90.95	93.01
10	7	5	*133*	88.09	90.84
7	5	5	*98*	89.01	91.85
5	3	3	*78*	87.14	89.78
3	3	2	*63*	85.77	90.00
2	2	2	*53*	85.26	87.56
2	2	0	*43*	85.03	87.40
2	0	0	*33*	85.66	87.90
0	0	0	*23*	85.07	87.27

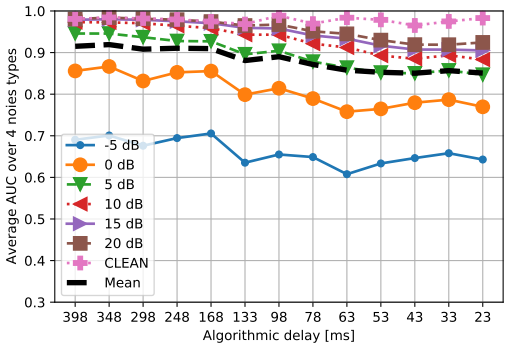


Figure 3: *Average AUC at decreasing algorithmic delays on the, to the model, unknown noise types of AURORA2 test set B*

5. Conclusions

In this paper we proposed a novel approach of training a supervised VAD using adversarial multi-task learning, where the model is trained friendly to the VAD labels and adversarially to the noise types aiming to make the VAD more invariant to noise. This is done by introducing an additional sub-network which aims to classify the noise types to the model. The VAD is then trained adversarially to these. It is found that the adversarial multi-task training is capable of increasing the VAD performance especially in more noisy environments, and it is shown to increase performance both when presented to unknown and already known noise types. On SNR levels of -5 dB the performance is boosted with up to 4% AUC. This multi-task learning is only used when training the model and disregarded under testing, meaning the proposed method is cost-less once training has finished.

Furthermore it was investigated if this method can be useful in a low-latency application. This was done by reducing the kernel sizes of the DB resulting in lower algorithmic delays. It was found that even at an algorithmic delay as low as 23 ms, at which point the DB is completely disregarded, the performance of the method was still good. When decreasing the algorithmic delay from 398 ms to 23 ms the performance is only reducd by 7% AUC.

6. References

[1] Z.-H. Tan, A. Sarkar, and N. Dehak, "rvad: An unsupervised segment-based robust voice activity detection method," *Computer Speech and Language*, vol. 59, pp. 1–21, 2020. [Online]. Available: https://github.com/zhenghuatan/rVAD

[2] J. Ramírez, J. Gorriz, and J. Segura, *Voice Activity Detection. Fundamentals and Speech Recognition System Robustness*, 06 2007, vol. 6(9).

[3] C. Yu, K.-H. Hung, I.-F. Lin, S.-W. Fu, Y. Tsao, and J.-w. Hung, "Waveform-based voice activity detection exploiting fully convolutional networks with multi-branched encoders," *arXiv preprint arXiv:2006.11139*, 2020.

[4] A. Baevski, W.-N. Hsu, A. Conneau, and M. Auli, "Unsupervised speech recognition," *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[5] B. Chettri, E. Benetos, and B. L. Sturm, "Dataset artefacts in anti-spoofing systems: a case study on the asvspoof 2017 benchmark," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 28, pp. 3018–3028, 2020.

[6] P. Hoang, Z.-H. Tan, J. M. De Haan, and J. Jensen, "The minimum overlap-gap algorithm for speech enhancement," *IEEE Access*, vol. 10, pp. 14 698–14 716, 2022.

[7] H. Dinkel, S. Wang, X. Xu, M. Wu, and K. Yu, "Voice activity detection in the wild: A data-driven approach using teacher-student training," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 29, pp. 1542–1555, 2021.

[8] C.-C. Hsu, C. Kah Meng, T.-S. CHI, and Y. Tsao, "Robust voice activity detection algorithm based on feature of frequency modulation of harmonics and its dsp implementation," *IEICE Transactions on Information and Systems*, vol. E98.D, pp. 1808–1817, 10 2015.

[9] T. Kinnunen and P. Rajan, "A practical, self-adaptive voice activity detector for speaker verification with noisy telephone and microphone data," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013, pp. 7229–7233.

[10] P. Mahalakshmi, "A review on voice activity detection and mel-frequency cepstral coefficients for speaker recognition (trend analysis)," *Asian Journal of Pharmaceutical and Clinical Research*, vol. 9, p. 360, 12 2016.

[11] S. O. Sadjadi and J. H. Hansen, "Unsupervised speech activity detection using voicing measures and perceptual spectral flux," *IEEE SIGNAL PROCESSING LETTERS*, vol. 20, no. 3, p. 197, 2013.

[12] D. Rho, J. Park, and J. H. Ko, "Nas-vad: Neural architecture search for voice activity detection," *arXiv preprint arXiv:2201.09032*, 2022.

[13] Y. Lee, J. Min, D. K. Han, and H. Ko, "Spectro-temporal attention-based voice activity detection," *IEEE Signal Processing Letters*, vol. 27, pp. 131–135, 2019.

[14] X.-L. Zhang and J. Wu, "Deep belief networks based voice activity detection," *IEEE TRANSACTIONS ON AUDIO, SPEECH, AND LANGUAGE PROCESSING*, vol. 21, no. 4, p. 697, 2013.

[15] R. Zazo-Candil, T. N. Sainath, G. Simko, and C. Parada, "Feature learning with raw-waveform cldnns for voice activity detection," in *INTERSPEECH*, 2016.

[16] Y. Shinohara, "Adversarial Multi-Task Learning of Deep Neural Networks for Robust Speech Recognition," in *Proc. Interspeech 2016*, 2016, pp. 2369–2372.

[17] Z. Meng, J. Li, Y. Gong, and B.-H. F. Juang, "Adversarial feature-mapping for speech enhancement."

[18] H. Yu, Z.-H. Tan, Z. Ma, and J. Guo, "Adversarial network bottleneck features for noise robust speaker verification," *Proc. Interspeech 2017*, pp. 1492–1496, 2017.

[19] D. Pearce and H.-G. Hirsch, "The aurora experimental framework for the performance evaluations of speech recognition systems under noisy condition," vol. 4, 01 2000, pp. 29–32.

[20] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[21] J. Garofolo, L. Lamel, W. Fisher, J. Fiscus, D. Pallett, N. Dahlgren, and V. Zue, "Timit acoustic-phonetic continuous speech corpus," *Linguistic Data Consortium*, 11 1992.

[22] J. Barker, R. Marxer, E. Vincent, and S. Watanabe, "The third 'chime' speech separation and recognition challenge: Dataset, task and baselines," in *2015 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*, 2015, pp. 504–511.

[23] M. Kolbœk, Z.-H. Tan, and J. Jensen, "Speech enhancement using long short-term memory based recurrent neural networks for noise robust speaker verification," in *2016 IEEE Spoken Language Technology Workshop (SLT)*, 2016, pp. 305–311.

[24] M. Kolbæk, Z.-H. Tan, and J. Jensen, "Speech intelligibility potential of general and specialized deep neural network based speech enhancement systems," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 25, no. 1, pp. 153–167, 2017.

[25] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *CoRR*, vol. abs/1502.01852, 2015. [Online]. Available: http://arxiv.org/abs/1502.01852

[26] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *AISTATS*, 2010.

Choice of speech corpora 5

A fundamental part of the work done is choosing what datasets to use for training and testing, and how to use these datasets in a way that both resembles a real-life scenario while also staying true to the academia. This is also presented in section **3** of the paper.

5.1 Aurora-2 database

As the original work introduced in [2] is trained and tested on the Aurora-2 database, it seems intuitive to also use this database for further work. In academia the go-to approach for testing a VAD algorithm on the Aurora-2 data base is to test on one file at a time and evaluate the performance as the mean performance over the dataset. In the Aurora-2 dataset all files follow the same structure. That is:

- A short duration of silence in the beginning. Typically around 200 ms
- A spoken utterance in the middle. Typically 400-2000 ms.
- A short duration of silence in the end. Typically around 200 ms

An example of an audio file of the Aurora-2 database is shown in fig. 5.1.

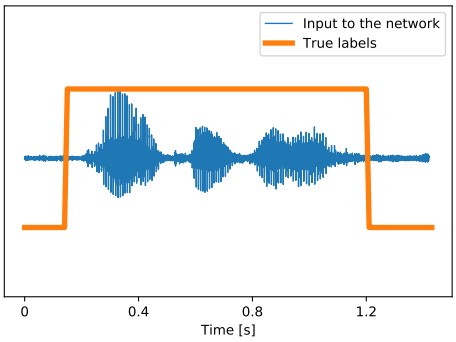


Figure 5.1: An example of an audio file from the Aurora-2 database

As mentioned in chapter 2 a CNN is simply aiming to minimise its loss and therefore does not necessarily find patterns in the things the designer is aiming for. In the case of VAD it is desired to find patterns in speech. But due to the typical structure of the files of the dataset it is of interest to find if this structure is learnt by the network and thus artifically enhancing the VAD performance. First will be analysed how much context (i.e. the number of future and past samples) is needed to generate a VAD output. Following this analysis a few experiments will be carried out.

5.1.1 Analysis of context

In this section context needed to generate a VAD output will be found and analysed. The analysis will focus on the algorithm as three seperate blocks: (EB, FB, DB) each introducing their own context and then summing it in the end to find the total context.

First we will consider some properties that applies to the context in all three blocks. The necessary context will be computed similarly to how much padding is needed to generate a VAD label spanning 10 ms when only 10 ms of audio is available as input.

As introduced in section 2.2.1, the size of the feature map n through a layer will shrink depending on the filter size k as noted by eq. (5.1) unless padding is applied:

$$n^{[l+1]} = n^{[l]} - k + 1 \tag{5.1}$$

This expression will be rewritten and will be the foundation for further discussions on context:

$$n^{[l]} = n^{[l+1]} + k - 1 \tag{5.2}$$

5.1.1.1 Encoder Block

The context introduced by the EB can simply be found by considering eq. (5.2) where it is seen that the necessary padding to retain the size of the feature map is $k - 1$. This covers both past and future context, and thus to find how much context is needed on each side of the input it is divided by 2:

$$\frac{k-1}{2} \tag{5.3}$$

And in order to find the combined context in seconds introduced in the EB, the context of the four layers are summed up and divided by the sampling frequency:

$$\frac{\sum_{n=1}^{4}(k_{EBn} - 1)}{fs} \tag{5.4}$$

5.1.1.2 Framing Block

In the FB a stride dependent on the sampling frequency is introduced. This stride is equal to the number of samples that fit in a 10 ms frame, while the kernel size is the double of that. This means that there is 50% overlap between frames. The stride does not have any direct influence on the size of the context, thus the algorithmic delay introduced in this layer is the overlap resulting in ≈ 5 ms.

The way in which the FB is downscaling the feature map from time samples into 10 ms frames is illustrated in fig. 5.2. In this example 10 outputs is generated on an 8 kHz signal. The vertical dashed lines shows the additional context needed to generate the output feature map.

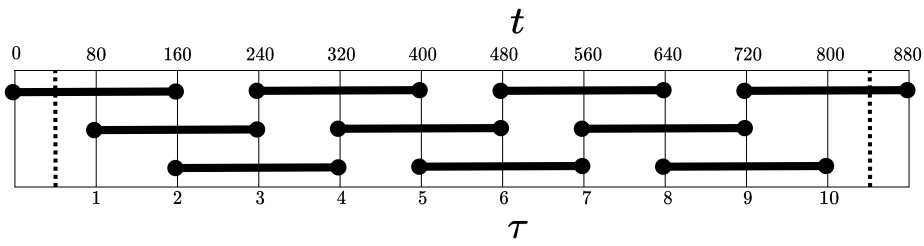


Figure 5.2: Illustration showing that the context needed to generate n samples is $\frac{fs}{100} \cdot (n+1)$

From fig. 5.2 it is seen that in order to generate n VAD labels a total of:

$$\frac{fs}{100} \cdot (n+1) \tag{5.5}$$

samples is needed.

5.1.1.3 Decoder Block

The DB takes as input the frames on a 10 ms basis generated by the FB. This needs to be taken into account when calculating the context being introduced. Given that $\frac{fs}{100}$ time samples is needed to generate a 10 ms frame, the context size will increase by the same factor.

Thus the context introduced by the DB can be found as:

$$\frac{k-1}{2} \cdot \frac{fs}{100} \tag{5.6}$$

5.1.2 Finding total context

Combining the context needed to retain the correct size of the feature map through the layers as seen in eq. (5.4), eq. (5.5) and eq. (5.6) and dividing by the sampling frequency the total context in seconds can be found as:

$$\text{context in seconds} = \frac{\sum_{n=1}^{4}(k_{EBn}-1) + \frac{fs}{100} + \sum_{i=1}^{3}(k_{DBi}-1) \cdot \frac{f_s}{100}}{f_s} \tag{5.7}$$

Inserting the original filter sizes as presented in [2] in the expression in eq. (5.7), the necessary input size to generate one output of a 10 ms frame is 806 ms. 10 ms of these correspond to the output frame, leaving 398 ms of both past and future context. This effectively means that in order to label a file from the Aurora-2 database, it is necessary to pad 398 ms of zeroes in both the end and the beginning.

In order to illustrate the effect of this, on fig. 5.3 is shown a randomly selected file from the Aurora-2 database. Towards the beginning and the end of thsis file is padded 398 ms of zeroes to make sure the output is of correct size. Furthermore is illustrated the amount of context that is needed to generate one VAD output as red, while the yellow square is the 10 ms frame.

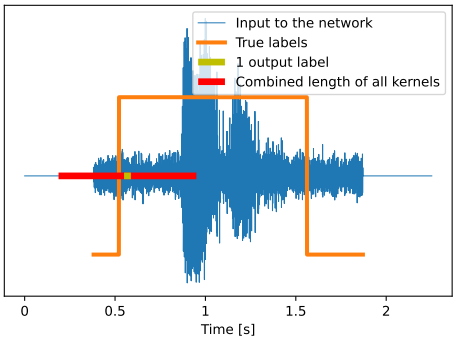


Figure 5.3: Illustration of how much context is needed to generate a VAD output

A very interesting observation is that the length of the zero padding is larger than the length of the silence in the beginning and the end. Thus in practice it is possible that the network learns that if sufficient filter weights are convolved with these zeroes, the VAD is to be labelled as non-speech. To test this theory a series of experiments has been carried out. The idea behind this experiment is:

- *If the network is actually learning the charateristics of speech, it should be able to generalize to an input that has a different structure but still contains speech and noise similar to that of the training set*

To achieve this different structure of the files, a number of randomly chosen audio files is concatenated. Thus the files themselves are unchanged, but the ratio between audio and zero padding is significantly increased. An example of the proposed way to concatenate audio is seen in fig. 5.4.

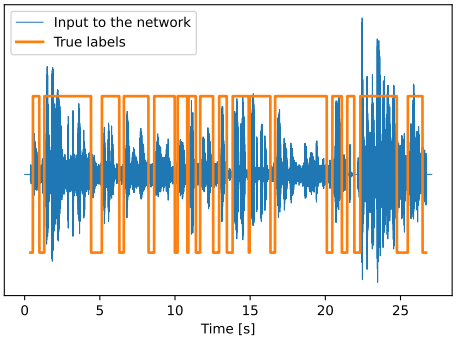


Figure 5.4: 10 sound files randomly chosen and concatenated. 398 ms of zeroes are padded to both the end and the beginning.

A total of four different simulations has been carried out to validate whether the network is actually learning the characteristics of speech. These are:

- The network is trained on single files as seen in fig. 5.3 and tested on similar single files
- The network is trained on single files as seen in fig. 5.3 and tested on concatenated files as in fig. 5.4

- The network is trained on concatenated files as in fig. 5.4 and tested on single files as in fig. 5.3
- The network is trained on concatenated files as in fig. 5.4 and tested on similarly concatenated files

The AUC values are found as the average over the 4 noise types of Aurora-2 test set B. The AUC values are found at each SNR level, and the results are presented in table 5.1.

AUC Aurora-2	Clean	20 dB	15 dB	10 dB	5 dB	0 dB	-5 dB	**Mean**
Trained on single - test on single	98.81	98.61	98.44	98.47	97.96	97.00	95.60	*97.84*
Trained on single - test on concat.	81.60	70.52	67.99	65.92	60.29	56.81	54.70	*65.40*
Trained on concat. - test on single	97.41	97.24	95.34	90.20	82.69	73.30	67.03	*86.17*
Trained on concat. - test on concat.	98.47	98.44	98.22	97.58	94.36	84.90	66.08	*91.15*

Table 5.1: AUC values when training and testing on the Aurora-2 database

It is seen that when the network is trained on single files the performance is way superior when testing on single files as compared to testing on concatenated files. Meanwhile, when trained on concatenated files the difference in performance is way lower. This proves the idea that when training this network on individual files of the Aurora-2 database, the characteristic structure of the files is learned alongside the characteristics of speech. For this reason, the rest of this work will be trained on randomly concatenated files, while testing will be done concatenating files in the same order in every test to minimise the randomness introduced by this.

5.1.3 TIMIT database

So far only the Aurora-2 database has been used in this work. However, to make sure the future results obtained generalise beyond the Aurora-2 database an additional database will be introduced. This will be the (TIMIT) [19] database, which is sampled at 16 kHz and contains full spoken sentences instead of only letters and numbers like Aurora-2. The TIMIT database contains only clean speech, therefore the files have been corrupted with noise by the author as described in appendix A. The noise types used are *street*, *bus*, *cafe* and *pedestrian* from the CHiME-3 [20] database while *babble* and *Speech Shaped Noise (SSN)* are generated by the authors of [21]

The same experiment as that of table 5.1 is repeated for the TIMIT database and the results are shown in table 5.2.

AUC TIMIT	Clean	20 dB	15 dB	10 dB	5 dB	0 dB	-5 dB	**Mean**
Trained on single - test on single	87.51	87.10	86.38	85.23	84.87	84.36	83.96	*85.63*
Trained on single - test on concat.	58.52	58.66	58.71	58.83	58.75	57.70	56.36	*58.22*
Trained on concat. - test on single	94.33	93.97	93.09	90.59	84.39	73.22	59.59	*84.17*
Trained on concat. - test on concat.	95.25	95.38	94.78	93.15	90.22	83.24	72.72	*88.78*

Table 5.2: AUC values when training and testing on the TIMIT database

Once again the same tendency is observed. When the network is trained on concatenated files it generalises well to both single and concatenated files during testing. For the TIMIT dataset the noise types are the same for both the training set and the testing set. However the actual noise segments are unique to the two sets and thus the only overlap between the training and testing is the noise type.

Introducing Adversarial Multi Task Learning 6

In this chapter will be elaborated the proposed method for improving VAD performance by introducing adversarial multi-task learning. The method is shortly described in section 2.2 of the paper, in chapter 4. The aim of this method is to answer the first research question:

- *Can we potentially increase the noise-robustness of a VAD without increasing its computational cost and latency to the execution-time?*

The proposed method is inspired by the following previous works:

- *Adversarial Multi-task Learning of Deep Neural Networks for Robust Speech Recognition* [22]
- *Adversarial Network Bottleneck Features for Noise Robust Speaker Verification* [23]

6.0.1 Adversarial Multi-task Learning of Deep Neural Networks for Robust Speech Recognition

In [22] is proposed a method for increasing the noise robustness of a speech recognition system by introducing adversarial multi-task learning during training. In their work the network is learning from two different targets using two seperate sub-networks. The primary target is the ordinary speech recognition application, and the secondary target is the noise types by which the audio is corrupted. Contrary to ordinary multi-task learning, the secondary task is learned adversarially to the primary task. Thus the goal is to minimise the networks ablility to recognize noise types while maximising the performance of the speech recognition.

In fig. 6.1 the method is illustrated as originally proposed in [22]. In this example the parameters of the three sub-networks are denoted θ_x, θ_y and θ_z which are optimised with respect to the loss functions L_y and L_z:

- Parameter set θ_y is optimised with respect to loss L_y only
- Parameter set θ_z is optimised with respect to loss L_z only
- Parameter set θ_x is optimised with respect to both loss L_y and L_z. However, the gradient with regards to L_z is reversed such that adversarial multi-task learning is achieved

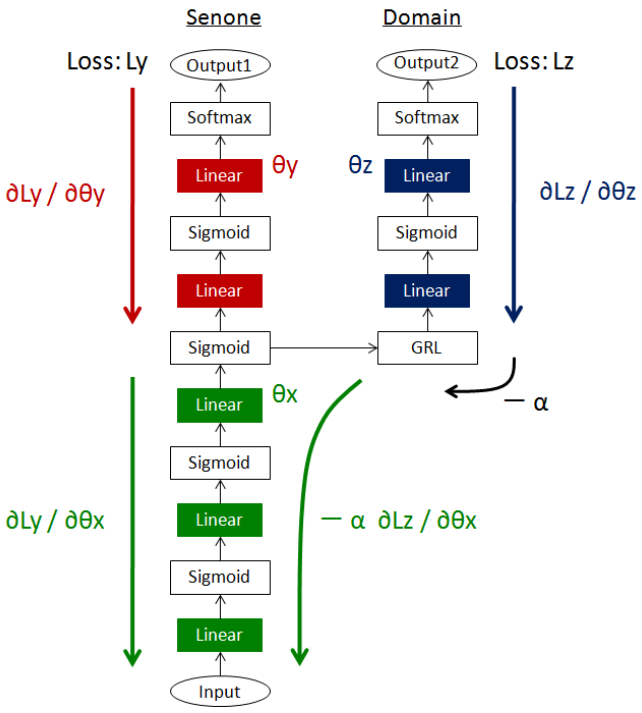


Figure 6.1: Adversarial multi-task learning applied to speech recognition in [22]

6.0.2 Adversarial Network Bottleneck Features for Noise Robust Speaker Verification

In [23] a similar idea has been applied to a speaker verification task. In this work however, the additional sub-network for classifying noise types is cascaded after the original network. The original network and the discriminative network are then trained in turns. The same loss function is shared, however the target changes depending on which part is currently under training. While training the original network all the speech is labelled as clean regardless of its actual noise type, and when training the discriminative network the actual noise types are used as labels. This approach is similar to that of a Generative adversarial network (GAN).

6.1 Relating to this work

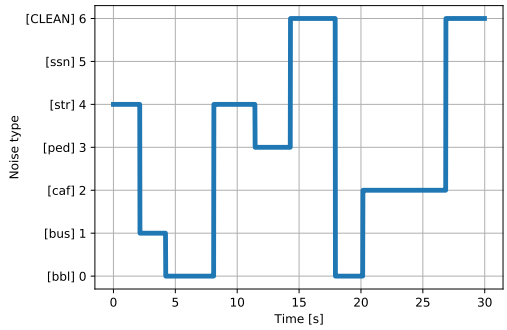
In both of these cases the discriminative part of the network is discarded after training, and thus no additional cost is introduced to the forward step. Because both approaches have shown great results in different speech applications, it is of interest to find if a similar approach can be used to enhance the performance of a VAD.

Both of these works are considering only feed forward neural networks, and therefore some slight modifications will have to be done before it can be applied to the network of fig. 3.2 which is a FCN. First will be considered an approach similar to that of fig. 6.1 where two things are important to consider:

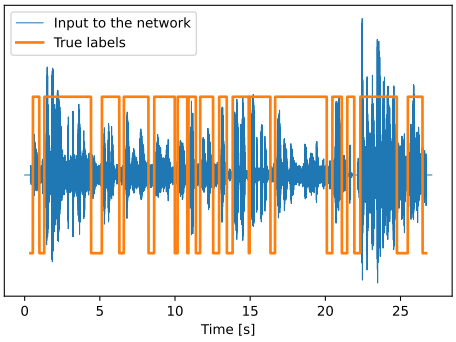
- Where do we split the original network into two sub-networks, such that the output o the first sub-network can be used as input to the discriminative network?
- What will be the structure of the discriminative network?

The original network can be split into two sub-networks at two places: Between the EB and the FB or between the FB and the DB. As we are interested in making the VAD frames on a 10 ms basis invariant to noise, it has been chosen to split the network between the FB and the DB. Additionally, this also results in lower computational cost during training as the size of the feature map is already downscaled significantly by the FB.

Now that the input to the discriminative network has been decided upon, let us move on to finding the structure of the discriminative network. As presented in chapter 5, during training files are randomly chosen and concatenated to keep the network from learning the structure of the audio files instead of the speech characteristics. Because of this different noise types may be present in the same audio segment of concatenated files. Thus while training on the TIMIT database the labels for the discriminative network may look like fig. 6.2a while the corresponding VAD labels may look like fig. 6.2b. Due to the problem being a multi-class classification task the cross entropy loss function as presented in section 2.2.3.1 will be used as loss function.



(a) Example of truth labels for noise types for loss **Lz**



(b) Example of waveform with truth VAD labels for loss **Ly**

As the noise types is varying in time similarly to the VAD labels, it is desired to introduce a sub-network with the same time-invariant capabilities as the rest of the network, i.e. a convolutional network. An obvious solution is to replicate the DB and use as the discriminative network. Applying all this results in the new network as seen in fig. 6.3

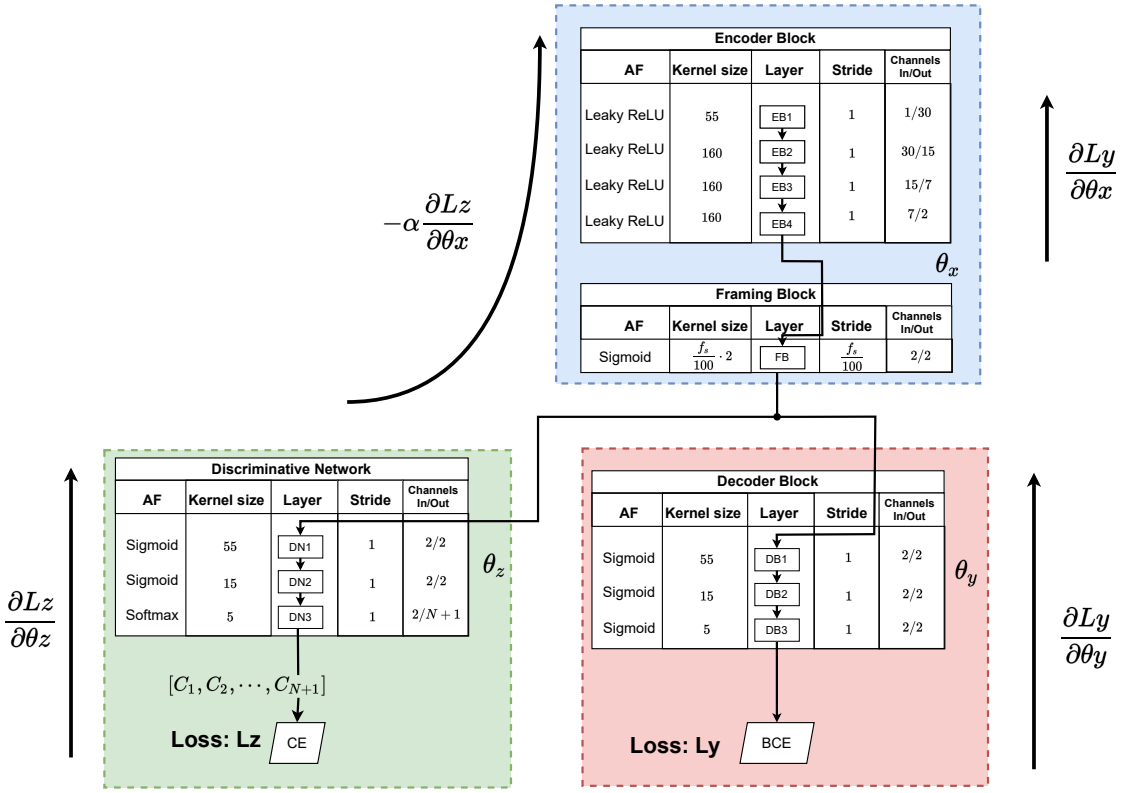


Figure 6.3: The network for VAD after introducing adversarial multi-task learning

6.2 Experiments

Having estabilised the structure of the network utilising adversarial multi-task learning, this section will contain a series of experiments as described in section **4.1** of the paper. First we will find the optimal weight factor α, which determines the contribution of the discriminative network to the EB.

For finding the optimal value of α, a series of experiments has been carried out with α swept over the values $10^n n \quad \in [-2, -1, 0, 1, 2]$. In these experiments the network has been trained from scratch and tested after 20 epochs as was decided upon in section 3.2.1.2 To limit the randomness in these experiments, the same initial parameter values (see eq. (2.30) and eq. (2.28)) has been used in all experiments. The sweep has been made on both the Aurora-2 and the TIMIT database. The results are presented in table 6.1. In this table the performance is found as the average AUC over both the noise types and the SNR levels.

Table 6.1: AUC values on the validation sets using different values of α

α	**0**	**0.01**	**0.1**	**1**	**10**	**100**
Aurora-2 A	93.90	93.93	94.44	94.58	**95.18**	92.97
Aurora-2 B	91.15	91.22	**92.10**	91.99	91.85	90.99
TIMIT	88.78	89.98	**90.32**	89.3	88.91	87.94

From this it is seen that for a wide range of α values the performance is enhanced compared to the reference with no adversarial multi-task learning (denoted by $\alpha = 0$). On both Aurora-2 test set B and the TIMIT test set the optimal value of α is found to be 0.1, whereas the optimal value for Aurora-2 test set A is found to be 10. Furthermore it is seen that as the value of α converges towards 0, the performance gets closer to that of the reference, and on the other hand when the value gets too high, the performance decreases below the reference. As the most promising value of α is found to be 0.1, the further simulations will be trained using this.

Having found the performance is increased by introducing adversarial multi-task learning, it is of interest to investigate this more thoroughly. In table 6.2 is found the performance at each individual SNR level averaged over the noise types of Aurora-2 test set A and B and TIMIT

Table 6.2: AUC values on the test sets of AURORA-2 and TIMIT with and without adversarial multi-task learning

	Clean	20 dB	15 dB	10 dB	5 dB	0 dB	-5 dB	**Mean**
Aurora-2 A - $\alpha = 0$	**98.46**	**98.46**	**98.37**	98.19	97.05	91.19	75.58	*93.90*
Aurora-2 A - $\alpha = 0.1$	98.42	98.40	98.32	98.19	**97.19**	**92.81**	**77.78**	***94.44***
Aurora-2 B - $\alpha = 0$	**98.46**	**98.44**	**98.22**	**97.58**	94.36	84.90	66.08	*91.15*
Aurora-2 B - $\alpha = 0.1$	98.42	98.37	98.13	97.39	**94.67**	**87.51**	**70.24**	***92.11***
TIMIT - $\alpha = 0$	95.44	95.41	94.63	92.64	88.60	82.67	71.96	*88.76*
TIMIT - $\alpha = 0.1$	**95.62**	**95.49**	**94.92**	**93.91**	**90.75**	**84.94**	**74.26**	***90.32***

It is seen that the performance increase is mainly found at the lower SNR levels, while the performance at higher SNR levels is approximately unchanged. To further illustrate this the results for the Aurora-2 test set B and TIMIT are visualised in fig. 6.4:

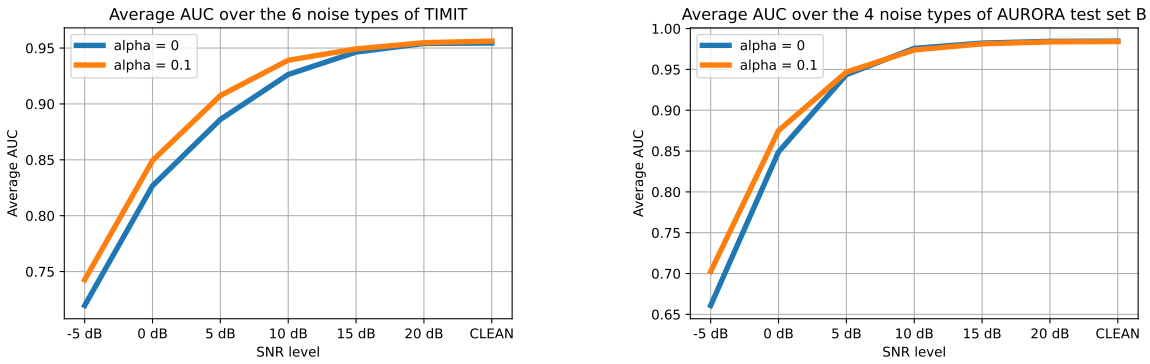


Figure 6.4: AUC on the TIMIT and Aurora-2 databases found at each SNR level

Thus it is found that the performance of the VAD has been increased without adding further cost during the forward step after training is finished and thus answering research question 1.

6.3 Additional work done after paper submission

After submitting the paper to Interspeech 2022 an additional experiment has been carried out. The aim of this experiment is to test if the performance can be improved further by instead of concatenating random noise types as seen in fig. 6.2a during training, only similar noise types and SNR levels (similar to how testing was already carried out) are concatenated for a more realistic setting.

The experiment is carried out with similar settings as earlier, such that the only change is the way in which files are concatenated. The same parameter initialisation as used previosuly is used in this experiment. These experiments are only made on the Aurora-2 test sets.

In table 6.3 the experiment for finding the optimal value of α has been repeated concatenating only files of similar noise type and SNR level.

Table 6.3: AUC values on the validation sets using different values of α

α	**0**	**0.01**	**0.1**	**1**	**10**	**100**
Aurora-2 A	94.08	94.15	**95.18**	94.74	94.30	94.24
Aurora-2 B	88.64	91.16	92.49	**92.69**	91.13	90.38

Similar to the results of table 6.1 the best performance is observed at different α values for the different test sets. On test set A $\alpha = 0.1$ is found to be optimal while $\alpha = 1$ is found optimal for test set B. Once again, at a wide range of α a performance increase is observed. In order to stay true to the work published in the paper, the following comparisons will be made using $\alpha = 0.1$.

The next experiment is similar to that of table 6.2 and the results can be seen in table 6.4. The goal is to find at what SNR levels the performance increase is observed by introducing adversarial multi-task learning.

Table 6.4: AUC values on the test sets of Aurora-2 and TIMIT with and without adversarial multi-task learning

	Clean	20 dB	15 dB	10 dB	5 dB	0 dB	-5 dB	**Mean**
Aurora-2 A - $\alpha = 0$	**97.96**	**97.95**	97.85	97.62	96.59	92.12	78.79	*94.08*
Aurora-2 A - $\alpha = 0.1$	97.91	97.90	97.88	**97.66**	**96.94**	**93.44**	**84.49**	***95.18***
Aurora-2 B - $\alpha = 0$	**97.96**	**97.93**	97.10	95.05	89.56	79.01	63.79	*88.64*
Aurora-2 B - $\alpha = 0.1$	97.91	97.90	**97.54**	**96.54**	**94.20**	**89.11**	**74.25**	***92.49***

Similar to table 6.2 the largest increase of performance is found at the lowest SNR levels. In fact the performance increase at the low SNR levels is even greater when trained on concatenated files of similar type instead of random types (6-10% instead of 2-4%).

As a last experiment is found the performance using randomly concatenated files against using concatenated files of similar type during training. These results are presented in table 6.5

Table 6.5: AUC values on the test sets of AURORA-2 and TIMIT with and without adversarial multi-task learning

	Clean	20 dB	15 dB	10 dB	5 dB	0 dB	-5 dB	**Mean**
Aurora-2 A - random noise	**98.42**	**98.40**	**98.32**	**98.19**	**97.19**	92.81	77.78	*94.44*
Aurora-2 A - similar noise	97.91	97.90	97.88	97.66	96.94	**93.44**	**84.49**	***95.18***
Aurora-2 B - random noise	**98.42**	**98.37**	**98.13**	**97.39**	**94.67**	87.51	70.24	*92.11*
Aurora-2 B - similar noise	97.91	97.90	97.54	96.54	94.20	**89.11**	**74.25**	***92.49***

Here it is seen that in fact the performance has been further improved by concatenating only audio files of similar noise type and SNR level during training. Especially at lower SNR levels where a performance increase of 4-7% in terms of AUC is observed.

6.4 Conclusion on research question 1

As mentioned in chapter 6, the purpose of the work in this chapter was to find an answer to research question 1:

- *Can we potentially increase the noise-robustness of a VAD without increasing its computational cost and latency to the execution-time?*

In section 6.0.1 and section 6.0.2 was conducted a small literature study and two ways of introducing adversarial multi-task learning to speech processing tasks was described. It was decided to use the approach described in section 6.0.1 as inspiration for the method proposed in this work, which can be seen in fig. 6.3. In section 6.2 an important hyperparameter was tuned, and during this tuning the performance increase achieved by introducing adversarial multi-task learning was found.

Across 3 different test sets it was found that the performance was increased by 0.5-1.5% AUC averaged over every noise type and SNR level. The performance was significantly increased at lower SNR levels, while the performance remains unchanged at higher SNR levels. At -5 dB SNR the performance was increased by up towards 4% AUC.

Thus it was found that by introducing adversarial multi-task learning to the network, the performance can be increased without introducing any additional cost once training is finished and research question 1 is successfully answered.

After the paper was submitted to Interspeech 2022 it was found that the performance can be increased further by only concatenating files of similar noise type and SNR level during training.

Investigating algorithmic delay

7

Having successfully answered research question 1 in chapter 6, this chapter will focus on research question 2:

- *How will it affect the performance of the VAD if we allow it to use less future samples to generate a VAD output and thus decrease the algorithmic delay?*

This research question is more of a feasibility study than it is a question about finding the best solution. It is expected that with a shorter algorithmic delay will follow a decreased performance. For this reason it will be investigated how well the network is performing at different algorithmic delays. The theory presented in this chapter is an extension of section **2.3** of the paper, while the experiments will be an extension of section **4.2**.

As it has already been found that the network performance was increased by introducing adversarial multi-task learning, this work will build on top of that. First will be investigated how much algorithmic delay is introduced by the network presented in fig. 6.3. This analysis will focus on one block at a time (i.e. EB, FB, DB), and afterwards it will be investigated how the algorithmic delay can be reduced.

7.0.1 Finding the algorithmic delay

Conveniently, a large part of the work towards finding the algorithmic delay of the network has already been investigated under the discussion of datasets in chapter 5. In this the goal was to find how much context is needed to generate a VAD output. This context spans both past and future samples, and thus the algorithmic delay can be found by dividing the size of the context found in eq. (5.7) by 2 (such that past samples are disregarded).

The algorithmic delay introduced by the network is then:

$$\text{Algorithmic delay in seconds} = \frac{\sum_{n=1}^{4}(k_{EBn}-1) + \frac{f_s}{100} + \sum_{i=1}^{3}(k_{DBi}-1)\cdot\frac{f_s}{100}}{2f_s} \tag{7.1}$$

Inserting the filter sizes of fig. 6.3 the algorithmic delay is found to be:

$$\text{Algorithmic delay in seconds} = 0.398 \tag{7.2}$$

7.0.2 Reducing the algorithmic delay

As the goal of this work is to reduce the algorithmic delay and find how the performance reacts to this, it will now be investigated how much algorithmic delay is introduced by each block. In chapter 5 the context introduced in each layer was found and combined to form eq. (5.7). Similarly to finding the combined algorithmic delay, the delay introduced by each block can be found by dividing the total context by 2:

$$\text{Delay of EB} = \frac{\sum_{n=1}^{4}(k_{EBn} - 1)}{2fs} \approx 0.018s \tag{7.3}$$

$$\text{Delay of FB} = \frac{\frac{fs}{100}}{2fs} \approx 0.005s \tag{7.4}$$

$$\text{Delay of DB} = \frac{\sum_{i=1}^{3}(k_{DBi} - 1) \cdot \frac{f_s}{100}}{2fs} \approx 0.375s \tag{7.5}$$

Because the FB introduce a stride of of $\frac{fs}{100}$, the majority of the algorithmic delay is induced by the DB. For this reason only the DB will be considered in this work on algorithmic delay. From eq. (7.5) it is seen that the algorithmic delay of the DB is directly proportional to the filter sizes. Therefore, by shrinking the filter sizes the algorithmic delay will decrease as well.

7.1 Experiments

In order to investigate the performance under reduced algorithmic delay, a series of experiments using varying filter sizes have been carried out. Unlike earlier experiments, all of the models are trained using the same initial parameters in only the EB and FB, while DB is initialised randomly in every experiment due to the varying filter sizes.

The filter sizes and their corresponding algorithmic delay and performance averaged over every noise type and SNR level on the Aurora-2 test set A and B are listed in table 7.1. In fig. 7.1 the performance at each SNR level averaged over the 4 noise types of Aurora-2 test set B is plotted

DB1	DB2	DB3	AD [ms]	Aurora-2 B	Aurora-2 A
55	15	5	*398*	92.11	94.44
45	15	5	*348*	91.91	94.38
35	15	5	*298*	90.81	93.44
25	15	5	*248*	91.04	93.32
15	10	5	*173*	90.95	93.01
10	7	5	*133*	88.09	90.84
7	5	5	*98*	89.01	91.85
5	3	3	*78*	87.14	89.78
3	3	2	*63*	85.77	90.00
2	2	2	*53*	85.26	87.56
2	2	0	*43*	85.03	87.40
2	0	0	*33*	85.66	87.90
0	0	0	*23*	85.07	87.27

Table 7.1: VAD performance in terms of AUC at different kernel sizes and algorithmic delays tested on Aurora-2 test sets A and B

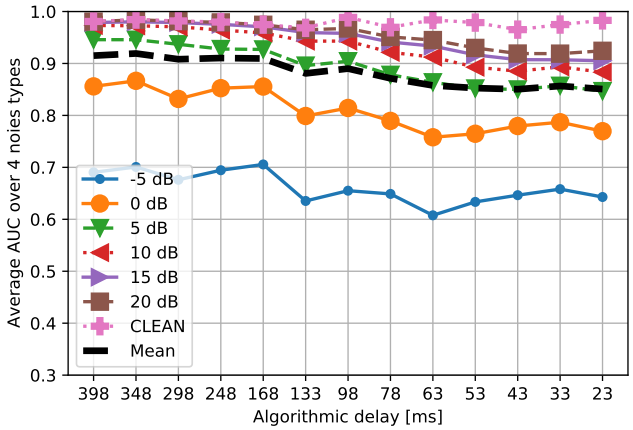


Figure 7.1: The VAD performance at decreasing algorithmic delays at each SNR level

The filter sizes is decreased in small steps until the DB is completely disregarded. At this point the algorithmic delay is lowered to 23 ms. It is seen that the performance is steadily decreased as the algorithmic delay is lowered, however the degradation is not significant. When decreasing the algorithmic delay from 398 ms to 23 ms the performance is only reduced by 7% AUC. It is interesting to note that in the case of clean speech the performance seems to be unaffected by the lower algorithmic delay, whereas every other SNR level seems to be following the same pattern in terms of performance decrease.

7.2 Conclusion on research question 2

This chapter has been focusing on research question 2:

- *How will it affect the performance of the VAD if we allow it to use less future samples to generate a VAD output and thus decrease the algorithmic delay?*

First the algorithmic delay of the network presented in fig. 6.3 was calculated to be used as a reference. Most of the work towards computing this delay was already done in chapter 5 when discussing how to use the dataset. It was found that the majority of the algorithmic delay of the network is introduced by the DB, and following this the filter sizes of this block was reduced in small steps. The results are listed in table 7.1 where it was found that performance decrease is significantly lower than the corresponding reduction in algorithmic delay. In fact it was possible to reduce the algorithmic delay from 398 ms to 23 with only 7% reduction in AUC.

Part III

Implementation considerations

Introduction 8

So far this work has focused on algorithm selection, analysis and investigation of methods to increase the noise robustness and reduce the algorithmic delay introduced by the network in order to answer research questions 1 and 2. However, if the VAD is ever to be implemented in a real-world scenario it is a neccessity that it operates in real-time. This next part of the work will focus on this aspect and thereby answer research question 3. First will be discussed what the goal is, and based on this discussion some key metrics which to optimize for will be decided.

8.1 Motivation

In this section will be discussed what the aim of this part of the work is. So far the term "real-time" has been used loosely as the aim at which the VAD is supposed to operate. In a typical application of a VAD is as a pre-processing step before more complex audio signal processing tasks such as speech recognition, speaker verification or speech enhancement. A typical speech recognition task is wake-word or hot-word detection, speaker verification is used to verify the identity of a speaker, while speech enhancement is typically used to enhance the intelligibility of a noise corrupted signal. These are all applications which requires low latency in order to provide the best user experience. Furthermore, given that the VAD is often used in conjunction with more computationally expensive signal processing algorithms, it is important that the latency and computational cost introduced by the VAD is minimal.

Some of these applications may be implemented on small devices like Intercom systems, headsets or hearing aids. These devices are limited in both computational power, power consumption and physical area available. So far all of these metrics have been disregarded in this work, and due to the limited timeframe and scope of this project, not all of the metrics will be considered equally. Four important metrics when considering an embedded system is *execution time, physical area, power consumption* and *precision* (or *noise* when minimising)[24]. These four metrics are all dependent on each other and optimising for one will potentially lead to degradation for other metrics. From these four metrics a cost function can be established:

$$C = f(A,T,P,N) = a_i \cdot A + a_2 \cdot T + a_3 \cdot P + a_4 \cdot N \tag{8.1}$$

We are interested in finding the methods that leads to the best trade-off between the metrics described by eq. (8.1). To do this we introduce the *Gajski-Kuhn Y-chart*, which is mainly used for development of integrated circuits:

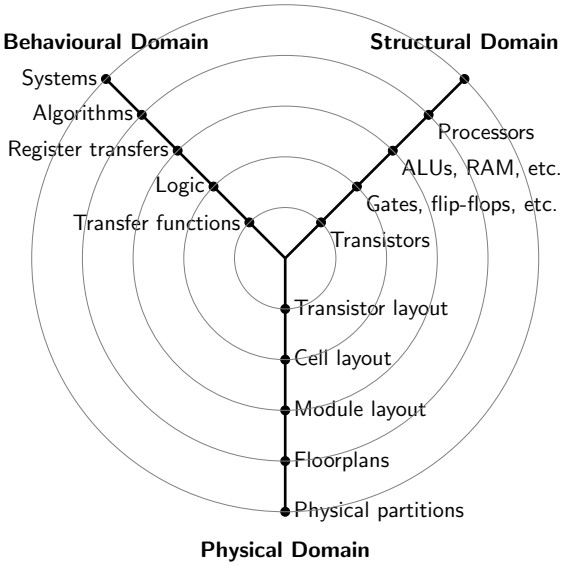


Figure 8.1: Gajski-Kuhn Y-chart

The Y-chart depicts different important perspectives to consider when integrating an algorithm onto a piece of hardware. According to the model, the hardware design is perceived within three *domains*, and within these domains five different abstraction levels are defined.

In this project the main focus will be on the behavioural domain, while some considerations will be based on the structural domain. Due to the scope and time frame of this project the physical domain is disregarded. Furthermore, as this will be a feasibility study on the whether the VAD part proposed algorithm as shown in fig. 6.3 is suitable for a real-time application, only the higher abstraction levels will be considered.

8.2 Survey on metrics

Having outlined the need for investigating the metrics related to algorithm execution in small embedded devices, this section will provide a short description of each metric and end up choosing which are most relevant for this work.

8.2.1 Area

The amount of hardware available on an embedded device is an important factor to consider when implementing algorithms. In the case of a CNN, first recall that the operations involved are denoted as:

$$y_i^{(l)} = \sum_{j=1}^{N} w_j^{(l)} y_j^{(l-1)} + b^{(l)} \tag{8.2}$$

where

$y_i^{(l)}$ is a single feature for the layer l.
$w_j^{(l)}$ is the j^{th} weight of the kernel
$y_j^{(l-1)}$ is a single feature from the previous layer $l-1$
$b_i^{(l)}$ is the bias of the l^{th} layer

Thus, the only operations taking place in a convolutional layer are multiplications and additions. For this reason it will be assumed that the computations will be taking place on Multiply ACcumulate (MAC) units, which are hardware units that perform one multiplication and accumulates the result through an addition. Given that the calculation of features within a layer is independent of each other, a large amount of parallelism can be achieved and thus the execution time can be increased by introducing more MAC units [25]. In return this obviously increases the demand for hardware and also increases the physical area required.

Additionally, the more memory is needed, the more Random Access Memory (RAM) is required. In the case of a FCN, the kernel weights, the biases and the intermediate features are saved in the RAM[25].

8.2.2 Execution time

As briefly mentioned above, the execution time is highly dependent on the available hardware. More hardware results in better utilisation of the parallelism in the system, and thus more computations can be carried out simultaneously. Other factor that play a role in the execution time is the number representation used, i.e. if the calculations are carried out as floating point or fixed point, as well as the number of bits used to represent a digit.

8.2.3 Power consumption

The power consumption of an embedded system is dependent on a variety of factors. Worth noting is the amount of memory reads, the amount of computations to be carried out, the clock frequency at which the system is operating and the number representation and bit width[26]. Additionally, the power consumption is affected by low-level factors such as instruction scheduling, adressing modes and even the size of the transistors[26]. These factors are mostly out of scope of this project, and therefore power consumption will not be considered in depth.

8.2.4 Precision

Precision is the last metric to be considered, and is also the only one which has already been considered and optimised in this project. As with the other metrics, optimising the precision can leave to a degradation in area, execution time and power consumption and vice versa.

8.2.5 Choice of metrics

Having outlined the four main metrics for designing an embedded system, it is clear that within the time frame of this project not all of them can be considered equally. As already mentioned the power consumption will mostly be disregarded, leaving time to work on the

aspects of area and execution time. In the following sections some relevant literature will be presented for methods on optimising these two metrics.

8.3 Survey on methods - litterature overview

As mentioned in section 8.2.2 and section 8.2.1, both the execution time and physical area needed are dependent on the number representation used, as well as the number of parameters and computations to be carried out. Therefore the presented literature will focus on these aspects while being methods optimising on high abstraction levels.

8.3.1 Pruning

First let us consider the method for reducing the model size proposed in [27]. In their work on some well-known network models like *VGG-16* and *AlexNet* they have managed to reduce the model size by 90% by pruning the network. Their approach for pruning a feed-forward neural network is illustrated in fig. 8.2:

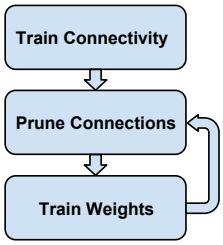


Figure 8.2: The proposed approach to pruning [27]

Their approach is a 3-step process, which in a feed-forward neural network first trains the connection between neurons, then prunes x% of the smallest connections by setting them equal to zero, and lastly re-training the weights using the new updated connections. This is an iterative approach that can be repeated until satisfactory performance and network size is found. The pruned connections are not being retrained and thus remain zero. In order to apply the method on a CNN, in which weights and biases are the only parameters, the "Train connectivity" step is disregarded and instead the pruning connections weights are being pruned

8.3.1.1 Regularization

Because all weights below a threshold are being pruned, it is important that only the least important weights are below this threshold. In ensuring this a penalty term is added to the loss function. In [27] it was found that using L2-regularization yields the best performance under retraining.

$$L2 = \lambda \sum_{j=0}^{p} \theta_j^2 \tag{8.3}$$

As seen above L2 regularisation adds a penalty term that aims to minimise the squared magnitude of the parameters.

8.3.2 Quantization

The second approach to be considered in this work is quantization of the network. Quantization is the process of mapping values from high-precision number representation to one with lower precision, and thus fewer bits. Let us first consider floating-point representation and fixed-point representation.

8.3.2.1 Floating point representation

So far the simulations made in this project has been carried out in the default datatype of PyTorch, which is a 32-bit floating point value using 1 sign bit, 8 exponent bits and 23 mantissa bits also referred to as *single precision.* The floating point operations are handled accordingly to the *IEEE-754* standard, which is illustrated in fig. 8.3. In IEEE-754 the first bit of the mantissa is implicitly =1, such that the effective length of the mantissa is 24 bits [28, p. 169-188]. The following section on floating point representation is based on [28, p. 169-188].

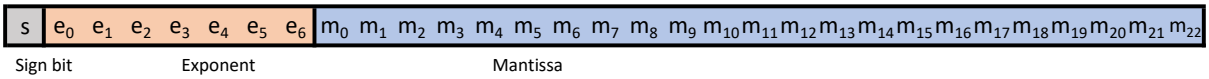


Figure 8.3: Representation of a single precision floating point number as in IEEE-754

In floating point the value of a number is represented by:

$$x \Longrightarrow \tilde{x} = (-1)^s \cdot m \cdot b^e \tag{8.4}$$

where:

s = the sign bit
e = the exponent expressed as an integer $\sum_i^N e_i \cdot b^{N-1}$
m = the mantissa expressed as a scalar $\sum_i^N m_i \cdot b^{-i}$
b = the base/radix used. In this case it will be 2
N = the number of exponent or mantissa bits

Considering eq. (8.4) it is obvious that not an infinite number of values can be represented, therefore floating point is considered *finite precision*. This means that an error will be introduced when converting a continuous signal into finite precision. This process of mapping a continuous infinite value into a discrete value is known as quantization. The relative error introduced is defined as follows:

$$error = \left| \frac{x - \tilde{x}}{x} \right| : error \in \left[-2^{e-p}; 2^{e-p} \right] \tag{8.5}$$

where

x = the continuous infinite value
$\tilde{x}$ = the discrete finite approximation
p = the number of bits of the mantissa (24 in IEEE-754)

By finding the maximum of the numerator and the minimum of the denominator we can find the maximum error. In the case of fixed e we have:

$$\max |x - \tilde{x}| = 2^{e-p} \tag{8.6}$$
$$\min |x| = 1 \cdot 2^0 \cdot 2^e - \kappa \tag{8.7}$$

By inserting into eq. (8.5) we get an expression for the largest relative error. Also referred to as the *machine precision*:

$$\epsilon = \max \left| \frac{x - \tilde{x}}{x} \right| = \frac{2^{e-p}}{2^e \pm 2^{e-p}} = \frac{2^{-p}}{1 \pm 2^{-p}} \simeq 2^{-p} \tag{8.8}$$

Thus it is found that larger p provides lower quantization errors and thus higher accuracy. This property will be the foundation for the next work in this project.

8.3.2.2 Floating point arithmetic

As mentioned in section 8.2.1, a FCN consist only of additions and multiplications when disregarding the activation functions. For this reason it is interesting to also consider the complexity of these operations using different number representations.

First let us consider floating point additions.

Floating point addition

The process of adding two floating point numbers is a process in several steps:

- First the exponents are aligned, such that the smaller number is written using the same exponent as the larger number. For example consider the addition $1 \cdot 2^1 + 1.25 \cdot 2^5$, then we can align the exponents by rewriting as $0.0625 \cdot 2^5 + 1.25 \cdot 2^5$
- Then we add the mantissas. Continuing the example we get $(0.0625 + 1.25) \cdot 2^5 = 1.3125 \cdot 2^5$
- Sometimes it is necessary with a third step to re-normalize the numbers, i.e. such that the mantissa remains in the range of $[1; 2]$ when using radix-2.

Floating point multiplication

Similarly to addition, floating point multiplications involves several steps. The example used below will be similar to the one used in the addition: $1 \cdot 2^1 \times 1.25 \cdot 2^5$

- First the exponents are added: $2^1 + 2^5 = 2.024 \cdot 2^5$
- Secondly the mantissas are multiplied: $1 \cdot 1.25 = 1$
- Lastly the mantissa is normalized, and the exponent is adjusted accordingly: $2.024 \cdot 2^5 = 1.012 \cdot 2^6$

8.3.2.3 Fixed point representation

Now let us consider a different number system, *fixed point*, which will ease the computational requirements for performing additions and multiplications. A fixed-point number consists of an integer part, a fractional part and a sign bit (the sign bit is omitted in the case of unsigned numbers). The name *fixed-point* arise from the fact that a fixed number of bits are reserved for the integer part and the fractional part, as well as these bits always representing one and only one value. In fig. 8.4 is illustrated an 8 bit fixed point number consisting of 1 sign bit, 4 integer bits and 3 fractional bits, where also the value represented by each bit is shown. The number format is *signed magnitude*.

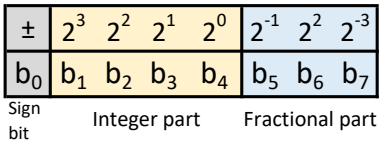


Figure 8.4: An example of a signed magnitude fixed-point number

Each bit in signed magnitude representation is a value of 2^n, where n is an integer, such that the numerical value of a signed magnitude number is:

$$x = (-1)^{x_0} \cdot \sum_{i=-M}^{N-1} x_i \cdot 2^{-i} \tag{8.9}$$

Similar to the floating point representation, a finite number of bits results in a finite number of potential values. In fixed-point representation the interval between representable values is called the *quantization step*, which will be denoted by: $\Delta = 2^{-(N-1)}$, where N is the number of fractional bits. Just like in floating point representation, this means that a larger number of fractional bits will result in a smaller quantization step and thus higher accuracy.

Having found Δ we can now define the dynamic range as:

$$\text{Dynamic range} = \left[-2^M + \Delta; 2^M - \Delta\right] \tag{8.10}$$

where M is the number of integer bits.

However, with signed magnitude follows two problems:

- We have a representation for both -0 and +0
- Arithmetic operations cannot be carried out directly

2's complement

Because the main point of introducing fixed point numbers is to ease the arithmetic operations, there is no point in looking any further on the signed magnitude representation. Instead we consider *2's complement* notation which has only one representation of 0, and in which arithmetic operations can be carried out directly, as shown in the example in eq. (8.11).

$$\begin{array}{rrl} 0000 & 1111 & (15) \\ +\ 1111 & 0101 & (-11) \\ ========== & & \\ 0000 & 0100 & (4) \end{array} \tag{8.11}$$

Positive numbers are represented in the same way as in signed magnitude, however for negative values we invert all the integer and fractional bits, and add 1 to the Least Significant Bit (LSB).

Using 2's complement results in the same quantization step as for signed magnitude, however adding 1 to the LSB for negative values acts as a bias in the value represented. This in combination with inverting all bits after the sign in case of a negative results in values being represented as:

$$x = -x_0(1-\Delta) + \sum_{i=-M}^{N-1} x_i \cdot 2^{-i} - x_0 \cdot \Delta \tag{8.12}$$

And the dynamic range is slightly changed now that we only have 1 way to represent 0:

$$\text{Dynamic range} = \left[-2^M; 2^M - \Delta\right] \tag{8.13}$$

Additionally, multiplications can be carried out efficiently in 2's complement using methods such as *shift and add multiplications* or *Booths algorithm* [29]. However, as this project is only focusing on optimizations on higher abstraction levels, we will not consider these multiplication algorithms any further.

8.3.3 Quantization - continued

Having introduced two widely used number formats, IEEE-754 floating point and 2's complement fixed-point, we will in this section discuss two ways of performing the quantization. It has already been established that the fixed-point arithmetic is more simple than floating point arithmetic. In fact:

- an 8-bit fixed point addition requires 3.8× less area and 3.3× less energy than a 32-bit fixed point addition. Compared to a 32-bit floating point, the 8-bit fixed point addition requires 116× less area and 30× less energy [25] [30].
- an 8-bit fixed point multiplication requires 12.4× less area and 15.5× less energy than a 32-bit fixed point multiplication. The 32-bit floating point multiplication requires an additional 27.5× more area and consumes 18.5× more energy than the 8-bit fixed point [25] [30].

Due to the fact that additions and multiplications are computationally cheaper, requires less area and consume less power when carried out using fixed-point arithmetic, floating point numbers will not be considered any further. Additionally, it is seen that lowering the bit width of the fixed point numbers can be greatly beneficial.

In the following we consider two different approaches to quantization. The first approach is the "naive quantization", where we simply choose one fixed bit width to be used through

every part of the network. Afterwards we consider the method proposed in [31] where the different layers operate with different bit widths.

8.3.3.1 Naive quantization

In the naive approach to quantization we use the same bit width in every part of the network. Therefore this bit width will be used to represent both the feature maps, the weights, the biases and the activation functions.

Every time a value is quantized, or an operation is performed using quantized numbers, some additional *quantization noise* is added to the output:

$$\tilde{x} = x + \text{noise}_{\text{quantization}} = x + n_x \tag{8.14}$$

This error introduced by quantization noise will grow through the layers of the network, thus deeper networks are more prone to quantization noise [31]. The ratio between quantization noise and the true signal is called Signal to Quantization Noise Ratio (SQNR) and is defined as:

$$\text{SQNR}_{\text{DB}} = 10\log_{10}\frac{\mathbb{E}[x^2]}{\mathbb{E}[n_x^2]} \tag{8.15}$$

The lower the SQNR, the higher the performance of the network can be achieved.

8.3.3.2 Cross-layer optimized quantization

Next we consider the method for efficient quantization presented in [31]. The main idea is to optimize the bit widths in each layer with regards to the SQNR. The quantization noise introduced is an unknown quantity, instead we estimate it:

$$\text{SQNR}_{\text{DB}} = \kappa \cdot \beta \tag{8.16}$$

where

κ = the quantization efficiency
β = the bit width

In their work an important observation is that the SQNR introduced by a convolutional layer can be described as:

$$\frac{1}{\text{SQNR}_{w^{[l+1]} \cdot x^{[l]}}} = \frac{1}{\text{SQNR}_{w^{[l+1]}}} + \frac{1}{\text{SQNR}_{x^{[l]}}} \tag{8.17}$$

where

$x^{[l]}$ = the feature map of the l'th layer
$w^{[l+1]}$ = the weights of the (l+1)'th layer

And additionally, the combined SQNR introduced through multiple layers can be described as:

$$\frac{1}{\text{SQNR}_{\text{output}}} = \frac{1}{\text{SQNR}_{w^{[1]} \cdot x^{[0]}}} + \frac{1}{\text{SQNR}_{w^{[2]} \cdot x^{[1]}}} \cdots \frac{1}{\text{SQNR}_{w^{[L]} \cdot x^{[L-1]}}} + \frac{1}{\text{SQNR}_{x^{[l]}}} \tag{8.18}$$

Now that we have a definition of the SQNR introduced through a network, we can consider some key points that will be important for selecting the optimal bit widths:

- Every quantization step contributes equally, i.e. none of them are weighted
- The network performance will be bottlenecked by the quantization step providing the largest SQNR
- Doubling the network depth results in 3 dB decrease of SQNR.

From eq. (8.18) we see that we can obtain the same total SQNR in infinitely many ways, for example by using smaller bit widths in some layers and larger bit widths in other layers. From this we can write an optimization problem which aims to minimize the model size while still maintaining a fixed SQNR. To keep the focus of this project on the main ideas and applications of this quantization strategy, the derivations of the optimization problem will not be described.

The problem is formulated as:

$$\begin{aligned} \min_{\lambda_i} \quad & -\sum_i \rho_i \log_{10} \lambda_i \\ \text{s.t.} \quad & \sum_i \lambda_i \leq \frac{1}{\text{SQNR}_{\text{min}}} \end{aligned} \tag{8.19}$$

where:

$\rho_i =$ the number of parameters being quantized in the i'th quantization step.
$\lambda_i = \frac{1}{SQNR_i}$, the SQNR introduced in the i'th quantization step.

This is a convex optimization problem with the *water-filling* solution [32]. Introducing Lagrange multipliers and solving for the Karush Kuhn Tucker (KKT) conditions we get the following expression for the optimal bit widths in two layers:

$$\beta_i - \beta_j = \frac{10 \log_{10}(\frac{\rho_j}{\rho_i})}{\kappa} \tag{8.20}$$

Thus the difference between the optimal bit width of two quantization steps is inversely proportional to the difference of parameters to be quantized in dB. This is then scaled by the quantization efficiency. This means that when fixing the bit width in any quantization step, the minimum bit widths of every other quantization step satisfying the SQNR constraint can be found.

Experiments 9

While the previous chapter introduced different methods for minimising the size of the network and easing the computations involved, this chapter will implement these methods on the model presented in fig. 6.3.

We will continue the work towards developing a low-latency VAD, by further optimising the network in which the DB was disregarded resulting in the lowest algorithmic delay of 28 ms. However, since this had a negative impact on the VAD performance as compared to the original network with an algorithmic delay of 398 ms, this network will also be optimised.

Thus 2 different cases will be explored and their potential for further optimization discussed:

	Kernel sizes							
Layer	**EB1**	**EB2**	**EB3**	**EB4**	**FB**	**DB1**	**DB2**	**DB3**
Case 1: High algorithmic delay	55	160	160	160	160	55	15	5
Case 2: Low algorithmic delay	55	160	160	160	160	N/A	N/A	N/A

Table 9.1: Kernel sizes of the two cases considered

With these kernel sizes the number of parameters in each layer of the network can be computed as:

$$(n \cdot m \cdot l + 1) \cdot k \tag{9.1}$$

where

- n = kernel width
- m = kernel height
- l = input channels
- k = output channels
- 1 = bias

Thus for the two cases, the number of parameters in the network is:

	No. of parameters
Case 1: High algorithmic delay	93,392
Case 2: Low algorithmic delay	93,692

Table 9.2: Number of parameters in the models used in each case

Due the the majority of parameters residing in the EB, only the first 3 convolutional layers are pruned. These 3 layers contain a total of 90,502 parameters. All the results in this chapter will be the average AUC of the 4 noise types of Aurora-2 test set A. Thus a total of (noise types $\times$ SNR levels) $\implies (4 \times 7 = 28)$ AUC values in unique settings are averaged.

9.1 Pruning

First we investigate the VAD performance after pruning. The approach used is the iterative method presented in [27] which has already been described in section 8.3.1. Their work found that in order to obtain the best performance under pruning it was beneficial to add a penalty term in the form of L2-regularization to the loss function. The weight factor of this regularization is very application specific, and will therefore have to be found experimentally. This is however not the only unknown. The hyperparameters that will have to be decided upon before pruning is listed below:

- The weight factor of L2-regularization
- The learning rate
- The number of pruning iterations
- The number of parameters pruned in each iteration
- The re-training time after each pruning iteration

Due to the limited time frame of this project not all hyperparameters can be found experimentally, and thus some of them will be set initially based on intuition. Only the learning rate and L2 weight factor will be found experimentally

9.1.1 Finding hyperparameters

It has been decided to run parameter sweeps for finding the optimal learning rate and L2 weight factor experimentally. The number of pruning iterations is set to 5, the re-training time is set to 2 epochs after each pruning iteration, and the number of parameters pruned in each iteration is set to 10% of the remaining parameters. This leads to that the number of remaining weights after pruning is:

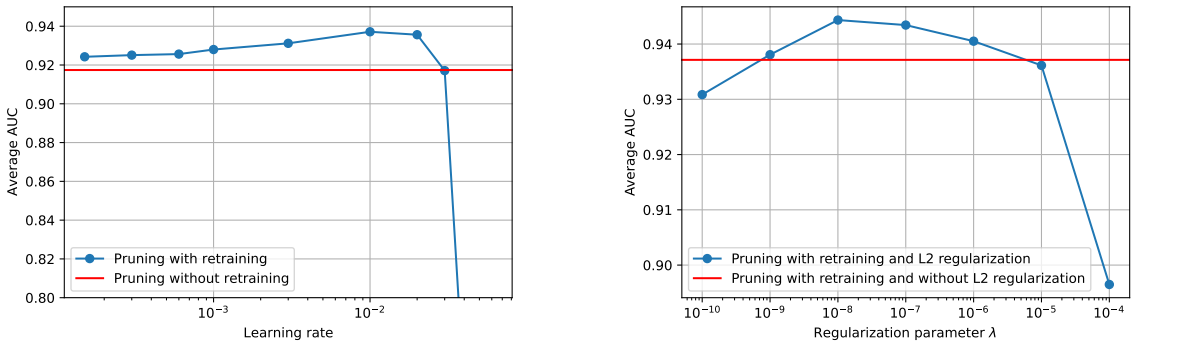
$$\text{remaining weights} = (1-p)^r \implies (1-0.1)^5 = 59\% \tag{9.2}$$

where p is the part of parameters pruned each iteration, and r is the number of iterations. Thus in these sweeps 59% of the weights still remain.

In section 9.1.1 parameter sweeps for finding the optimal learning rate and L2 weight factor is shown. First the optimal learning rate is found in fig. 9.1a whereafter the optimal L2 weight factor used in conjunction with the optimal learning rate is found.

In fig. 9.1a the AUC without retraining is used as reference and shown by the red line. The performance for different learning rates is shown by the blue line. It is seen that a too low learning rate results in a performance similar to the reference without retraining, while a too large learning rate causes the weights to explode and the performance to decrease. The optimal learning rate is found to be 10^{-2}

Afterwards this learning rate is used to find the optimal L2 weight factor, as seen in fig. 9.1b. The red line denotes the performance found using the optimal learning rate in fig. 9.1a. It is seen that too large weight factors leads to poorer performance. This is because the network is now more focused on minimising the L2-term and thus neglecting minimising the loss from the VAD. Interestingly, the performance is also seen to decrease when using "too low" L2 weight factors. This behaviour is indeed unexpected as a lower L2 weight factor eventually means the L2 penalty term will be disregarded. For now this behaviour will not be investigated any further, and instead the optimal L2 weight factor of 10^{-8} will be used for further experiments.

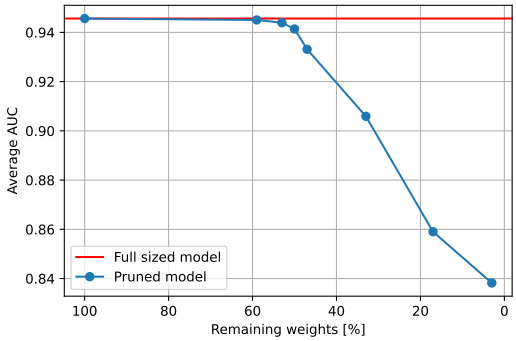


(a) Parameter sweep of learning rate

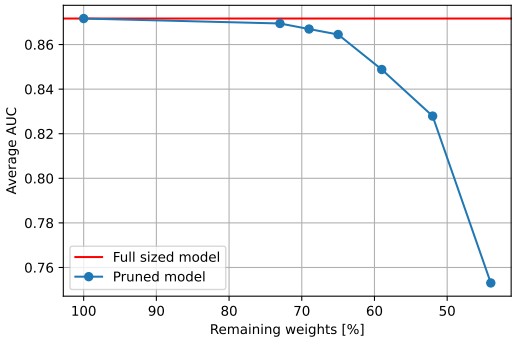
(b) Parameter sweep of L2 weight with optimal learning rate

Figure 9.1: Parameter sweeps for finding optimal hyperparameters

Having found the optimal hyperparameters, we will now investigate how different levels of pruning will affect the performance of the VAD. For varying the aggressiveness of the pruning, the number of parameters pruned in each iteration is varied, while the rest of the hyperparameters are kept fixed as noted in the beginning of section 9.1.1. In this experiment the two cases presented in chapter 9 are both investigated. The results are shown in fig. 9.2. The red line is denoting the performance with all parameters still present, and the blue line represent the performance of the pruned networks.



(a) The average AUC after pruning weights with the full model



(b) The average AUC after pruning weights with the low-delay model

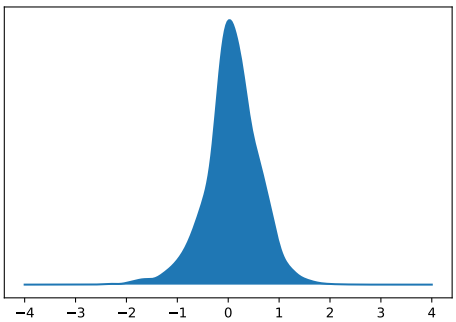
Figure 9.2: Experiments for finding the performance degradation at different aggressiveness of the pruning

For case 1 with the large algorithmic delay, a performance decrease is seen with less than 60% of the weights still remaining. From this point the performance seems to decrease linearly with further reduction of network size. For case 2, with the low algorithmic delay, a slight performance decrease is seen already when the remaining weights go below 75% and from this point the performance is decreased drastically as the network size is decreased.

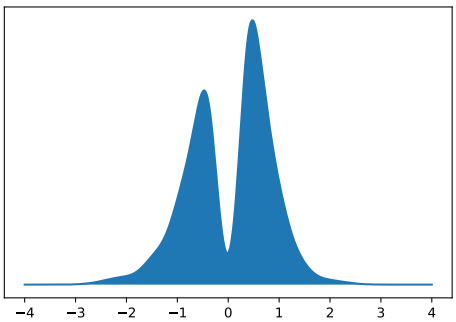
An interesting observation is that the the network with the large algorithmic delay can be pruned very aggressively and still perform as well - or even better than the un-pruned model with low algorithmic delay. With only 20% of the weights remaining in the network with high algorithmic delay, the performance is similar to that of the un-pruned low algorithmic delay. Additionally, by reducing the network size to 3% the AUC is only 3% lower than that of the low algorithmic delay.

Due to this only the network with high algorithmic delay will be investigated further in terms of pruning. For further work it is chosen to prune the network by 59% which results in 53415 remaining parameters (in the first 3 layers of the EB).

In fig. 9.3 the kernel density estimate of the weights of the layers involved in pruning is shown before and after.



(a) The weights before pruning



(b) The weights after pruning

Figure 9.3: Kernel density estimates of layers EB 1 to 3

It is seen that the weights are initially normal distributed around 0. Then the smallest weights are removed, but due to the retraining we instead get 2 approximately normal distributions on each side of 0.

9.1.2 Memory considerations

Pruning weights from the network leads to a new problem when it comes to the memory requirement. The parameters of the un-pruned network can be stored in a contiguous format in the same order as they are used, leaving no need for further addressing in memory. However, when some weights are pruned from the network there is no need for a place of these weights in the memory. As this sparsity increase there will be more gaps between weights. In order for the network to then fetch the correct parameter at the right time, an additional variable storing the address of where the parameters fits in the network is needed. Thus each parameter of the sparse network will now take up two variables in memory. The reduction in memory is therefore not directly proportional to the number of parameters pruned from the network. One way of storing the addresses of each parameter is using Compressed Row Storage (CRS) as shown in [33]. This is illustrated in fig. 9.4. The downside of storing addresses using this format is that the necessary bit width to store the addresses is dependent on the number of parameters in the original network, which in this case was found in chapter 9 to be ≈53,000 parameters. In order to store integers as large as this, a total of 17 bits is needed.

Index	0	1	2	3	4	5	6	7	8	9	10	11
Value	0	0	1.7	0.9	0	0	-0.7	0	0	0	0	0.4

Figure 9.4: Storing the addresses of each parameter as its absolute index

For this reason a different approach of storing the addresses is proposed in [33]. Instead of storing the absolute value of the addresses, instead the relative difference between the addresses of two adjacent non-pruned weights can be stored. In case the relative difference is larger than can be represented by the chosen bit width, instead a filler zero can be inserted. This acts as a regular parameter with its own address, but weight 0. This approach is illustrated in fig. 9.5.

Index	0	1	2	3	4	5	6	7	8	9	10	11
Difference			**2**	**1**			**3**				**4**	**1**
Value			1.7	0.9			-0.7				0	0.4

Filler zero

Figure 9.5: Storing the relative difference between parameters instead of addresses. Filler zeroes are ínserted if the relative difference can not be represented by the chosen bit width of addresses

9.1.2.1 Minimising memory requirements

In the following will be looked further into the approach illustrated in fig. 9.5, and the optimal bit width for reducing memory requirements will be found. In table 9.3 is shown the number of parameters that has a relative address difference that falls within the representable range of different bit widths.

Relative difference	<2	<4	<8	<16	<32	>32	Total
Occurences	51986	261	263	284	307	314	53,415

Table 9.3: Distribution of the relative difference in addresses

From this it is clear that a large number of the parameters are in fact still adjacent or separated by a maximum of 1 pruned weight, while the remaining weights are somewhat equally distributed among the larger relative differences. Having found the indeces of the pruned weights it is of further interest to find the bit width for representing the relative differences that minimise the required memory. For finding the needed memory 3 parameters are needed:

- The bit width of parameters
- The bit width of relative difference in adresses
- The number of filler zeroes

The bit width of parameters will be discussed later under quantization, and is therefore disregarded for now. Instead we focus on the bit width of the adresses and the number of filler zeroes needed,

Given a bit width β and the relative difference of address d, the number of filler zeroes z can be found as:

$$z = \lceil \frac{d+1}{\beta} - 1 \rceil \tag{9.3}$$

For example let us consider the $\beta = 2$bit example. If the relative address difference is larger than 3 a filler zero is needed. For every increment of 4 in the address an additional filler zero is needed. By applying eq. (9.3) to every parameter in the pruned part of the network it is possible to find the optimal bit width. It is important to note that each filler zero will have to be stored as a parameter, thus increasing memory cost.

In section 9.1.2.1 the total number of parameters needed to be stored, the number of these which are filler zeroes, and the total number of bits needed to store the addresses are denoted for different bit widths:

Bit width	1	2	3	4	5
Parameters needed (including filler zeroes)	71,156	61,777	57,241	55,041	54,019
Filler zeroes	17,741	8,362	3,826	1,626	604
Bits needed	71,156	123,554	228,964	440,328	864,304

Table 9.4: Comparison of the implication on memory requirements using different bit widths for storing the relative difference in addresses

It is seen that the larger the bit width, the fewer filler zeroes is needed. However, the cost of the extra bit width is larger than the cost of the filler zeroes. This is due to the very large amount of parameters that can be addressed using a single bit as shown in table 9.3. Even though an additional 17,741 filler zeroes are needed (which is 33% of the actual parameters), a bit width of 1 for addressing is found to be optimal. This is assuming the filler zeroes can be stored using a single bit, which is not necessarily the case in every architecture.

9.1.2.2 Conclusion on pruning

In this section has been investigated whether pruning is a valid approach for reducing the size of the network. Due to the majority of parameters residing in the first 3 layers of the EB, only these are pruned. Pruning was investigated for both the network with low and high algorithmic delay. The network with the large algorithmic delay was succesfully pruned by 41% while the network with the low algorithmic delay was successfully pruned by 25% with only an insignificant performance decrease. From this point on only the network with the large algorithmic delay was considered further. The approach would have been the same using the low-delay model.

Pruning the model size by 41% corresponds to pruning away 37,087 of 90,502 parameters.However, pruning the network results in the need for addressing of parameters as these are no longer contiguous in memory thus increasing the model size. It was found that the remaining 53,415 parameters can be addressed efficiently using only 71,156 bits in section 9.1.2.1.

9.2 Quantization

In this section quantization will be applied on the VAD algorithm. First some initial considerations on choosing the number of integer bits based on the dynamic range as presented in section 8.3.2 is discussed. Afterwards the VAD performance at different levels of quantization is found experimentally. The quantization is performed using both the naive approach presented in section 8.3.3.1 and the approach aiming to minimise the model size while maximising the SQNR presented in section 8.3.3.2

9.2.1 Dynamic range considerations

This part of the work will find the optimal number of integer bits based on the discussion of dynamic range in 2's complement presented section 8.3.2 and parameters from the trained model. First recall that a number in 2's complement is represented as:

$$x = -x_0(1-\Delta) + \sum_{i=-M}^{N-1} x_i \cdot 2^{-i} - x_0 \cdot \Delta \tag{9.4}$$

And the dynamic range is then:

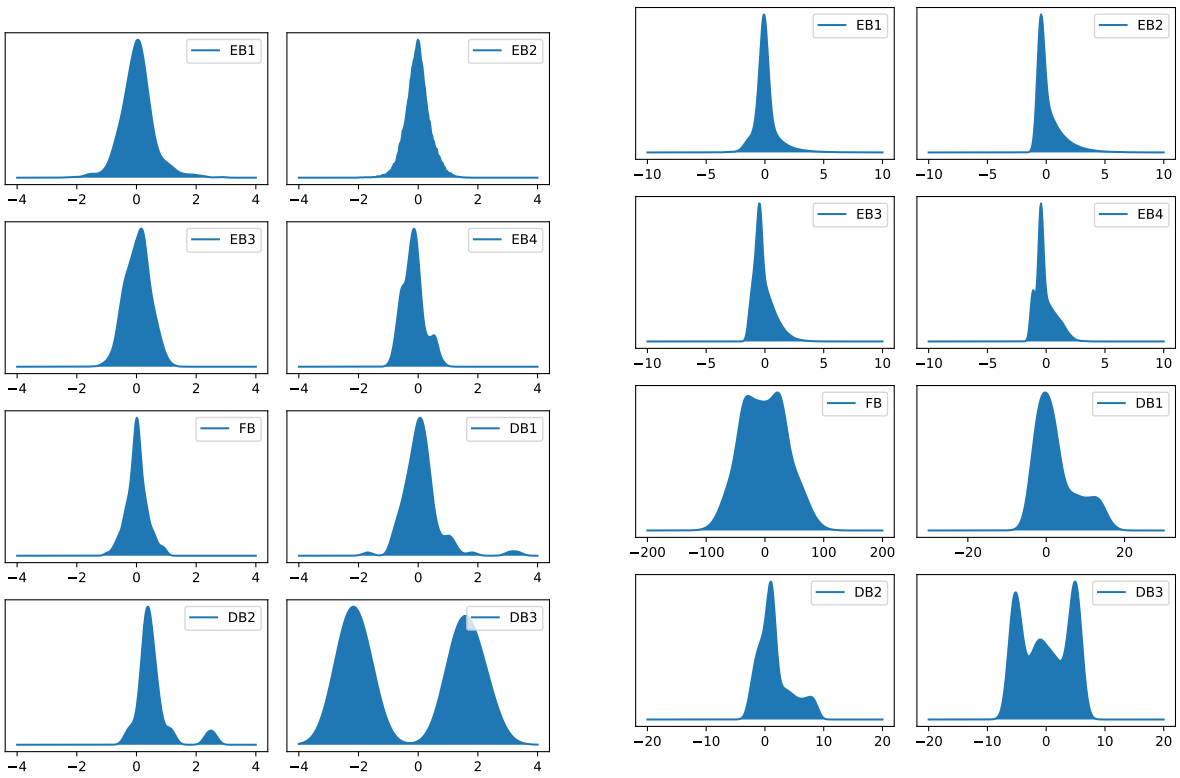
$$\text{Dynamic range} = \left[-2^M; 2^M - \Delta\right] \tag{9.5}$$

where M is the number of integer bits and N is the total number of integer plus fractional bits. Thus in order to avoid overflow and underflow it is necessary to make sure the numbers represented in the VAD remains within the dynamic range at all times.

In fig. 9.6 kernel density functions of the filter weights and the features output from the convolutional layers are shown. First let us consider the weights. It is seen that in most of the layers the weights are approximately gaussian distributed with a mean around 0. Additionally, the majority of the weights are within the range of $[-2;2]$. Some of the layers have some outliers which falls inside the range of $[-4;4]$. All weights of the trained network falls inside this range. Thus in order to fully cover the dynamic range of the weights 2 integer bits are needed.

Next let us consider the features output from the convolutions after being batch normalised and before the activation functions. These are shown in section 9.2.1. The figure is made using samples from a total of 5 minutes of audio where the noise type and SNR levels are randomized. The features are therefore representative for every environment considered in this work. It is seen that for the 4 layers of the EB the features are mostly falling within the range of $[-4;4]$ with only a few outliers. Next up is the FB, which contains way larger values than the earlier layers. This is due to no batch normalisation being used in this layer as explained in section 2.3.3.3. To fully represent the features from this layer a total of 7 integer bits is needed such that the dynamic range covered is $[-128;128]$. The features from the DB seems to fall in between those of the EB and FB. To fully cover their dynamic range 5 integer bits is needed for $[-32;32]$.

The raw audio input is not included in this, however as these experiments are carried out using .WAV files, every sample lies in the interval $[-1;1]$.



(a) Kernel density functions of weights

(b) Kernel density functions of features output from convolutional layers

Figure 9.6: Kernel density functions of the weights and features of the trained un-pruned network.

Due to having different dynamic ranges for the weights and the features of each block of the network, it is not trivial to decide on the number of integer bits to use. For this reason some more in depth discussions will follow on the topic.

As these features are sampled *after* the convolutions and *before* the activation functions it seems reasonable to consider the different activation functions as well. In the EB the Leaky ReLU activation function is used. The Leaky ReLU activation function is described as:

$$f(x) = \max(\alpha \cdot x, x) \tag{9.6}$$

Thus for every positive feature the value is unchanged. Because of this it is of interest to have the full dynamic range covered. Next let us consider the sigmoid activation function which is used in the FB and DB layers. This is described as:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{9.7}$$

As the sigmoid function only output values between 0 and 1 this has an effect on the dynamic range of the features. In fig. 9.7 is illustrated how the resolution of the sigmoid activation function is affected by the dynamic range used, and in section 9.2.1 the exact resolution of the output is shown. For a range of different dynamic ranges is drawn a box on top of the function. These boxes represent the resolution of both the input and the output of the sigmoid

function. The horizontal lines depict the resolution of the input, while the vertical lines depict which part of the output can be activated when having the aforementioned resolution in the input. It is seen that when no integer bits are used, and the dynamic range is thus $[-1;1]$, only a very small part of the sigmoid function will ever be activated. In contrast, when using 3 integer bits, and a dynamic range of $[-8;8]$ almost the full output resolution is used. Thus the effect of features outside this range will only have a minimal effect on the performance. Additionally, as the VAD is a binary classification problem it is not necessarily a problem that the features converge towards 0 or 1 in these parts of the network.

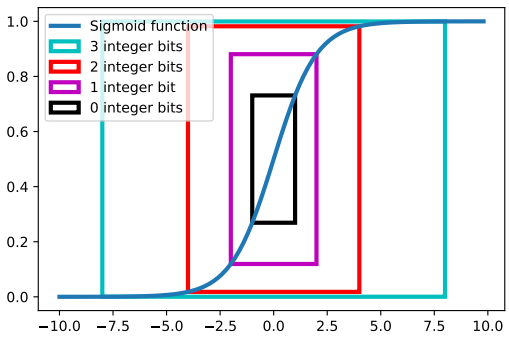


Figure 9.7: Resolution of the sigmoid function using different numbers of integer bits - and thus having different dynamic ranges

To further highlight this the resolution of the outputs from sigmoid is shown in the table below. Working with a dynamic range of $[-8;8]$ the resolution of the output is $[0.0003;0.9997]$.

Integer bits	**3**	**2**	**1**	**0**
Sigmoid resolution	$[0.0003;0.9997]$	$[0.0180;0.9820]$	$[0.1192;0.8808]$	$[0.2689;0.7310]$

Table 9.5: Resolution of the sigmoid activation functions using different number of integer bits. It is assumed that the number of fractional bits is very large in this case, such that the dynamic range is $\approx [-2^M;2^M]$.

Further experiments have been carried out to investigate the effect of the dynamic range. First is an experiment in which it is found what the probability of overflow occuring before the activation function is for each layer at different choices of integer bits. As the overflow mainly seems to occur in the FB block our intuition is confirmed. The overflow in the sigmoid layers is as previously mentioned not necessarily a problem, due to the fact that we work with a binary classification problem.

Integer bits	4	3	2	1
EB1	0	0.001	0.011	0.053
EB2	0	0.001	0.014	0.076
EB3	0	0.001	0.001	0.024
EB4	0	0	0.933	0.016
FB	0.720	0.866	0.933	0.965
DB1	0.008	0.182	0.329	0.607
DB2	0	0.040	0.186	0.336
DB3	0	0	0.577	0.736

Table 9.6: The probability of overflow in each layer using different numbers of integer bits

9.2.2 Experiments fixed point performance

The following section will contain experiments finding the VAD performance at different levels of quantization. Additionally, quantization and pruning will be tested when used in conjunction.

Having analysed the effect on the dynamic range of the network in theory, this next experiment will aim at finding the effect of dynamic range in practice. In this experiment the pre-trained model is quantized into different dynamic ranges. The tool used is *QPyTorch* [34], which is a python module that can be used in extension with PyTorch to simulate low-precision arithmetic of a network. The features used in this work are:

- Numbers can be represented as floating point and fixed point
- For floating point the number of bits used for exponent and mantissa can be set. (Cannot go higher than a single precision number)
- For fixed point the integer bits and fractional bits can be set.
- "Nearest rounding" is used for quantization.
- The number format for each variable can be set individually, allowing for different bit widths through the network
- Overflowing values will not roll over, but will instead saturate at the nearest representable value

Even though the module has some good features for this project it also has some minor limitations:

- Overflow within an operation is not detected, i.e. if the intermediate results in the MAC overflows but the final result is within dynamic range
- The arithmetic operations are still carried out using floating point arithmetic, however this is not of importance as all values used will be quantized beforehand.

Experiment on dynamic range

As mentioned briefly above, first will be tested experimentally what effect the dynamic range has on the performance. These results are seen in section 9.2.2 where is found the VAD

performance using fixed-point representation. The word length is fixed at 32 bits, whereas the number of integer bits - and therefore also fractional bits is varied. However the number of fractional bits is so large that it will have no effect when decreased due to the quantization step being very small. Thus the effect of integer bits is isolated. The experiment is carried out using the network with large algorithmic delay.

Dynamic range	± 0.5	± 1	± 2	± 4	± 8	± 16	± 32	32 bit floating point
AUC	92.72	93.42	94.01	94.05	94.07	94.12	94.19	94.20

Table 9.7: Performance using different number of integer bits. Total word length is 32 bits in all tests

It is seen that even with very small dynamic ranges the performance of the VAD is still good. Between having a dynamic range of $[-2; 2]$ and 32 bit floating point the difference in performance is minimal. This suggests that the network is not heavily affected by overflow and the exact magnitude of features is less important than their sign.

Following the analysis of dynamic range and the experiment in section 9.2.2 it is decided that only two integer bit will be used for further experiments, resulting in a dynamic range of $[-4; 4]$. This way the performance is proven to be good in section 9.2.2 while most of the features and weights can be fully represented as shown in fig. 9.6

9.2.2.1 SQNR optimized quantization

An important part of this work on quantization is to find the performance of two different approaches to quantization. The naive approach as described in section 8.3.3.1 and the approach which aims to optimise for SQNR while minimising the model size as described in section 8.3.3.2.

First we will find the optimal bit widths for each layer, using the first layer of the EB as a reference. In section 8.3.3.2 it was described that the optimal bit width is given as:

$$\beta_i - \beta_j = \frac{10\log_{10}(\frac{\rho_j}{\rho_i})}{\kappa} \tag{9.8}$$

where:

β = the bit width of the layer
ρ = the number of parameters in the layer

Given that the number of parameters contained in each layer is:

Layer	EB1	EB2	EB3	EB4	FB	DB1	DB2	DB3
Bit width	1680	72015	16807	2242	642	222	62	22

Table 9.8: The number of parameters in each layer

The optimal bit widths are found to be:

Layer	EB1	EB2	EB3	EB4	FB	DB1	DB2	DB3
Parameters	β_0	$\beta_0 - 5$	$\beta_0 - 3$	$\beta_0 - 1$	$\beta_0 + 1$	$\beta_0 + 3$	$\beta_0 + 5$	$\beta_0 + 6$

Table 9.9: The optimal bit width for each layer for minimising model size while maximizing SQNR

Thus the layer containing more parameters need a smaller bit width.

Further experiments

Now that the optimal bit widths are known, a series of 24 experiments is carried out on the network with the large algorithmic delay. The aim is to find the VAD performance under different levels of quantization. The experiments are as follows:

- a reference - found using single precision floating point
- 13 experiments using the naive approach to quantization. For each experiment the bit width is lowered by 1, covering the range from 16 bits to only 4 bits.
- 10 experiments using the SQNR optimised approach. The bit widths are found as table 9.9. For each experiment, the bit width of every layer is reduced by 1.

The results from these experiments are shown in table 9.10. Each case has been tested on both the full network and the pruned network. Additionally, the resulting network size in bits is denoted for each case. The model sizes for the pruned network is including filler zeroes and bits needed for adressing as described in section 9.1.2.1. Thus the model size is increased by 71,156 bits when the network is pruned.

	Bit width per layer								Model size ($\times 10^3$ bits)		AUC (%)	
	EB1	**EB2**	**EB3**	**EB4**	**FB**	**DB1**	**DB2**	**DB3**	**Pruned**	**Not pruned**	**Pruned**	**Not pruned**
Case #0	32	32	32	32	32	32	32	32	1765	2514	94.01	94.23
Case #1	16	16	16	16	16	16	16	16	1078	1257	93.64	93.92
Case #2	15	15	15	15	15	15	15	15	1015	1178	93.58	93.89
Case #3	14	14	14	14	14	14	14	14	952	1100	93.53	93.86
Case #4	13	13	13	13	13	13	13	13	889	1021	93.34	93.70
Case #5	12	12	12	12	12	12	12	12	826	942	93.12	93.70
Case #6	11	11	11	11	11	11	11	11	763	864	92.96	93.02
Case #7	10	10	10	10	10	10	10	10	700	785	92.07	93.02
Case #8	9	9	9	9	9	9	9	9	637	707	92.07	92.74
Case #9	8	8	8	8	8	8	8	8	574	628	92.17	92.55
Case #10	7	7	7	7	7	7	7	7	511	550	91.84	92.58
Case #11	6	6	6	6	6	6	6	6	448	471	91.96	91.99
Case #12	5	5	5	5	5	5	5	5	385	392	90.55	85.90
Case #13	4	4	4	4	4	4	4	4	322	314	41.38	39.61
Case #14	16	11	13	15	17	19	21	22	806	891	93.54	94.07
Case #15	15	10	12	14	16	18	20	21	743	812	93.42	93.72
Case #16	14	9	11	13	15	17	19	20	680	734	93.11	93.42
Case #17	13	8	10	12	14	16	18	19	617	655	93.07	93.14
Case #18	12	7	9	11	13	15	17	18	554	577	92.90	93.60
Case #19	11	6	8	10	12	14	16	17	491	498	92.08	93.27
Case #20	10	5	7	9	11	13	15	16	428	420	91.99	92.91
Case #21	9	4	6	8	10	12	14	15	366	341	91.29	92.19
Case #22	8	3	5	7	9	11	13	14	303	262	90.60	91.91
Case #23	7	2	4	6	8	10	12	13	241	184	89.59	90.50

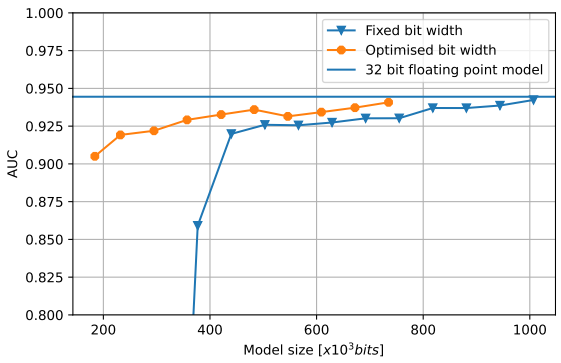
Table 9.10: Performance of the VAD and model sizes at different levels of quantization, both with and without prior pruning.

9.3 Analysis of results

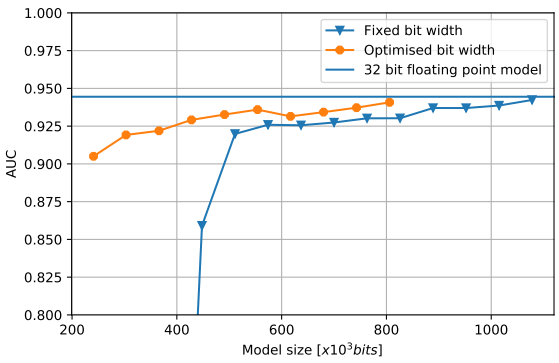
It is seen that as the bit widths are decreased, the performance of the VAD is also decreased, but not exceedingly so. An interesting observation is that using the naive quantization, an approximately linear decrease in performance is seen from a bit width of 16 until a bit width of 5, where after the performance explodes and simply stops working. This is believed to be due to the quantization step being as large as 0.5 when using only 4 bits, which is not enough to contain the more fine grained features of the sigmoid layers. In contrast when using the SQNR optimised approach it is seen that the approximately linear decrease in performance is kept even at bit widths as low a 2 in the EB.

Using the bit width optimised approach it is possible to reduce the un-pruned model size by almost $3\times$ with very limited decrease in performance (case #14). Using the naive approach it is possible to half the model size with very limited decrease in performance (case #1)

In section 9.3 is visually illustrated the performance at different quantization levels introduced in table 9.10. Here it is seen that the SQNR optimised approach is outperforming the naive quantization in terms of performance compared to model size for both the pruned and un-pruned models.



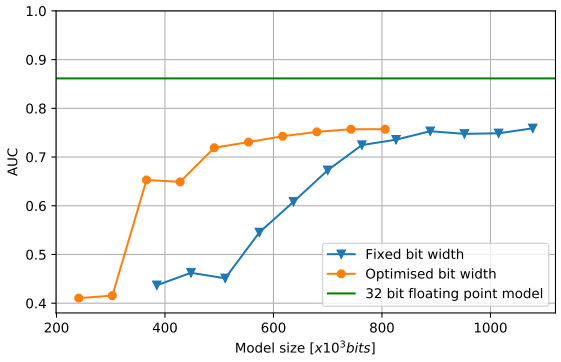
(a) Large algorithmic delay - quantized only



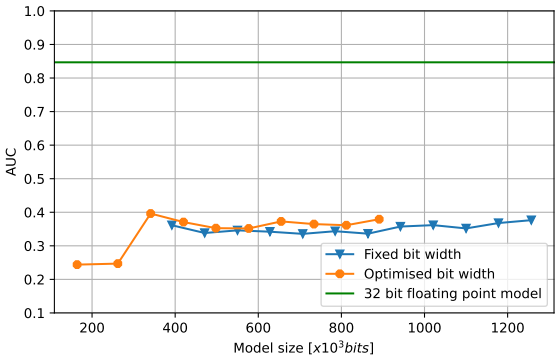
(b) Large algorithmic delay - pruned and quantized

Figure 9.8: Visualization of the results from table 9.10 - comparison of the two approaches to quantization for large algorithmic delay

In section 9.3 is the same experiment as shown in section 9.3 repeated for the model with the low algorithmic delay. Interestingly, the results from this experiment is in sharp contrast to those using the model with large algorithmic delay. In fact a drastical decrease in performance is observed at every quantization level. In particular the already pruned model seems unfit for quantization.



(a) Low algorithmic delay - quantized only



(b) Low algorithmic delay - pruned and quantized

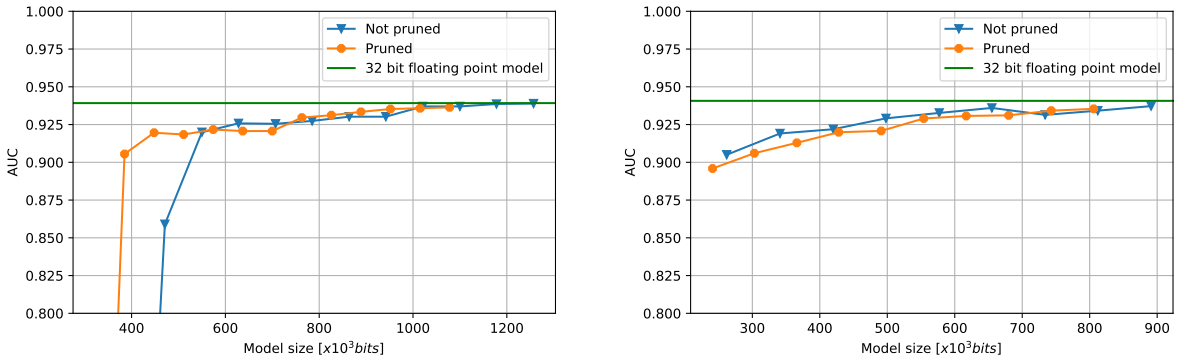
Figure 9.9: Visualization of the results from table 9.10 - comparison of the two approaches to quantization for low algorithmic delay

9.3.1 Thoughts on combined pruning and quantization

In the previous section it was discussed how the choice of quantization approach affected the VAD performance compared to the network size. The VAD has also been tested both with and without pruning as a step before quantization, and under which circumstances using both in combination is beneficial will be discussed in this section. In section 9.3 it was found that applying quantization to the model with low algorithmic delay decreased the performance so badly that the VAD is no longer capable of detecting speech, thus only the model with large algorithmic delay will be further considered.

In section 9.3.1 is the VAD performance after quantization plotted against the model size for both the pruned and the un-pruned. The results are the same as already presented in

table 9.10.



(a) Fixed bit width quantization with and without pruning

(b) SQNR optimised quantization with and without pruning

Figure 9.10: Visualization of the results from table 9.10 - comparison of performance with and without prior pruning

It is seen that when using the naive approach to quantization, it is possible to reduce the model size more if the model is pruned beforehand. However, for SQNR optimised quantization the performance is generally better when the model is not pruned beforehand.

The reason why the model size is eventually getting lower when the model is not pruned is that in a pruned layer it is necessary to consider addressing of parameters as already described in section 9.1.2.1. It was found that a total of 71,156 bits is the minimum needed to store the addresses only. As the bit width of the parameters get smaller, the addressing has a larger effect on the model size. The different combinations of using pruning and quantization will be further discussed in the next section.

9.3.2 Conclusion on quantization

Having carried out simulations for finding the performance of the VAD using different approaches to quantization, this section will be a discussion rounding off the work on quantization. Two different approaches to quantization has been considered, and both of these have been used both standing alone and in combination with pruning. For all of the experiments two models have been considered: one with an algorithmic delay of 398 ms and one with an algorithmic delay of 23 seconds. For the following four cases, the results considered are all listed in table 9.10.

Naive quantization without pruning

This approach to quantization is the most simple. The entire model is quantized to the same bit width. This approach generally leads to bad performance when compared to the approach using SQNR optimised bit widths as seen in section 9.3 and section 9.3, however it outperforms the model which has been pruned before the naive quantization is applied as seen in section 9.3.1.

Naive quantization with pruning

This approach prunes the model before it is quantized with a fixed bit width for the entire model. In fig. 9.2 it was found that the large-delay model can be pruned by 41% while the low-delay model can be pruned by 25% with only insignificant performance decrease. In section 9.3.1 it is seen that pruning before quantization does not lead to better performance at the same model size as compared to the un-pruned model. This is partly because the addressing needed takes up a significant part of the model size as the bit widths are decreased.

Optimised quantization without pruning

This approach has shown the best performance. The bit widths used in each layer of the network is minimised while maximising for the SQNR as found in section 8.3.3.2. From section 9.3 and section 9.3 it is seen that this approach to quantization outperforms the naive approach, and from section 9.3.1 is is seen that the performance is better when the model is not pruned.

Optimised quantization with pruning

This approach shows worse performance than SQNR optimised quantization without pruning, however it still outperforms the model using naive quantization - both pruned and un-pruned.

9.4 Conclusion on experiments

Both pruning and quantization have been implemented and an extensive series of experiments has been carried out. The results when applied to the network of fig. 6.3 from these experiments are all seen in table 9.10, and for further analysis of these results a series of plots have been made. From this analysis of the results it was found that the best approach for minimising model size while retaining VAD performance was using the SQNR optimised quantization approach with no prior pruning.

The methods have both been tested for both the model with large and small algorithmic delays. It was found that the model with the large algorithmic delay is way more suitable for pruning and quantisation than the model with low algorithmic delay.

Architecture considerations 10

In this chapter the work on pruning and quantization presented in the two previous chapters will be discussed in the context of a real-world implementation. This discussion will be based on two different types of architectures commonly used for embedded devices - a Digital Signal Processor (DSP) and an Field Programmable Gate Array (FPGA)/Application-Specific Integrated Circuit (ASIC).

The discussion will focus on fixed point arithmetic, memory alignment before and after pruning, and the potential for implementing the SQNR optimised quantization

10.1 DSP

First will be discussed how the network with aforementioned optimisations can be implemented on a DSP. More specifically will be considered the *Tencilica Hifi3 DSP* that is used in the low-power peripheral audio solution *Dialog DA14195* that is currently in use by RTX.

The DA14195 is an open audio platform for high-end active headphones, which is used for various speech processing tasks such as noise cancellation, noise reudction, voice enhancement and echo cancellation. It integrates a 32-bit ARM Cortex-M0 microcontroller and a 32-bit Tencilica Hifi3 DSP in the same chip [35]. Due to the large amount of MAC operations in the network only the DSP will be considered while the ARM microcontroller will be disregarded.

Detailed information and schematics of the DA14195 is not publicly available, and for this reason only the DSP will be considered. The discussions of this section is based on the datasheet of the DSP [36].

10.1.1 Architecture

The Hifi3 DSP is a Single Instruction/Multiple Data (SIMD) processor that can work in parallel on two 24/32 bit data items or four 16-bit data items. Additionally, it supports fixed-point MAC of four 24x24-bit, four 32x16-bit or four 16x16-bit operands per cycle. Two 32x32-bit operands can be multiplied per cycle. The floating point unit supports two IEEE-754 floating point MACs per cycle

The load/store unit is capable of loading or storing up to two 24-bit or 32-bit elements, four 16-bit elements or one 64 bit element in each cycle.

The Hifi3 DSP is based on the Very Long Instruction Word (VLIW) which supports execution

of three operations in parallel in just a single instruction. These three operations are carried out by different slots in the architecture:

- Slot 0: Responsible for loading and storing, bitstream and Huffman operations. Also contains an Arithmetic Logic Unit (ALU) for performing core operations
- Slot 1: Contains an ALU and a MAC unit as well as a unit for performing floating point operations.
- Slot 2: Contains an ALU and a MAC unit.

In fig. 10.1 the architecture of the Hifi3 DSP is illustrated

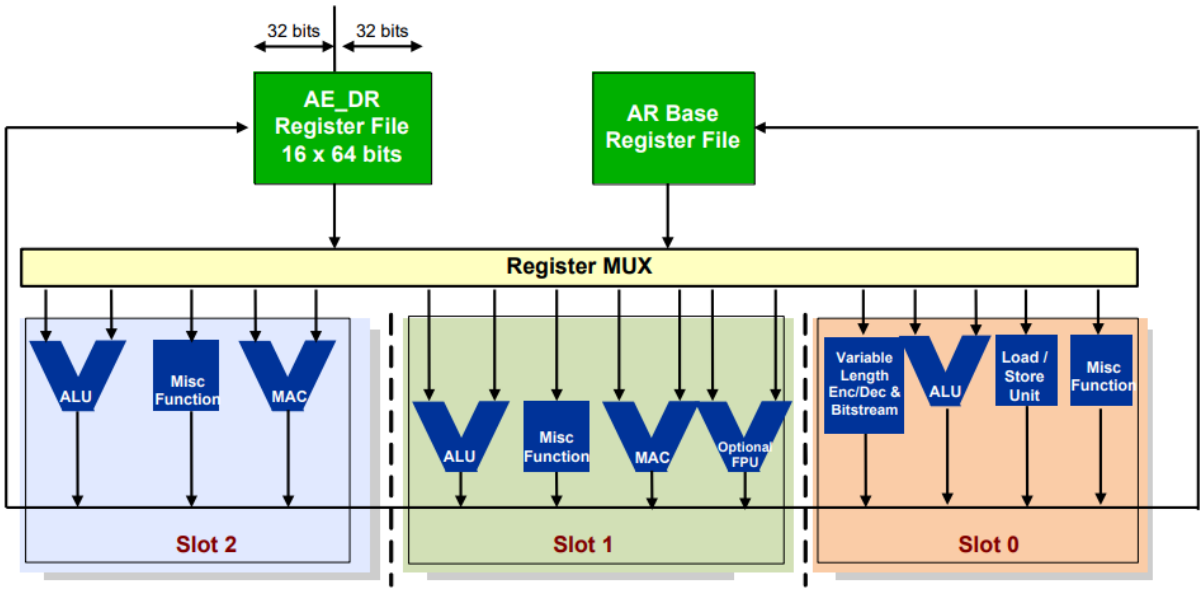


Figure 10.1: Architecture of the Hifi3 DSP [36]

10.1.2 Quantization

Based on the above description of the architecture, this section will provide an analysis on how to apply quantization of the network when implemented on a DSP. Two things are particularly important to consider in terms of quantization. That is:

- Which bit widths can the parameters be stored as?
- Which bit widths is supported by the MACs?

This was briefly mentioned in the architecture overview. Only elements of 16, 24, 32 or 64 bits can be loaded and stored using standard instructions. The same is the case when it comes to MACs. Thus the architecture of the DSP is putting a constraint on the potential for introducing quantization into the network.

First of all, the SQNR optimised approach is not applicable as the DSP can not operate on different bit widths of each layer. Secondly, the DSP does not support operations using fewer than 16 bits.

Considering the cost function presented in eq. (8.1), if it is considered more important to reduce the memory than the the execution time and number of operations it can be beneficial to encode multiple parameters of smaller bitwidth within a single word. This would however

introduce the need of a custom routine than can fetch the parameters correctly from memory and therefore introduce an additional overhead in computation time.

As the smallest possible operations supported by the MACs of the DSP is 16x16-bits zeroes would have to be padded as Most Significant Bit (MSB) until a valid bit width is reached.

If computation time is of higher importance than memory cost, it is beneficial to use the naive approach to quantization to 16 bit fixed point as presented in section 8.3.3.1.

10.1.3 Pruning

In section 8.3.1 was found how much the network can be pruned before performance is decreased. In this section will be discussed how this can be implemented on a DSP similarly to the section of quantization above.

Additionally, in section 9.1.2 was described how introducing pruning would ruin the contiguous alignment of parameters in memory, and thus introduce the need for an additional routine that can fetch the correct parameters at the right time. This was followed by an additional analysis on how the remaining parameters can be saved in the most efficient way in terms of memory usage. It was found that the most optimal way to save the locations of the remaining parameters was by using the relative difference in address at a bit width of 1 and then include filler zeroes, also of bit width 1, when the maximum representable relative difference was exceeded. As also explained in section 10.1.2, on the DSP it is not possible to save any parameters using a bit width of 1. For this reason both the relative differences in memory addresses and the filler zeroes would have to be saved as a minimum of 16 bits - unless an additional routine for unpacking the 1-bit words is introduced. This would while reducing the memory requirement lead to a significant decrease in computation time.

Following the same logic as in section 9.1.2.1, a 16-bit word for the relative difference in addresses of each parameter would have to be saved. With a bit width of 16 it is not necessary to include any filler zeroes. After pruning 53.415 parameters was still remaining while 37.087 was removed. In section 10.1.2 it was found that at least 16 bits is necessary in order to perform the operations on the DSP. Having the relative difference in addresses being stored using the same bit width as the parameters the memory requirement would only increase by introducing pruning.

10.1.4 Conclusion on DSP implementation

In section 10.1.2 and section 10.1.3 was discussed whether the methods for reducing the computational time and memory requirements from chapter 8 is suitable for implementation on a DSP. It was found the SQNR optimised quantization and pruning is not feasible for DSP implementation due to the limitation in customizable word lengths. This problem can be circumvented by introducing an additional routine responsible for unpacking the parameters of smaller bit widths stored together within a larger word. This would however lead to increased computation time. For an implementation on a DSP the most feasible approach is found to be quantization to 16 bits with no prior pruning.

10.2 FPGA and ASIC

In section 10.1 it was discussed how the optimisations considered in chapter 8 could potentially by implemented on a DSP. It was found that due to the limitation of bit widths supported by the DSP only the naive approach to quantization proved to be feasible.

In this section will instead be considered a potential implementation on customizable hardware architecture like FPGA and ASIC. As these architectures are only rarely used by RTX, this discussion will instead be based on a type of FPGAs used for educational purposes at Aalborg University. The FPGA of interest is of the Altera Cyclone V series [37]. More specifically it is chosen to consider the Cyclone V GTD5 which specifications can be found in [38]. As was the case in the discussion of DSP implementation, the focus will be on how pruning and quantization as presented in chapter 8 can be utilised on this architecture.

The discussion will focus on how the optimisations of chapter 8 can be implemented on this FPGA.

As seen in eq. (2.2) the convolutional operations involved in the network is similar to those of a Finite Impulse Response (FIR) filter. For this reason let us consider the approach for implementing a FIR filter on an FPGA as presented in [39]. The approach is illustrated in fig. 10.2:

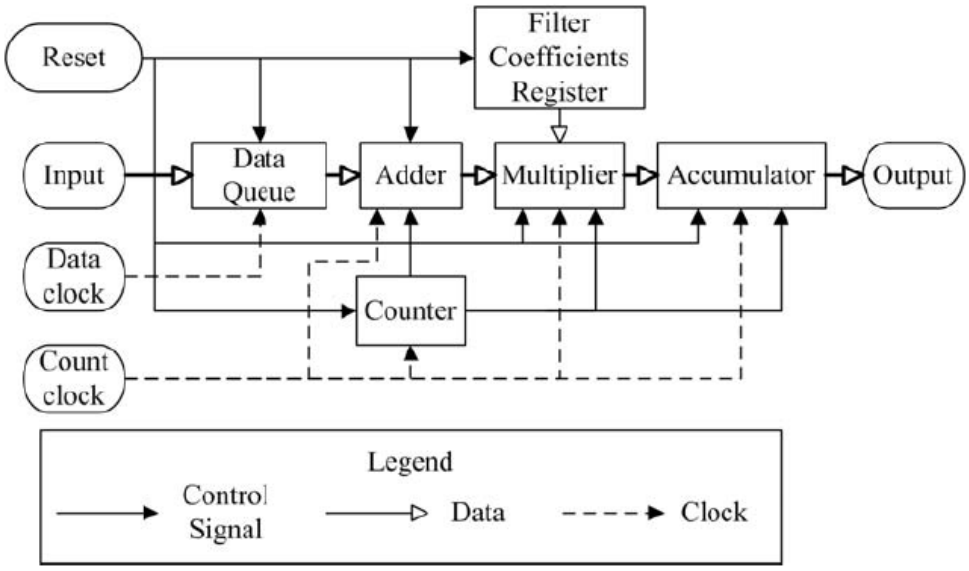


Figure 10.2: Approach for implementing a FIR filter as presented in [39]

Disregarding the clock signals, the rest of the figure will be used as foundation for this discussion on pruning and quantization on an FPGA. To ensure a natural flow of this discussion, quantization will be considered first followed by pruning.

10.2.1 Quantization

This discussion on how to utilise quantization on an FPGA will focus on the *multiplier* and *accumulator* blocks of fig. 10.2. Thus the only operations that will be considered is multiplications and additions.

On an FPGA the adders and multipliers can be set up the way in which the designer wants, that includes both fixed point and floating point as well as different word lengths. Thus it is also possible to use varying word lengths in the different parts of the networks such that the SQNR optimised quantisation can be implemented.

Additionally, in terms of storing the parameters in memory is important to consider what at which word lengths the parameters can be stored. On an FPGA the built-in memory blocks can be configured to store variables using the desired bit width.

10.2.2 Pruning

Having shortly outlined that quantisation is no problem on an FPGA, this section will continue the discussion on pruning with focus on the specific FPGA Altera Cyclone V GTD5. The main issue that will be discussed is how to address the parameters after pruning as found in section 9.1.2, i.e. how to make sure we only perform perform computations using the remaining parameters.

As was already found in section 9.1.2 we need to store both the parameters and their respective address in the original un-pruned network. For this reason we will first present the two types of embedded memory available in the Altera Cyclone V GTD5: Memory 10K (M10K) and Memory Logic Array Block (MLAB).

The Altera Cyclone V GTD5 contains 446 M10K memory blocks, each of which can store 10 Kb of data and are ideal for larger memory arrays. Additionally it contains and 679 MLAB. These are optimised for the implementation of shift registers allowing for fast data transfer and are therefore useful in a variety of DSP applications and thus also useful for storing parameters of the network in this work. Each of these contains 640 bits of memory. Thus the combined memory of the Altera Cyclone V GTD5 is:

- 4,460 Kb M10K memory
- 424 Kb MLAB memory

For the fastest possible data transfer it is desired to use the MLAB as much as possible.

In table 9.10 was found the model sizes for both the pruned and the non-pruned network at different levels of quantisation. The majority of the cases considered are larger than 424 bits and can thus not fit inside the MLAB. In this case it is necessary to either pick a larger FPGA or resort to using the M10K memory instead which contains more than enough memory.

10.3 Conclusion on research question 3

The previous 3 chapters have focused on answering research question 3:

- *How can we reduce the computational cost of the VAD in a real time application while maintaining performance?*

In chapter 8 was discussed different methods for optimising the network on a high abstraction level. More specifically, these methods have been pruning and quantisation. Both of these methods have been based on recent up-to-date academic publications. Only one approach to pruning was investigated, while two different approaches to quantisation was discussed.

Pruning was achieved through an iterative process where parameters were pruned and the remaining parameters retrained in turns. When pruning parameters from a network the parameters can no longer be stored contiguous in memory and applied to the right convolutions with no further addressing. For this reason two approaches for storing the address of the parameters have been discussed. Quantisation was discussed as a naive approach where the entire network operate on the same bit width, as well as an approach aiming to maximise the SQNR while minimising the network size.

Following this introduction to the methods, in chapter 9 these methods have been applied to the network of this work presented in fig. 6.3. As this has been a natural extension from research question 1 and 2, the model includes adversarial multi-task learning and has been tested at both a large and a low algorithmic delay. It was found that the best performance can be achieved using the SQNR optimised quantisation with no prior pruning for the model with the large algorithmic delay. The model with low algorithmic delay has proven unfit for both pruning and quantisation.

Finally in chapter 10 was investigated how the results obtained in chapter 9 can be utilised on different hardware architectures. Both DSP and FPGA architectures have been discussed. It was found that while a DSP allows for fast development and efficient computations of the convolutions of the network, it is limited in terms of quantisation. The DSP investigated is currently in use by RTX and only supports arithmetic fixed-point operations at word lengths of 16, 24, 32 and 64 bits. From table 9.10 it is seen that the word length can be decreased significantly lower than 16 bits while still maintaining performance. Thus in order to fully utilise the quantisation an FPGA implementation is necessary. Using an FPGA also allows for using the SQNR optimised approach to quantisation. An FPGA implementation is however expected to result in higher development time.

Part IV

Final thoughts

Discussion 11

In this last part of the project will be an evaluation of the work done. First will be a discussion on some of the results obtained throughout the work, whereafter some recommendations for further implementation will be listed.

In the following section will be discussed some questions that is still left unanswered. That includes the quality of the datasets, some important trade-offs and the approach to lowering the algorithmic delay.

11.0.1 Datasets

In this work has been used the Aurora-2 and the TIMIT databases. Both of these follows a characteristic structure in which the speech is always present in the middle part of the files while the beginning and the end contains no speech. It was found that when training the network on these individual files it was overfitting to this structure. In order to avoid this overfitting, instead a number of files was concatenated after each other.

In the case of the Aurora-2 database around 73% of the audio is labelled as speech and the speech segments is very short utterances of single letters and numbers. The TIMIT database contains 85% speech and the speech segments is full sentences. Thus the two databases both got some challenges seperating it from a real-life scenario: Aurora-2 consists of very short utterances while TIMIT is very biased towards speech. However, TIMIT does actually resemble a real life conversation quite good, as it contains full length sentences of different speakers only interrupted by a short silence. The same goes for Aurora-2 even though the utterances are of much shorter duration

In both cases a real-life scenario is resembled much better when audio files are concatenated than when audio files are used ony by one. This real-life resemblance can be further enhanced by only concatenating audio corrupted by the same noise type at the same SNR level.

11.0.2 Algorithmic delay vs computational complexity trade-off

Early in this work was introduced that one of the goals of this work is to find a VAD algorithm with low latency. In order to ensure low latency two aspects are of importance. That is the algorithmic delay and the computational overhead, i.e. the time it takes to compute. In chapter 7 was investigated how the network is performing when the algorithmic delay is lowered.

In chapter 9 was then investigated how the network is performing after pruning and quantization. These experiments were carried out on both the network introducing 398 ms of algorithmic delay and the network introducing only 23 ms of algorithmic delay. It

was found that while the large algorithmic delay network could be moderately pruned and heavily quantized and still perform well, the network with low algorithmic delay was very sensitive to both pruning and quantization. In fact the performance was so heavily degraded that it was worse than randomly guessing.

Because of this it seems that we can not have both low algorithmic delay and low computational complexity at the same time. Therefore, this trade-off is dependent on the scenario and hardware platform in which the VAD is to be used.

11.0.3 Other approaches to reducing algorithmic delay

In chapter 7 was investigated the performance under different algorithmic delays. In this work it was assumed that the past and future context was of the same size. This obviously puts a natural constraint on how much the algorithmic delay can be lowered, but also limits the total context on which the VAD decision has to be made. Therefore we will shortly list other potential approaches that can possibly lead to an even lower algorithmic delay or even better performance at similar algorithmic delays. Due to the limited time frame of the project these have not been thoroughly investigated, and it is therefore not known if they are actually better approaches.

- Instead of considering the middle frame as the frame to be labelled, instead this frame can be shifted forward in time. This way the past context will be larger than the future context and thus the VAD decision can be made on a larger foundation of samples without increasing the algorithmic delay.
- When lowering the algorithmic delay a shorter timespan of audio is considered. A potential way of increasing performance is to introduce a "hold-time" after detecting speech. This will reduce the flickering of the VAD labels and reduce the sensitivity to short breaks in the speech.

11.1 Recommendations - from academia to industry

This work has mainly been an academic investigation of a deep learning based VAD which is potentially to be applied in an industrial context. Therefore some recommendations on how to take this from an academic project to an industrial implementation will be listed below:

- The vast majority of operations included in the forward step is MACs. Thus an architecture that is optimised for these operations - like a DSP - is desired.
- Even though pruning the network has proved useful for reducing the memory requirement, this removes the potential for storing parameters contiguously in memory and introduces the need for more advanced methods for retriving parameters from memory.
- As the network contains $\approx$93,000 parameters this is the minimum memory requirement. It has not been investigated how much memory is needed to store the intermediate values of the network, thus the actual memory requirement is higher.
- For a fully optimised implementation of the network an FPGA or ASIC platform is necessary. However this also comes at a cost of a higher development time, so in many cases a DSP implementation will be sufficient.

- If an FPGA or ASIC approach is taken it can be beneficial to treat the VAD as a system of its own. Then when speech is detected an interrupt or similar can be sent to the system to which it is attached (for example noise cancelling), and thereby reducing power consumption by not running the rest of the system when not necessary
- It is important to include many different noise types in the training set, as the performance of the VAD is proven to be better when presented to known noise types.
- The lower the algorithmic delay, the lower is the VAD performance. It may be beneficial to investigate for further optimisations on this aspect before an implementation is considered.
- The network contains a lot of inherent parallellism that has not been investigated in this work. Before an implementation this will have to be investigated for further optimisation.

Conclusion 12

This thesis has been on VAD with focus on a potential real-time implementation in mind. To limit the scope of the project to match the time-frame, three research questions were formulated early which has then been the starting point of this work.

1. *Can we potentially increase the noise-robustness of a VAD without increasing its computational cost and latency to the execution-time?*
2. *How will it affect the performance of the VAD if we allow it to use less future samples to generate a VAD output and thus decrease the algorithmic delay?*
3. *How can we reduce the computational cost of the VAD in a real time application while maintaining performance?*

The thesis has been carried out in three parts. Below will be concluded on the work done in these three parts separately

Part 1

In part 1 the scope of the project, RTX' interests and the three research questions around which the work revolved was introduced. As the research questions aim to further investigate and improve the performance of a VAD algorithm under different conditions, it was decided to use an already existing method as a framework for this work. As framework was chosen a method proposed in [2] which has proven state-of-the-art performance on the Aurora-2 database. This method resorts to a FCN which essentially is a deep neural network that consists only of convolutional layers and activation functions. In order to understand this method in depth, some relevant theory on convolutional neural networks was presented in chapter 2.

In the paper where the method was originally published, only a few design choices were presented. Therefore it was up to the author of this thesis to decide on a series of design choices. Among these are which optimiser to use, the learning rate, the number of training epochs etc. These choices were made based on the theory presented in chapter 2 and can be seen in the beginning of chapter 3.

Having settled on some design choices, the last work in part 1 of the project was to set up a simulation environment implementing the network of [2]. As no source code was available this had to be built from scratch. The approach towards building this simulation environment is presented in chapter 3. After setting up the simulation environment the results of the original paper was attempted reproduced. Some slight deviations between the original results and the reproduced results was found, however as the difference was not deemed significant it was chosen to carry on using the implementation as is.

Part 2

Part 2 of this thesis revolved around research question 1 and 2. As part of this work was submitted a paper to *Interspeech 2022* which can be seen in chapter 4. The paper is used as the main reporting format of part 2, however some more detailed discussions that were left out of the paper due to lack of space are provided afterwards in this report.

The three key points of the paper are:

- By introducing adversarial multi-task learning to the network training the performance can be increased without introducing any cost in run-time. (Research question 1)
- When reducing the algorithmic delay of the network the performance of the VAD is decreased, however not significantly. It was possible to reduce the algorithmic delay from 398 ms to 23 ms with approximately 7% degradation in terms of AUC. (Research question 2)

Training and testing has been done on both the Aurora-2 and the TIMIT databases. It was found that the method is prone to overfitting on both databases due to the characteristic structure of the files and the large filter sizes of the network. Therefore it was chosen to concatenate files such that the network was learning characteristics of speech instead of the structure of the files. Here it was found that the performance can be further increased by concatenating only files of similar noise type and SNR level.

Part 3

The last part of this work focused on research question 3. Whereas the first two parts have been considering the network on an algorithmic level without considering an implementation, this third part changes the scope of the work and consider exactly this aspect. In chapter 8 is introduced some methods that can potentially be useful in terms of a real-time implementation on a device with only limited memory and computational power. The two concepts considered are pruning and quantization. Pruning is the process of reducing the number of parameters in the network while still maintaining performance - thus leading to lower memory requirements. Quantization is the process of converting to fixed-point representation and lowering the number of bits used to represent numbers. Both of these concepts are based on up-to-date papers proposing methods which have proven efficient on other neural networks.

In chapter 9 these methods are applied on top of the networks resulting from part 2. Both the network with an algorithmic of 398 ms and 23 ms are considered. Following the simulations, in chapter 10 is discussed if these methods are feasible optimisations before implementation on a DSP and an FPGA. In chapter 10 it is found that in order to fully utilise the optimisations presented in chapter 8 a customizable architecture like FPGA or ASIC is needed

Summary

Shortly summarised, the three research questions can be answered by:

1. By introducing adversarial multi-task learning under training, the performance of the VAD can be improved. Especially under noisy conditions.

2. The algorithmic delay of the network can be reduced by lowering the filter sizes. Especially those of the DB. When decreasing these filter sizes the VAD performance is decreased as well. It has been found that the algorithmic delay can be lowered from 398 to 23 ms while only degrading the performance by 7% in terms of AUC
3. The computational cost can be lowered by pruning the network and quantizing the parameters and feature maps to fixed-point representations of smaller bit widths. This however gives rise to discussions on how to align the data in memory, which architecture to implement the network on etc.

12.1 Further work

Further works on this project will include additional considerations on lowering the latency while maintaining performance. In particular it is of interest to investigate further alternative approaches towards reducing the algorithmic delay as also shortly covered in the discussion in chapter 11. The computational overhead that is also an important part of the latency has not been covered deeply in this work. Therefore to get a deeper understanding of the exact latency introduced by the network it is of interest to investigate this further.

Bibliography

[1] Z.-H. Tan, A. K. Sarkar and N. Dehak, 'Rvad: An unsupervised segment-based robust voice activity detection method,' *CoRR*, vol. abs/1906.03588, 2019. [Online]. Available: http://arxiv.org/abs/1906.03588.

[2] C. Yu, K.-H. Hung, I.-F. Lin, S.-W. Fu, Y. Tsao and J.-w. Hung, *Waveform-based voice activity detection exploiting fully convolutional networks with multi-branched encoders*, 2020.

[3] R. Zazo-Candil, T. N. Sainath, G. Simko and C. Parada, 'Feature learning with raw-waveform cldnns for voice activity detection,' in *INTERSPEECH*, 2016.

[4] Y. Lee, J. Min, D. K. Han and H. Ko, 'Spectro-temporal attention-based voice activity detection,' *IEEE Signal Processing Letters*, vol. 27, pp. 131–135, 2020. DOI: 10.1109/LSP.2019.2959917.

[5] D. Rho, J. Park and J. H. Ko, *Nas-vad: Neural architecture search for voice activity detection*, 2022. DOI: 10.48550/ARXIV.2201.09032. [Online]. Available: https://arxiv.org/abs/2201.09032.

[6] D. J. B. Pearce and H.-G. Hirsch, 'The aurora experimental framework for the performance evaluation of speech recognition systems under noisy conditions,' in *INTERSPEECH*, 2000.

[7] I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[8] C. Nwankpa, W. Ijomah, A. Gachagan and S. Marshall, *Activation functions: Comparison of trends in practice and research for deep learning*, 2018. DOI: 10.48550/ARXIV.1811.03378. [Online]. Available: https://arxiv.org/abs/1811.03378.

[9] D. Choi, C. J. Shallue, Z. Nado, J. Lee, C. J. Maddison and G. E. Dahl, *On empirical comparisons of optimizers for deep learning*, 2019. DOI: 10.48550/ARXIV.1910.05446. [Online]. Available: https://arxiv.org/abs/1910.05446.

[10] G. Hinton. (). ' lecture 6a overview of mini-batch gradient descent,' [Online]. Available: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[11] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2014. DOI: 10.48550/ARXIV.1412.6980. [Online]. Available: https://arxiv.org/abs/1412.6980.

[12] J. Dellinger. (2019). 'Weight initialization in neural networks: A journey from the basics to kaiming,' [Online]. Available: https://towardsdatascience.com/weight-initialization-in-neural-networks-a-journey-from-the-basics-to-kaiming-954fb9b47c79.

[13] X. Glorot and Y. Bengio, 'Understanding the difficulty of training deep feedforward neural networks,' in *AISTATS*, 2010.

[14] K. He, X. Zhang, S. Ren and J. Sun, 'Delving deep into rectifiers: Surpassing human-level performance on imagenet classification,' *CoRR*, vol. abs/1502.01852, 2015. [Online]. Available: http://arxiv.org/abs/1502.01852.

[15] J. Huber, *Batch normalization in 3 levels of understanding*, 2020. [Online]. Available: https://towardsdatascience.com/batch-normalization-in-3-levels-of-understanding-14c2da90a338.

[16] W. Koehrsen, *Overfitting vs. underfitting: A complete example*, 2018. [Online]. Available: https://towardsdatascience.com/overfitting-vs-underfitting-a-complete-example-d05dd7e19765.

[17] M. cmglee, *Receiver operating characteristics*. [Online]. Available: https://en.wikipedia.org/wiki/Receiver_operating_characteristic.

[18] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai and S. Chintala, 'Pytorch: An imperative style, high-performance deep learning library,' in *Advances in Neural Information Processing Systems 32*, Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[19] J. S. Garofolo, L. F. Lamel, W. M. Fisher, J. G. Fiscus, D. S. Pallett and N. L. Dahlgren, *Darpa timit acoustic phonetic continuous speech corpus*, 1993.

[20] J. Barker, R. Marxer, E. Vincent and S. Watanabe, 'The third 'chime' speech separation and recognition challenge: Dataset, task and baselines,' in *2015 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*, 2015, pp. 504–511. DOI: 10.1109/ASRU.2015.7404837.

[21] M. Kolbæk, Z.-H. Tan and J. Jensen, 'Speech intelligibility potential of general and specialized deep neural network based speech enhancement systems,' *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 25, no. 1, pp. 153–167, 2017. DOI: 10.1109/TASLP.2016.2628641.

[22] Y. Shinohara, 'Adversarial Multi-Task Learning of Deep Neural Networks for Robust Speech Recognition,' in *Proc. Interspeech 2016*, 2016, pp. 2369–2372. DOI: 10.21437/Interspeech.2016-879.

[23] H. Yu, T. Hu, Z. Ma, Z.-H. Tan and J. Guo, 'Multi-task adversarial network bottleneck features for noise-robust speaker verification,' in *2018 International Conference on Network Infrastructure and Digital Content (IC-NIDC)*, 2018, pp. 165–169. DOI: 10.1109/ICNIDC.2018.8525526.

[24] P. Koch, *Reconfigurable and low energy systems 2021, lecture slides 1*, 2021.

[25] V. Sze, Y.-h. Chen, T.-J. Yang and J. S. Emer, 'Efficient processing of deep neural networks: A tutorial and survey,' *Proceedings of the IEEE*, vol. 105, pp. 2295–2329, 2017.

[26] S. A. Pillai and I. T.B., 'Factors causing power consumption in an embedded processor - a study,' 2013.

[27] S. Han, J. Pool, J. Tran and W. J. Dally, *Learning both weights and connections for efficient neural networks*, 2015. DOI: 10.48550/ARXIV.1506.02626. [Online]. Available: https://arxiv.org/abs/1506.02626.

[28] V. Eijkhout, R. van de Geijn and E. Chow, *Introduction to High Performance Scientific Computing*. Jan. 2016. DOI: 10.5281/zenodo.49897.

[29] G.-K. Ma and F. Taylor, 'Multiplier policies for digital signal processing,' *IEEE ASSP Magazine*, vol. 7, no. 1, pp. 6–20, 1990. DOI: 10.1109/53.45968.

[30] M. Horowitz, '1.1 computing's energy problem (and what we can do about it),' in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014, pp. 10–14. DOI: 10.1109/ISSCC.2014.6757323.

[31] D. D. Lin, S. S. Talathi and V. S. Annapureddy, *Fixed point quantization of deep convolutional networks*, 2015. DOI: 10.48550/ARXIV.1511.06393. [Online]. Available: https://arxiv.org/abs/1511.06393.

[32] S. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.

[33] S. Han, H. Mao and W. J. Dally, *Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding*, 2015. DOI: 10.48550/ARXIV.1510.00149. [Online]. Available: https://arxiv.org/abs/1510.00149.

[34] T. Zhang, Z. Lin, G. Yang and C. D. Sa, *Qpytorch: A low-precision arithmetic simulation framework*, 2019.

[35] *Da14195 smartbeat™ low-power peripheral audio solution*. [Online]. Available: https://www.dialog-semiconductor.com/products/wireless-audio/da14195.

[36] *Cadence hifi 3 dsp user manual*. [Online]. Available: https://www.manualslib.com/manual/1484529/Cadence-Hifi-3-Dsp.html.

[37] Altera. (). 'Cyclone v device handbook,' [Online]. Available: https://people.ece.cornell.edu/land/courses/ece5760/DE1_SOC/cyclone5_handbook.pdf.

[38] ——, (). 'Cyclone v device overview,' [Online]. Available: http://www.altera.com/literature/hb/cyclone-v/cv_51001.pdf.

[39] J. Zhao, H. Wu and S. Gu, 'Digital filter design for cpt atomic clocks and fpga realization,' *Journal of Convergence Information Technology*, vol. 7, pp. 97–105, Mar. 2012. DOI: 10.4156/jcit.vol7.issue4.12.

TIMIT dataset A

This section aims to describe the approach of adding noise to the TIMIT data set

A.0.1 Structure of train set and test set

The goal is to artificially add noise to clean speech from the TIMIT database.

A.0.2 Training set

The training set of TIMIT consist of 4620 spoken sentences. In the training set, each file are used only once. There are 6 different noise types, and it is desired to train it on SNR levels of:

- -5 dB
- 0 dB
- 5 dB
- 10 dB
- 15 dB
- 20 dB

Additionally, $\frac{1}{7th}$ of the files will be without noise. Thus each unique noise type and SNR level will be used for $\frac{4620}{6 \cdot 7} \approx 110$ files. All of these files are placed in the same directory, allowing for multi condition training:

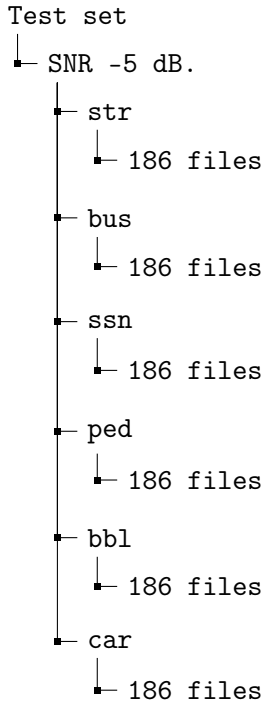
```
Training set
└─ 4620 unique files.  Equally distributed between the 7 noise types and 6 SNR levels
```

In the training set, each file is corrupted with a unique part of the noise file, such that no noise is seen more than once each epoch.

A.0.3 Test set and validation set

The test set contains 1680 spoken sentences. These will be split into a test set and a validation set using a $\frac{2}{3}$ and $\frac{1}{3}$ split. Thus 1120 files are used for test and 540 for validation. In both sets all files are used for each SNR level. The SNR levels are similar to those in the training set. Thus the files will be distributed equally between the 6 noise types. Thus each unique noise type and SNR level will be used for $\frac{1080}{7} \approx 160$ files for test and 80 for validation.

These files are placed in different directories as shown below, to allow for easy testing under different conditions:



The validation set has a similar structure with 80 files per folder instead of 160.

A.1 Approach for adding noise to the TIMIT dataset

A.1.1 Ensuring correct SNR

To ensure the noisy files end up having the desired SNR levels, the approach used is as follows:

To calculate the SNR of the files, the following equation is used:

$$\mathrm{SNR}_{\mathrm{dB}} = 20 \cdot \log_{10} \frac{\mathrm{speech}_{\mathrm{RMS}}}{\mathrm{noise}_{\mathrm{RMS}}} \tag{A.1}$$

Where the RMS is calculated as:

$$x_{RMS} = \sqrt{\frac{1}{N} \sum_{n=1}^{N} |x_n|^2} \tag{A.2}$$

However in order to ensure that the SNR represents only the speech active region of the files, only the parts which contains speech are used. Thus omitting the silent regions in the end and the beginning of the files. The audio files are labelled using the .WRD files, in which it is described what words are spoken at what time.

For finding the desired RMS of the noise signal, the following equation is used:

$$20 \cdot \log_{10} \frac{\text{speech}_{\text{RMS}}}{\text{y} \cdot \text{noise}_{\text{RMS}}} = \text{SNR}_{\text{dB}} \tag{A.3}$$

This is solved for y, where y is the scalar needed to multiply the noise signal to achieve the desired SNR level:

$$\text{noise}_{\text{scaled}} = \text{noise} \cdot \text{y} \tag{A.4}$$

Finally, the noise is simply added to the speech signal and the noisy signal is obtained:

$$\text{speech}_{\text{noisy}} = \text{speech} + \text{noise}_{\text{scaled}} \tag{A.5}$$

Additionally, when generating the noisy files the SNR level has been verified using the *snr* function in matlab.