# Deep Learning Based Adversarial Examples Detection for Speech Recognition

- A Comparative Study of Speech Processing Methods -

Master's Thesis
Group 19GR1070

Aalborg University
Signal Processing & Acoustics

AALBORG UNIVERSITY
STUDENT REPORT

**Signal Processing & Acoustics**
Fredrik Bajers Vej 7
9220 Aalborg Ø
http://www.es.aau.dk

**Title:**

Deep learning based adversarial example detection for speech recognition

- A comparative study of speech processing methods

**Theme:**

Master's thesis

**Project period:**

February 2019 - June 2019

**Project group:**

19GR1070

**Participants:**

Amalie Vistoft Petersen
Jacob Theilgaard Lassen
Sebastian Biegel Schiøler

**Supervisor:**

Zheng-Hua Tan

**Number of pages: 136 (total numbers incl. appendices)**
**Finished: June 5, 2019**

Abstract:

This project investigates the use of neural networks for detecting adversarial examples for speech recognition with five different feature extraction methods as input. These are STFT, MFCC, IMFCC, GFCC and IGFCC. Relevant theory in the areas of deep learning, adversarial examples and speech processing is examined, and a description is made of the available white box and black box datasets. A CNN model is implemented and evaluated w.r.t performance and robustness of the different feature extraction methods. This includes an investigation into how the performance is affected when only speech or nonspeech is present in the data. The addition of different types and amounts of noise in the data is investigated to determine how it affects the performance of the model. It is concluded that the CNN model is able to detect adversarial examples for speech recognition and the IMFCC and IGFCC feature extraction methods in general have highest accuracies. Furthermore the model is generally more robust to noise when its training set contains a wider range of noise types.

# Preface

This master's thesis was developed and written during the spring of 2019. The master's thesis was completed by three students: Amalie Vistoft Petersen, Jacob Theilgaard Lassen and Sebastian Biegel Schiøler. All studying on Aalborg Universities master program, Signal Processing and Acoustics, with a specialization in Signal Processing and Computing. References are denoted by numbers in order of appearance. Each number corresponds to a number in the bibliography. The bibliography can be found after the conclusion. Figures with no source are made by the authors.

The group would like to thank Saeid Samizade for help and guidance in the beginning of the project.

The project is divided into four main parts. The analysis, implementation and evaluation, discussion and conclusion, and appendices. The analysis explains and examines artificial neural networks, a general review of what adversarial examples are and an examination of the data that are used in the project, which evaluation methods are used for evaluating the experiments, and finally a problem description and problem statement are defined. The second part, implementation and evaluation, describes how the data for the project is handled, the evaluation of the used feature extraction methods together with the model. It investigates the influences of the speech and nonspeech that occurs in each example in the used data, and goes into the robustness of the feature extraction methods and the model in regards to noise. The third part contains the projects discussion and conclusion, and the fourth and final part contains the appendices containing description of used methods and tests results.

AAU, June 5th, 2019

Amalie Vistoft Petersen
<apet13@student.aau.dk>

Jacob Theilgaard Lassen
<jlasse14@student.aau.dk>

Sebastian Biegel Schiøler
<sschia14@student.aau.dk>

# Glossary

**AF** Activation Function.

**ANN** Artificial Neural Network.

**ANNs** Artificial Neural Networks.

**C&W** Carlini and Wagner.

**CNN** Convolutional Neural Network.

**CTC** Connectionist Temporal Classification.

**DCT** Discrete Cosine Transform.

**DFT** Discrete Fourier Transform.

**ERB** Equivalent Rectangular Bandwidth.

**FGSM** Fast Gradient Sign Method.

**GFCC** Gammatone Frequency Ceptral Coefficients.

**GPU** Graphics Processing Unit.

**IGFCC** Inverse Gammatone Frequency Ceptral Coefficients.

**IMFCC** Inverse Mel-Frequency Cepstral Coefficients.

**MFCC** Mel-Frequency Cepstral Coefficients.

**NN** Neural Network.

**NNs** Neural Networks.

**PR** Perfect Reconstruction.

**ReLU** Rectified Linear Unit.

**RMS** Root-Mean-Square.

**RNN** Recurrent Neural Network.

**SNR** Signal-to-Noise Ratio.

**STFT** Short-Time Fourier Transform.

**Tanh** Hyperbolic Tangent.

**VAD** Voice Activity Detection.

# Contents

# 1 | Introduction

Machine learning is a very popular tool with ongoing research and great practical potential in many areas. It has been found that machine learning models can be vulnerable to certain manipulated examples, also called adversarial examples [1]. These adversarial examples are very similar to a corresponding normal example with only small perturbations. However adversarial examples causes the model to classify differently and incorrectly. Especially Neural Networks (NNs) can be highly affected by adversarial examples [1].

There exist two types of adversarial attacks: Targeted and non-targeted attacks [2]. Targeted attacks attempts to get the model to predict a certain class. A real world example is when an autonomous car recognizes road signs. Here, it is possible to create targeted adversarial examples that can get a model to classify a stop sign as a 45km/h sign; robust to both variations in distance and angle [3]. Non-targeted attacks attempts to get the model to predict the class of the examples wrong without assigning a specific class to be predicted. Hence the only goal is to get the model to not predict the correct class. A real world example of this is in face-recognition. Here, the authors of [4] creates a special pair of printed eyeglasses that can make the model misclassify the users face as an arbitrary one, fooling state-of-the-art face-recognition algorithms.

Adversarial examples also exist in the audio domain [5]. Here an attack could be performed on devices such as Google Assistant [6] and Amazon Alexa [7]. Google Assistant is a virtual assistant, that is available on Google Home and mobile devices running Android [6]. Google Assistant can write emails, where the user dictates the content of the email. It is possible to open apps and do in-app purchases and search websites [8]. Google has announced that Google Assistant will increase support for purchasing products and sending money in the future. The Amazon Alexa also supports purchasing of items on Amazon through Alexa. Over 1 billion devices have voice-assisted access for consumers today. More and more possibilities for using one's voice to navigate devices will probably become available in the future. This underlines the dangers of successful adversarial attacks and motivates the ability to detect adversarial examples. In [5] the authors have created targeted adversarial examples which can fool a version of Mozilla's speech-to-text classifier called DeepSpeech [9]. These adversarial examples are almost not detectable by the human ear.

There are two types of targeted adversarial examples, white box and black box attacks. A white box attack is when the adversary has full access to the machine learning model, this includes its architecture, weights, biases, and so on. With a black box attack the adversary does not have access to the model and its parameters. Hence the adversary only have access to the output. In most cases the adversary would not have access to the model, however, it is still important to include both cases. Especially if you want to develop a model that can detect adversarial examples.

## 1.1 Initial Problem Description

The purpose of this project is to investigate the use of Artificial Neural Networks (ANNs) with different feature extraction methods for detection of adversarial examples for speech recognition. Research with a similar purpose have been performed by projects such as [10], [11] and [12]. This project will expand upon the work in [12]. The authors of [12] propose a Convolutional Neural Network (CNN) model to detect adversarial examples for speech recognition tasks. They use Mel-Frequency Cepstral Coefficients (MFCC) as a feature extraction method for the input to the CNN. They generate two datasets, a white box and black box dataset, containing both normal and adversarial examples. The authors show that it is possible to detect adversarial examples with their model and the feature extraction method MFCC.

The project will construct a similar CNN architecture as [12] and use the authors' generated datasets. Different feature extraction methods will be investigated to see how the performance of the model changes. Furthermore, the design, training, and evaluation of Artificial Neural Network (ANN)s will be investigated to ensure correct implementation and use of heuristics in order to accurately evaluate any new findings. The generation of adversarial examples is examined and the dataset is analyzed to yield a greater understanding of the difference between normal examples and adversarial examples. These insights will be used to motivate new investigation areas.

Therefore the following is researched:

- Chapter 2 - How are ANNs designed and trained, and what makes CNN different?
- Chapter 3 - What are adversarial examples, how are they created and what are the properties of the available data?
- Chapter 4 - What types of feature extraction methods are used in the project?
- Chapter 5 - Which metrics should be used to evaluate the CNN?

# Part I

# Analysis

# 2 | Artificial Neural Networks

It is important to get a deep understanding of Artificial Neural Networks in order to help detect adversarial examples. This chapter will describe the basics of ANNs, how they are trained, common problems for obtaining good solutions, and what common practices seem to mitigate the problems. In addition to this the chapter examines the Neural Network (NN) type CNN that is used in [12]. This examination includes the CNN's general structure together with a description of the convolution layer and pooling layer.

ANNs are made up of a series of layers with multiple artificial neurons [13]. There is an input layer, one or more hidden layers and an output layer as shown on Figure 2.1.

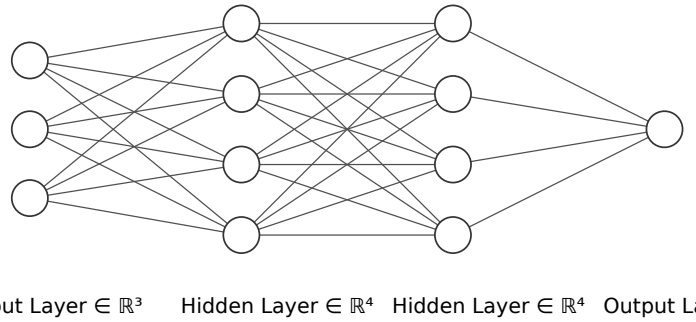



**Figure 2.1:** Artificial Neural Network with 3 inputs in the input layer, 4 neurons in the first hidden layer, 4 neurons in the second hidden layer, and 1 output in the output layer.

The circles represent neurons while the lines represent weighted connections. Each neuron has a series of weighted inputs and is said to 'fire' once a certain combination of inputs exceed a threshold. This is true for both the artificial neuron and the neurons in the brain [13]. Their difference comes from the fact that artificial neurons takes continuous real values as input while the real neurons inputs are binary. Figure 2.1 shows a fully connected network, meaning that each neuron in one layer has a weighted connection to all neurons/inputs in the previous layer.

## 2.1 Computing the Output of an Artificial Neural Network

Let the output $y_j$ of the $j$-th neuron with a series of inputs $x_i$ be defined as

$$y_j = g\left(\sum_i w_{i,j}x_i + b_j\right) \tag{2.1}$$

where $w_{i,j}$ is a weight attributed to each input, $b_j$ is the bias, and $g(\bullet)$ is called an Activation Function (AF).

This can be written in vector notation where the bold letters represent vectors and matrices as

$$\boldsymbol{y} = g\left(\bar{\boldsymbol{W}}\bar{\boldsymbol{x}}\right) \tag{2.2}$$

where $\bar{\boldsymbol{x}} = [1, \boldsymbol{x}]^T$ and $\bar{\boldsymbol{W}} = [\boldsymbol{b}, \boldsymbol{W}]$ to incorporate the biases directly into the weight matrix. The bar notation is dropped and $\boldsymbol{x}$, and $\boldsymbol{W}$ will contain the biases in future references.

The hidden layers will be denoted as $\boldsymbol{h}_k$ where the $k$ subscript indicate which hidden layer it is. Only the input layer will be referred to as $\boldsymbol{x}$ and $\boldsymbol{y}$ is only used for the output layer. The output of an ANN with $K$ hidden layers is defined as

$$\boldsymbol{y} = \boldsymbol{h}_K, \quad \boldsymbol{h}_k = g_k\left(\boldsymbol{a}_k\right), \quad \boldsymbol{a}_k = \boldsymbol{W}_k\boldsymbol{h}_{k-1}, \quad \boldsymbol{h}_0 = \boldsymbol{x}, \quad k = 1, ..., K. \tag{2.3}$$

A simple ANN with this notation is shown on Figure 2.2.

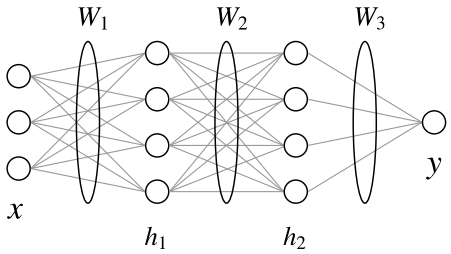



**Figure 2.2:** Artificial Neural Network with 2 hidden layers and the resulting 3 weight matrices.

The operations to go from input through all layers to the output layer is called forward propagation. The architecture of an ANN refers to the number of layers, the number of neurons in each hidden layer, and the operations to go from one layer to the next. It is possible to choose another operation than just a matrix multiplication and an activation function. Changing the architecture of a network thus refers to changing one or multiple of those factors.

The activation function must be non-linear in order for an ANN to be different from a linear transformation. This can be seen by taking a look at Equation (2.4) which is a simple ANN with one hidden layer.

$$\boldsymbol{y} = g_2(\boldsymbol{W}_2\boldsymbol{h}_1), \quad \boldsymbol{h}_1 = g_1(\boldsymbol{W}_1\boldsymbol{x}) \tag{2.4}$$

If $g(\bullet)$ is a linear function like $g(a) = ca$ then it follows that

$$\boldsymbol{y} = c\boldsymbol{W}_2\boldsymbol{h}_1, \quad \boldsymbol{h}_1 = c\boldsymbol{W}_1\boldsymbol{x} \tag{2.5}$$

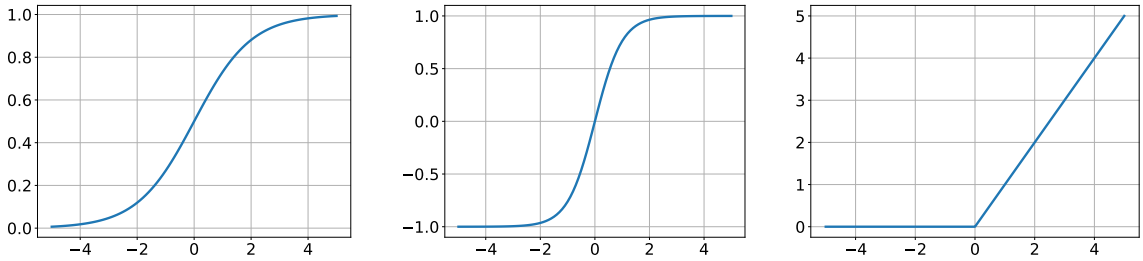
$$\boldsymbol{y} = c\boldsymbol{W}_2 c\boldsymbol{W}_1\boldsymbol{x} \tag{2.6}$$

$$\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} \tag{2.7}$$

which is just a linear transformation with $\boldsymbol{W} = c^2\boldsymbol{W}_2\boldsymbol{W}_1$. There are many different types of activation functions that can be used in an ANN and their merits will be described.

## Activation Functions

The activation functions that will be discussed are the Sigmoid, the Hyperbolic Tangent (Tanh), the Rectified Linear Unit (ReLU) and the Softmax activation functions. The three first mentioned AFs are seen in Figure 2.3.




**(a)** Sigmoid $g(a) = \frac{1}{1+e^{-a}}$

**(b)** Tanh $g(a) = \frac{e^{a}-e^{-a}}{e^{a}+e^{-a}}$

**(c)** ReLU $g(a) = \max(0, a)$

**Figure 2.3:** Three different activation functions.

The purpose of an AF is to introduce non-linearity in the hidden layers of the ANN and to perform predictions in the output layer [14]. Each AF has different merits but some common features have to apply to the AFs; they must be non-linear and differentiable [14]. The Sigmoid AF has the advantage of being bounded, producing only outputs between 0 and 1, which makes it applicable in the output layer for a classification task.

The Tanh AF is similar to the Sigmoid AF. One desired feature of the Tanh AF is that it is zero centred and bounded to output values between -1 and 1. This makes the Tanh more suited for negative inputs than the Sigmoid function because the Sigmoid function will make outputs near zero for highly negative inputs [14].

As shown in Figure 2.3c, the ReLU AF forces all input values less than zero to zero. An advantage of ReLUs is that it is extremely efficient to compute as the output can be found just be evaluating the sign of the input.

The Softmax AF outputs a probability distribution from an input vector of real numbers and it is defined as seen in Equation (2.8) [14].

$$g(a_i) = \frac{e^{a_i}}{\sum_j e^{a_j}} \tag{2.8}$$

It outputs a value between 0 and 1 for each output and the sum of all outputs equals one. This makes the Softmax AF suited for the output layer for classification problems with more than two outputs [14]. Here each output can be thought of as a probability that the input belongs in each class.

An ANN with non-linear activation functions allows for a non-linear transformation of the input to a new output and a shallow network with an infinite number of neurons can

in fact approximate any function perfectly due to the universal approximation theorem [15]. This approximation would not be perfect because it is not possible to use an infinite number of neurons in practice. Nonetheless, the ANN has shown to produce good results on complex tasks in practice [13].

The problem now is to find, what the weights should be in order to approximate the desired function. The weights can be learned in a supervised manner by giving the network inputs with a known desired output. The ANN should then learn to approximate the function that produced the desired output given said input in a process called *training* [13].

## 2.2 Training an Artificial Neural Network

The goal of training an ANN is to find the optimal weights $\boldsymbol{W}^*$ that produces the best approximation of the desired function. This can be done by formulating a loss function $E(\boldsymbol{W})$ that evaluates the distance between an output $\boldsymbol{y}$ and the known/desired output $d$. The optimal weights can then be found by minimizing the loss function such that

$$\boldsymbol{W}^* = \arg\min_{\boldsymbol{W}} E(\boldsymbol{W}). \tag{2.9}$$

The loss function is dependent on the task to be solved and should be defined accordingly. Detecting adversarial examples is a binary classification task. For binary classification the goal is to classify whether the input falls in one of two categories; 0 or 1. The loss function for binary classification is defined as

$$E(\boldsymbol{W}) = -\Big(d\ln(y) + (1-d)\ln(1-y)\Big) \tag{2.10}$$

where $y$ is an output between 0 and 1 and $d$ is either 0 or 1 depending on the actual category of the input [13].

Minima and maxima of the loss function can be found by finding the places where the gradient is 0 i.e. $\nabla E(\boldsymbol{W}) = 0$. This is a necessary condition for an optimal point but can be fulfilled by any flat point of the loss function. For convex problems this is an optimal point but most real-life complex problems are not convex in nature [13]. These flat points can be found using gradient descent. In gradient descent, the gradient of the loss function is computed and the weights are updated by taking a step in the opposite direction. The weights of each layer can then be updated as

$$\boldsymbol{W}_k^{t+1} = \boldsymbol{W}_k^t - \gamma \nabla_{\boldsymbol{W}_k^t} E(\boldsymbol{W}^t), \quad \gamma > 0 \tag{2.11}$$

where $t$ indicates the iteration and $\nabla_{\boldsymbol{W}_k^t} E(\boldsymbol{W}^{\boldsymbol{t}})$ is the gradient of the loss function with respect to the weights in layer $k$ at iteration $t$. The step size or learning rate $\gamma$ then determines the distance to go in the gradient direction. The learning rate should not be too small as it will increase the time it takes to find the optimal weights. Setting it too large can also lead to slow learning or even divergence.

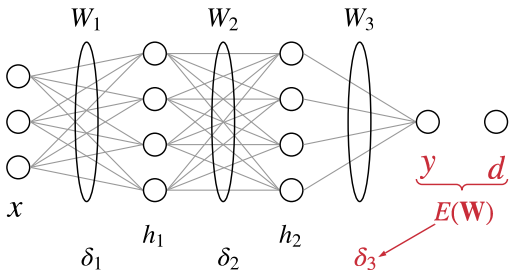
## Backpropagation

The algorithm used to compute the gradients with respect to the weights is called *backpropagation* and is computationally simple [13]. With forward propagation defined in Equation (2.3), backpropagation is defined as

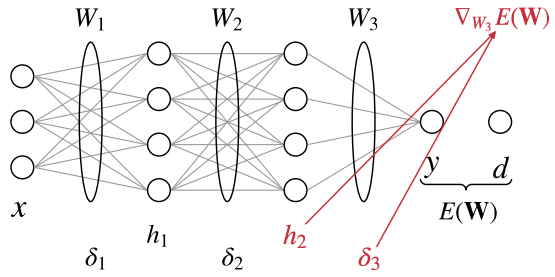
$$\nabla_{\boldsymbol{W}_k} E(\boldsymbol{W}) = \boldsymbol{\delta}_k \boldsymbol{h}_{k-1}^T, \quad \boldsymbol{h_0} = \boldsymbol{x} \tag{2.12a}$$

$$\boldsymbol{\delta}_k = g_k'(\boldsymbol{a}_k) \odot \begin{cases} \nabla_{\boldsymbol{y}} E(\boldsymbol{W}), & \text{if } k \text{ is an output layer } (k = K) \\ \boldsymbol{W}_{k+1}^T \boldsymbol{\delta}_{k+1}, & \text{otherwise} \end{cases} \tag{2.12b}$$

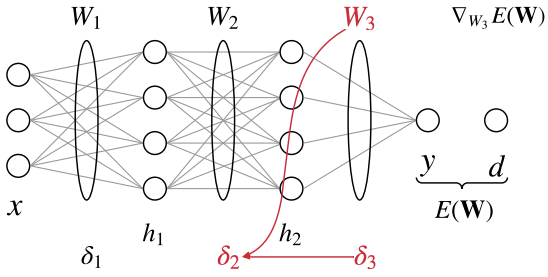
where $\odot$ is the element-wise multiplication and $g_k'(\boldsymbol{a}_k)$ is the derivative of the activation function with respect to $\boldsymbol{a}_k$. First the gradient of the loss function with respect to the output is computed in order to find $\boldsymbol{\delta}_K$. Then the gradient of the loss function with respect to $\boldsymbol{W}_K$ is computed. Before updating the weights using Equation (2.11), $\boldsymbol{\delta}_{K-1}$ is computed using $\boldsymbol{W}_K$ and $\boldsymbol{\delta}_K$. Then the weights can be updated and the gradient of the loss function with respect to $\boldsymbol{W}_{K-1}$ can be found. This process iterates backwards through the layers from output to input and updates the weights throughout. The process is shown on Figure 2.4.



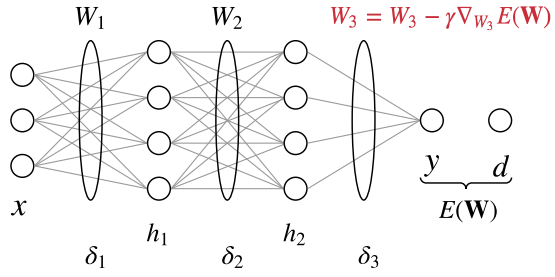

**(a)** Compute the loss and its gradient with respect to $y$ and find $\boldsymbol{\delta}_K$ using Equation (2.12b).




**(b)** Compute gradient of the loss with respect to $\boldsymbol{W}_k$ using Equation (2.12a).




**(c)** Compute $\boldsymbol{\delta}_k$ using Equation (2.12b).




**(d)** Update the weights using Equation (2.11) and to go **(b)**.

**Figure 2.4:** Diagrams showing the process of the backpropagation algorithm.

## Weight Update Methods

There are multiple options for updating the weights, as there are multiple data samples in the training set. Let $\mathcal{T}$ be a labeled training set with $N$ data samples of $\boldsymbol{x}$ and $d$ as

$$\mathcal{T} = \{(\boldsymbol{x}^i, d^i)\}_{i=1,\dots,N}, \tag{2.13}$$

then it is possible to compute the sum of the gradients over all samples in the training set. This is called batch gradient descent and is defined as

$$\boldsymbol{W}_k^{t+1} = \boldsymbol{W}_k^t - \gamma \sum_{(\boldsymbol{x},d)\in\mathcal{T}} \nabla_{\boldsymbol{W}_k^t} E(\boldsymbol{W}^t) \tag{2.14}$$

such that the weights are updated once every time the gradients of all data samples have been used (one *epoch*) [13, Chapter 8].

Another option is stochastic gradient descent which updates the weight after computing the gradient of each data sample [16]. Stochastic gradient descent is defined as

$$\boldsymbol{W}_k^{t+1} = \boldsymbol{W}_k^t - \gamma \nabla_{\boldsymbol{W}_k^t} E^i(\boldsymbol{W}^t), \quad i = t - N\left\lfloor \frac{t}{N} \right\rfloor \tag{2.15}$$

where $i$ is determined using the modulo operation. The loss $E^i(\boldsymbol{W})$ is computed using the $i$-th data sample in a shuffled training set $\mathcal{T}$. After every epoch $\mathcal{T}$ is shuffled. The weights are thus updated $N$ times per epoch. The advantage of stochastic gradient descent is the frequent weight updates but it has a lower chance of converging than batch gradient descent [13, Chapter 8].

A final option is mini-batch gradient descent which combines the two previous approaches. It splits the training set into subsets $\mathcal{B}_j$ of size $|\mathcal{B}_j| = B$ and then updates the weights using the sum of the gradients in each subset [13, Chapter 8]. It thus uses batch gradient descent for the gradients in each subset/batch and stochastic gradient descent for the sum of gradients for all subsets/batches. Mini-batch gradient descent is defined as

$$\boldsymbol{W}_k^{t+1} = \boldsymbol{W}_k^t - \gamma \sum_{(\boldsymbol{x},d)\in\mathcal{B}_j} \nabla_{\boldsymbol{W}_k^t} E(\boldsymbol{W}^t), \quad j = t - \frac{N}{B}\left\lfloor \frac{t}{\frac{N}{B}} \right\rfloor \tag{2.16}$$

where $j$ indicates the current mini-batch. In mini-batch gradient descent the weights are updated $N/B$ times per epoch. This means more frequent updates than batch gradient descent and a higher probability of convergence than stochastic gradient descent.

While batch gradient descent guarantees convergence under certain assumptions and the other two converges with high probability [13, Chapter 8] there is no guarantee that the local minima found is a good solution nor that it is able to find a minima in finite time. There is also no guarantee that the network performs well for data that is not available in the training set $\mathcal{T}$.

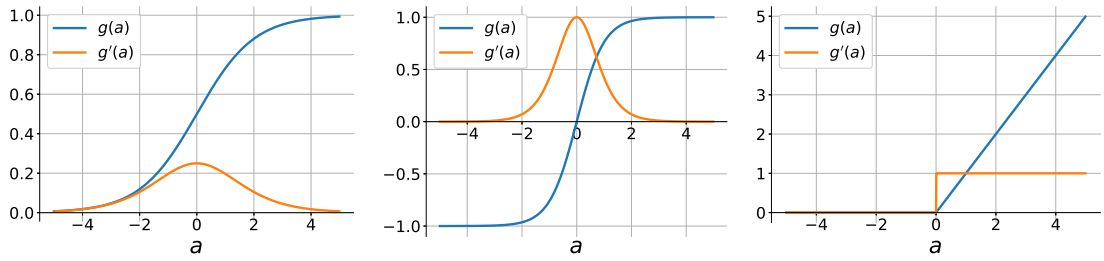
## 2.3 Problems and Practical Approaches for Training Deep Neural Networks

There are a series of challenges for obtaining good solutions (low cost function values) that also generalize well to unseen data. The challenges are especially prominent for deeper and more complex networks [13, Chapter 8]. These are the *gradient vanishing* and *gradient exploding* problems, an *underfitting/overfitting* model, and getting stuck at a poor solution such as a saddle point or a bad local minimum.

This section will describe these challenges along with some practical approaches to mitigate their effect.

### Gradient Vanishing Problem

The gradient vanishing problem occurs during backpropagation. As the gradient of the loss function is propagated backwards it is multiplied by the gradient of the activation function $g'_k(\boldsymbol{a}_k)$. For activation functions such as the sigmoid or the hyperbolic tangent, their gradients are in a range between 0 and 1 as shown in Figure 2.5. This means the gradient of the loss with respect to the weights will decrease as it propagates backwards through the layers. The result is that the weights of the first layers take very small steps towards a better solution i.e. they learn very slow [13].




**(a)** Sigmoid **(b)** Tanh **(c)** ReLU

**Figure 2.5:** Derivatives of different activation functions.

This explains why the ReLU AF is often used for hidden layers [13] as the gradient is always either 0 or 1. This means the activated units propagates the gradient backwards without diminishing it and thus helps mitigate the gradient vanishing problem.

### Gradient Exploding Problem

In NNs with many layers there are often regions with extremely large drops in the loss function due to small changes to the weights [13, Chapter 8]. These steep regions (cliffs) result in gradients with a very large magnitude. As a result the weights will perform a

very large step in the negative gradient direction. This is the gradient exploding problem and is a problem because the very large step could potentially overshoot a region of very good solutions near the cliff. An example of this is showed on Figure 2.6 where the gradient takes small steps towards a minima. Once it hits the cliff in the loss function the magnitude of the gradient will be so large, that the next step causes the weights to update in such a manner that it completely overshoots the minima right next to the cliff.

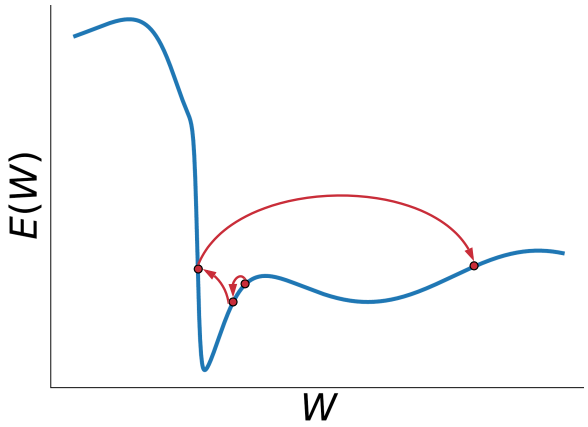



**Figure 2.6:** Simplistic example of the exploding gradients problem. The red dots represent different iterations of a gradient descent algorithm. The red lines represent the step taken from one iteration to the next.

The simple solution to this problem is to limit the gradient magnitude using *gradient clipping*. This is done as the gradient specifies the optimal direction but not the optimal distance. By setting a maximum to the gradient norm, it will avoid massively overshooting regions near cliffs [13, Chapter 8].

## Overfitting and Underfitting

In machine learning the main goal is to find a model that approximates a function well. This function is approximated with the ANNs using a training set $\mathcal{T}$. It is desired that the model not only approximates the function for the data available in the training set but also generalizes for unseen data. There are then two different objectives; decrease the error on the training set, and decrease the generalization error on previously unseen data. It is common to split the available data into a training set $\mathcal{T}$ and a test set $\mathcal{S}$ such that the networks ability to generalize can be evaluated fairly [13, Chapter 5].

During the training of a network the loss/error on the training set will decrease with more iterations. Underfitting is the problem when the network is unable to reach a sufficiently low cost value on the training set [13, Chapter 5]. This could be if the network does not train long enough or it is not complex enough to be able to describe the function it

attempts to approximate. Overfitting, however, is when the network trains too much and reaches very low cost values on the training set but not on the test set [13, Chapter 5]. This means that the network learns some small details specific to the training set that does not generalize well on the test set. The trade-off between overfitting and underfitting is shown on Figure 2.7, where the red line marks the optimum.

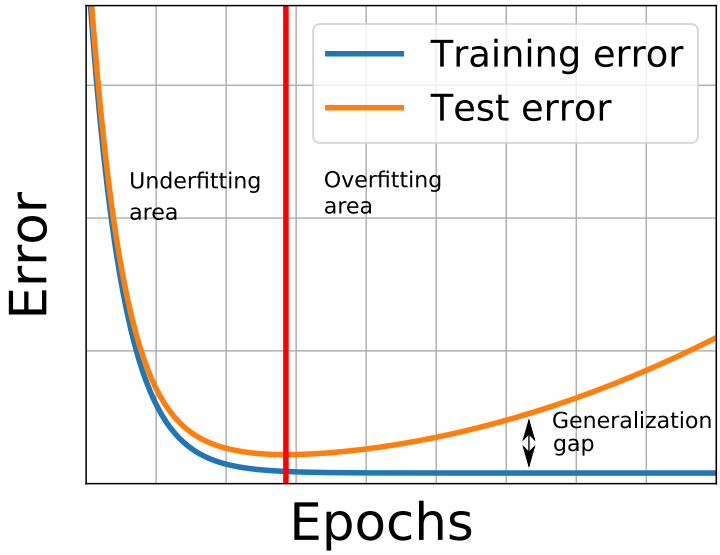



**Figure 2.7:** Trade-off between overfitting and underfitting.

Underfitting is a direct result of a lack of training time or data, or capacity of the ANN. Therefore it can be mitigated by increasing the amount of training data, increasing the number of iterations, or modifying the network architecture.

The problem of overfitting is fought with *regularization*. Regularization is a term that is added to the loss function explicitly or implicitly in order to limit or restrict the capacity of a network such that the network has a harder time to learn features specific to the training set. Formally, regularization is defined as

$$\hat{E}(\boldsymbol{W}) = E(\boldsymbol{W}) + \alpha\Omega(\boldsymbol{W}) \tag{2.17}$$

where $\Omega(\boldsymbol{W})$ is the regularization function and $\alpha$ determines the ratio between the original loss function and the regularization term. The regularization function can be expressed explicitly by defining the function and using it in the loss function as shown in Equation (2.17). It can also be expressed implicitly through various restrictions to the weights of certain layers in the network. Weight decay is one such regularization technique that changes the loss function to discourage weights with extremely large values. A 1-dimensional example of weight decay as a regularization technique is shown on Figure 2.8.

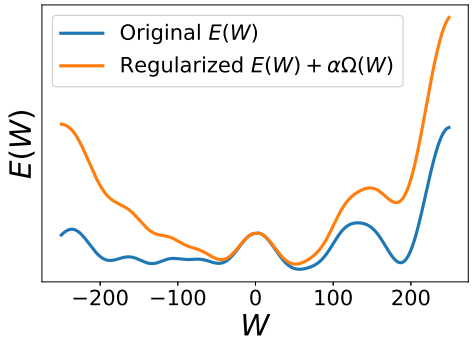



**Figure 2.8:** Weight decay as regularization.

One of the most common ways to fight overfitting through an implicit regularization function is with *early stopping* [13]. Early stopping means that the network stops training once the error on unseen data stops decreasing. This fixes the problem of training for too many iterations to the point where the error on the test set would start to increase. In order to still perform a fair evaluation of the network on unseen data, a third set of data is used; the validation set $\mathcal{V}$. The network is then trained on the training set $\mathcal{T}$ and during training the performance is evaluated on the validation set $\mathcal{V}$. The training is then stopped early when the error on the validation set stops decreasing/starts increasing. Alternatively the network weights can be saved for many, if not all, iterations and after training the final network will have the weights that resulted in the lowest validation error. Then the network can be evaluated on the test set $\mathcal{S}$ which contains data unseen to the network and should provide a good estimate for the network performance in a final real-life solution. When evaluating an implementation through iterative trial and error testing, it is necessary to have both a validation and test set. This is crucial due to the possibility of generalizing one's model when changing parameters between tests. If the data is only split into training and test, then it is possible to overfit the model changes/hyperparameters to the test set, see section 2.3. If the validation set is used to evaluate performance in between trials, then the test set can be used for testing without developer bias.

## Flat Regions, and Poor Solutions

Another problem that is often encountered when training ANNs is regions where small changes to the weights result in a constant loss function value [13, Chapter 8]. This means that the gradients are very small or 0 and the learning process is very slow. These regions will be referred to as flat regions. The flat regions are a problem if the value of the loss function is not sufficiently low to solve the task. There is also the possibility of getting stuck in a local minima with a higher loss than the global minima.

There are two main options for fighting this problem [13, Chapter 8]. First is to change the network architecture or apply a form of regularization in an attempt to reduce the

number of flat regions in the loss function. The second is to modify the gradient descent algorithm. A commonly used modification is to add momentum [17] to the gradient such that will continue to take larger steps if the previous gradients had a high norm [13, Chapter 8]. Similarly it is possible to adopt an adaptive learning rate to facilitate faster learning in flat regions. Adam [18] is a popular adaptive learning rate method that also utilizes momentum [13, Chapter 8].

These options do not solve the problem as this is an inherent problem to non-convex optimization. There is no guarantee that the optimization algorithm will converge to a global minima or even to a good solution. But momentum and adaptive learning rates seem to help nonetheless.

Furthermore since gradient descent is an iterative method, it will depend on initial values to the weights. For some initial values it will converge to a good solution but for others it will converge to a poor solution if it converges at all. It is possible that the choice of initial weights will result in numerical instability. Furthermore the initial weights can also affect the networks ability to generalize even if two different initializations converge to a solution with similar loss. There are thus a series of heuristic initialization strategies to try to avoid initializations that lead to instability. One such method is the He initialization which is meant for layers with a ReLU activation function [19]. Another method is the Xavier initialization which is meant for linear or Tanh activation functions [20]. Both methods makes the initial weights smaller for layers with more input units.

## 2.4 Convolutional Neural Networks

The same NN type and architecture used in [12] is used in this project. This is the NN type called CNN. This section will describe the basics of CNNs including its general structure along with a description of the CNNs convolution layer and pooling layer.

The CNN is one possible type of neural network model that can be used for applications which involve speech and it is a very strong form of implicit regularization. This claim is supported by studies found in [21], [22] and [12]. In general the CNN is used in areas such as computer vision. This is due to its specialization in handling data which has a grid-like topology, such as image data [13, Chapter 9]. When handling images or other similar data structures, ANNs only using fully connected layers quickly escalate in size and can require excessive amount of time to train [23]. Here CNN can reduce the parameters that are required to set up the model decreasing the complexity. [24]

A simplified illustration of a CNN is given in Figure 2.9.

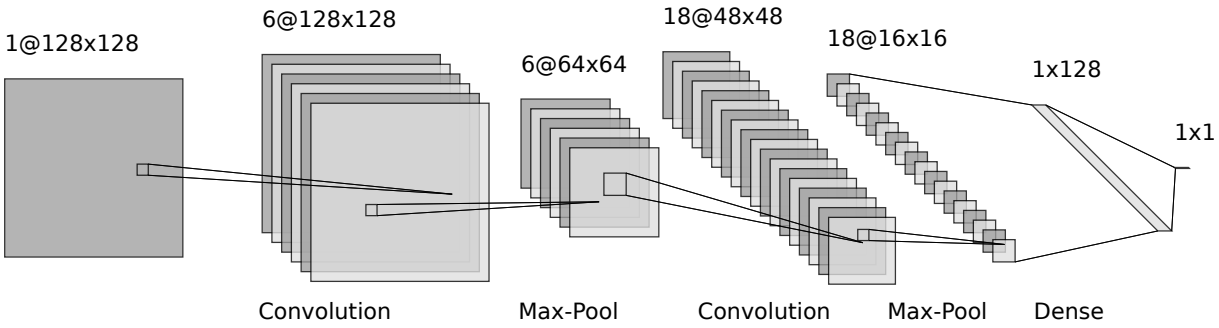



**Figure 2.9:** An simplified illustration of a CNN.

In Figure 2.9 an example is shown of how a CNN can be build with its sequence of different operations. From the input the first operation to be performed is convolution. Usually a non linear operation is performed with the convolution operation, such as ReLU or Tanh. A description of these are found in section 2.1. After the convolution operation a pooling operation can be performed. However, it is not required to have a pooling operation directly after a convolution. When creating a CNN it is possible to have as many convolution and pooling operations as found necessary. After these operations the CNN can utilize a fully connected layer (Dense) for classification, this has been described in section 2.1.

## Convolution

The convolutional layer extracts features from its input. This is achieved by using kernels. Each kernel is moved across its input with a specific distance, also called a stride. As the kernel is moved across the input an element wise multiplication between the input and the kernel is computed. The element wise multiplication is added together and the calculated scalar is placed in a matrix called the feature map. A small simplified example of the descriptions above is given in Figure 2.10. [25]

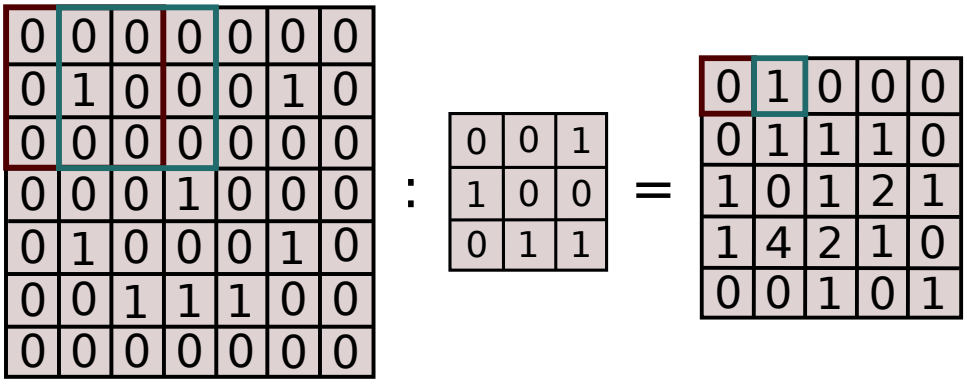



**Figure 2.10:** In this example the input size is 7x7, kernel size 3x3 with a stride of 1, resulting in an feature map of 5x5. The : operation denotes the Frobenius inner product between the highlighted area on the left and the matrix on the right.

Each kernel computes their own corresponding feature map. So by using different kernels it is possible to extract different features from the input. The value of the kernels are learned during the training of the CNN using backpropagation. The size and dimension of the feature map is dependent on three user determined parameters; depth, stride and zero-padding. [25]

**Depth** represents the number of kernels used in the convolution. Each kernel is different and therefore creates a different feature map. In Figure 2.9 it can be seen that the CNN has a depth of six, i.e. using six kernels in the first hidden layer. The more kernels used the more information is extracted from the input. However, more weights needs to be trained which increases the complexity of the model.

**Stride** is the number of entries the kernel move. When the stride is one the kernel moves one entry at a time. The higher the stride the smaller the feature map and vice versa.

**Zero-padding** applies zeroes to the border of the input. This enables more size control of the feature map.

Equation (2.18) shows how the width and height of the feature map can be calculated [26].

$$\begin{aligned} F_w &= \frac{(C_w - K_w + 2P)}{S_w} + 1 \\ F_h &= \frac{(C_h - K_h + 2P)}{S_h} + 1 \end{aligned} \tag{2.18}$$

Where $F_w$ and $F_h$ are the width and height of the feature map, respectively. The $C_w$ and $C_h$ are the input width and height. $K_w$ and $K_h$ are the width and height of the kernel, P is the zero padding and S is the stride.

The number of weights and biases in each convolution layer is determined by the number of kernels and size of kernel. Two kernels $\kappa \in \mathbb{R}^{C \times H \times W}$ with depth 3 (channels), height 3, and width 3 will give 54 weights without bias. Each kernel depth operates on the same depth in the input. Two kernels results in the output feature map having a depth of 2. If the input depth is 32 and the desired dimension of the output feature map has a depth of 1 then a kernel would have dimension $\kappa \in \mathbb{R}^{32 \times H \times W}$.

In Equation (2.19) a mathematical description of the output of a convolution layer is shown.

$$\boldsymbol{h}_k = g_k(\boldsymbol{a}_k), \quad \boldsymbol{a}_k = \boldsymbol{\kappa}_k \otimes \boldsymbol{h}_{k-1}, \quad \boldsymbol{h}_0 = \boldsymbol{x} \tag{2.19}$$

Where $\kappa_k$ is the kernel for each layer, $h_k$ is the generated feature map and $g_k(\bullet)$ is an AF.

## Pooling

Pooling is used for the reducing feature map size and typically it is used after the convolution. Various methods can be used when utilizing pooling, such as max, average and sum pooling. In Figure 2.11 an example is shown of max pooling.

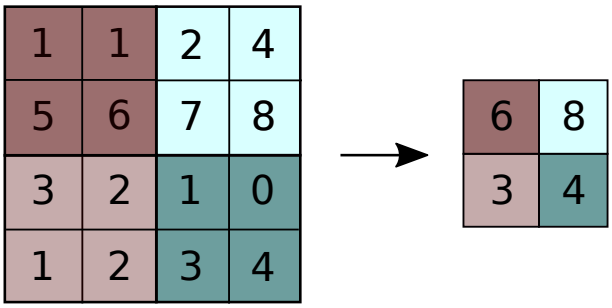



**Figure 2.11:** A simplified example of max pooling [27].

In Figure 2.11 the max pool kernel with size $2 \times 2$ and a stride of two. Max pooling selects the largest value and inserts it into a new matrix. If pooling is used it is applied to each feature map received from the convolution operation, resulting in new feature maps with smaller dimension but with the same depth as the previous output from the convolution operation. [25]

A property of pooling is the reduction of output size, while still preserving important feature information. However some information is lost, such as the exact occurrence of the information. By reducing the output size both computations and the number of parameters in the model is decreased. [27]

# 3 | Adversarial Examples

Neural networks produce state-of-the-art results in numerous domains such as image classification [28, 29] and speech recognition [9]. One concern with neural networks is its vulnerability to adversarial examples. This section aims to explain the concept of adversarial examples, followed by a general description of how these are generated. The project uses the two generated datasets from [12]. An overall description is given of the two methods used to generated the datasets and the examples in the two datasets are analysed. This should give an increased understanding into the two datasets individual characteristics.

The concept of adversarial examples is, in the case of images, to manipulate the original image to the degree that a human would not be able to tell the difference between the original and the manipulated image, but just enough to make a neural network wrongly classify the manipulated image. In [30] an example of such an adversarial image is constructed and it can be seen in Figure 3.1.

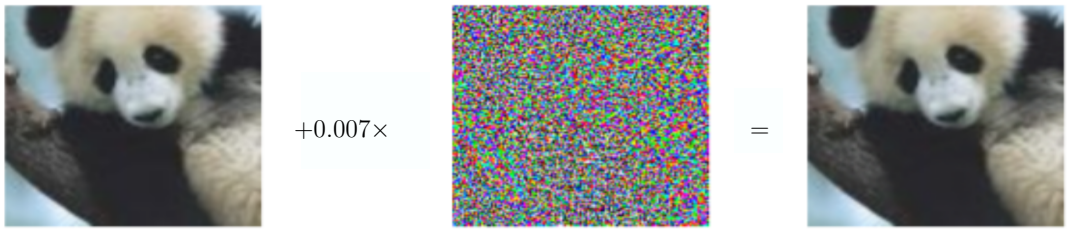



**Figure 3.1:** An adversarial example $\boldsymbol{x}'$ consists of an input image $\boldsymbol{x}$ and a small perturbation $\boldsymbol{\eta}$ [30].

The two images $\boldsymbol{x}$ and $\boldsymbol{x}'$ are applied to GoogLeNet [31], where it is seen that the image $\boldsymbol{x}$ is classified as a panda with 57.7 % confidence [30]. By adding the perturbation $\boldsymbol{\eta}$ to $\boldsymbol{x}$, the manipulated image $\boldsymbol{x}'$ is classified as a gibbon with 99.3 % confidence, although a human would still classify the image as a panda.

In the audio domain adversarial examples pose a real threat to devices using speech recognition systems as user interface. Systems and applications such as Google Assistant and Siri are vulnerable to audio adversarial attacks because an attacker could for example access malicious websites through Siri without the user knowing it. An adversarial example in the audio domain is seen in Figure 3.2.

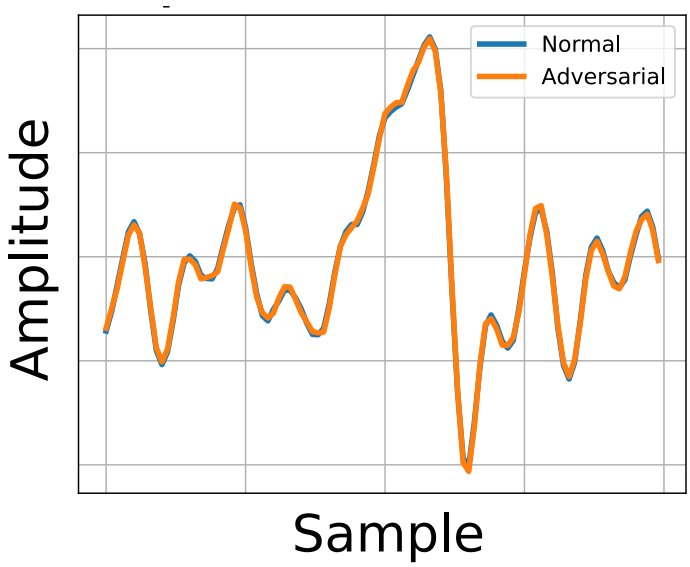



**Figure 3.2:** Audio adversarial example.

Adversarial example attacks can be both non-targeted and targeted. A non-targeted attack will try to make a neural network make the wrong classification where a targeted attack will try to make the neural network classify the adversarial example to a specific label decided by the attacker. Figure 3.2 is an example of a targeted attack. Figure 3.2 shows a part of an audio file where the normal audio signal is the phrase *"So now I fear nothing because it was those omens that brought you to me"* (blue) and the target is *"I need a reservation for sixteen people at the seafood restaurant down the street"* (orange).

The adversarial examples seen in Figure 3.2 is generated using the method described in [5]. The authors of [5] showed that it was possible to make a targeted adversarial examples in the audio domain which can attack the DeepSpeech speech-to-text classifier [9] with a 100 % success rate. In [5] the adversarial example were not robust to being played over the air. However, the work done in [32] show that targeted adversarial examples played through air is possible. So far the majority of the work regarding adversarial examples has focused on images but in recent years several methods for generating audio adversarial examples have been developed. Some methods for generating adversarial examples in the image domain can also be used for generating adversarial examples in the audio domain. Although some lessons from adversarial examples in the image domain do convert to the audio domain there are also many properties which do not convert. As an example [11] found that using input transformation as defense against adversarial attack does work in the image domain but not in the audio domain.

Some research in the area of detection adversarial examples in the speech recognition domain are [10], [11] and [12]. The authors of [10] floods an audio signals with random

noise on certain frequency bands in order to detect adversarial examples. They build on the idea that speech classifiers are robust to natural noise but adversarial examples are not. They find that based on observations of model robustness in regards to noise it takes less noise to change the models prediction of a class when the example is adversarial than when it is not. By using this observation they can calculate a score describing how much random noise needs to flood the signal for the model to change its prediction, and thus they are able set a threshold for detection if the example is adversarial or not.

The authors of [11] investigates if by exploiting temporal dependency it is possible to detect adversarial examples. The authors split up their normal and adversarial examples into $k$ segments send it through a speech recognizing model together with the full examples. They then select the $k$-th segment of the transcribed result of the full example and compare its the temporal dependency distance to the $k$-th segment from the split example. For a normal example the temporal dependency distance should be small and for targeted adversarial examples temporal dependency may not be preserved as well. The authors find that temporal dependency can be exploited for detecting adversarial examples and it is resistant to certain adaptive attacks.

## 3.1 Generating Adversarial Examples

Several methods of generating adversarial examples exist. In general the generation of adversarial examples can be described as an optimization problem as

$$\begin{aligned} \min_{\boldsymbol{x}'} \quad & \| \boldsymbol{\eta} \| = \| \boldsymbol{x}' - \boldsymbol{x} \| \\ \text{subject to} \quad & f(\boldsymbol{x}') = y' \\ & f(\boldsymbol{x}) = y \\ & y \neq y' \\ & \boldsymbol{x}' \in [-1, 1] \end{aligned} \tag{3.1}$$

where $\boldsymbol{\eta}$ is the perturbation added on $\boldsymbol{x}$, $f(\bullet)$ is a trained neural network model, $\boldsymbol{x}$ is an input to the model, $y$ is the predicted output of the model $f(\bullet)$, and $'$ marks the corresponding adversarial examples [2]. Equation (3.1) illustrates the general approach to generating adversarial examples but in specific generation methods both the objective function and the constraints can vary.

There exist two types of adversarial examples, white box and black box attacks.

**White box attacks:** The attacker has full knowledge of the victim model, which includes knowledge of the model architecture, model weights, etc. One of the advantages of using white box attacks is that the attacker is able to compute gradients to the input based on the output [2].

**Black box attacks:** The attacker has no knowledge of the victim model. The attacker only knows the input to the model and the corresponding output of the model. One way

of generating black box adversarial attacks is by making a white box attack on a different model and then use this former white box attack as a black box attack on the actual victim model [2].

The datasets from [12] use the Carlini and Wagner (C&W) attack [5] and Alzantot method [33] for the white box and black box dataset respectively. These are also the methods used to generate adversarial attacks in [10, 11].

## Carlini and Wagner's Attack

Generating adversarial examples in the audio domain does have other challenges than in the image domain. One of these challenges is that the adversarial example has to be adversarial through time. C&W has stated that:

*"targeted attacks seem to be much more challenging when dealing with speech recognition systems than when we consider artificial visual systems."*

— C&W [5]

In [5] C&W presents a method for generating targeted adversarial examples in the audio domain. This method is using the Connectionist Temporal Classification (CTC) which is also used for training many speech recognition networks. C&W uses the CTC loss function for generating adversarial examples in the audio domain. C&W uses the gradient of the loss function with respect to the perturbation $\boldsymbol{\eta}$ in order to minimize $\boldsymbol{\eta}$. This requires full knowledge of the victim model in order to backpropagate the gradient all the way back to the input.

Using this method, adversarial examples on DeepSpeech can be made with a mean perturbation of $-31$ dB between the original audio and the adversarial example. The adversarial examples seen in Figure 3.2 is made using C&W's method for audio. Many other methods can be used for generating audio adversarial examples. [5] found that the Fast Gradient Sign Method (FGSM) can be used for generating untargeted adversarial examples in the audio domain.

## Alzantot Method

Both FGSM and C&W's method are white box attacks. The Alzantot method [33] is a way of generating black box adversarial attacks in the audio domain. The Alzantot method is a genetic algorithm where no knowledge of the victim model is needed. The attack is focused only on the least significant bits of the input sample with the purpose of changing the human perception of the example as little as possible [33]. It generates a population of new samples with random perturbations to each and then ranks them based on the victim model's output for the target. A new population is then created from the highest ranked perturbations. This process is iterated until the victim model predicts the

sample as the target label or a set number of epochs is reached.

## 3.2 Dataset Description

It is important to get a good understanding of the data that is used in order to get key insights to better detect the adversarial examples. This is done by investigating the two datasets, white box and black box dataset, separately and looking into the examples in the time domain and in the frequency domain. It should be noted that the data generated for the project is made by a third party [12]. The method used to generate the targeted attacks are the method developed by Carlini and Wagner, and the Alzantot method described previously.

In both the white box and the black box datasets, three types of files exist; normal examples, original examples and adversarial examples. The original examples are used to generate the adversarial examples and are not used in either dataset. The datasets thus consist only of normal examples and adversarial examples.

### White Box Attacks

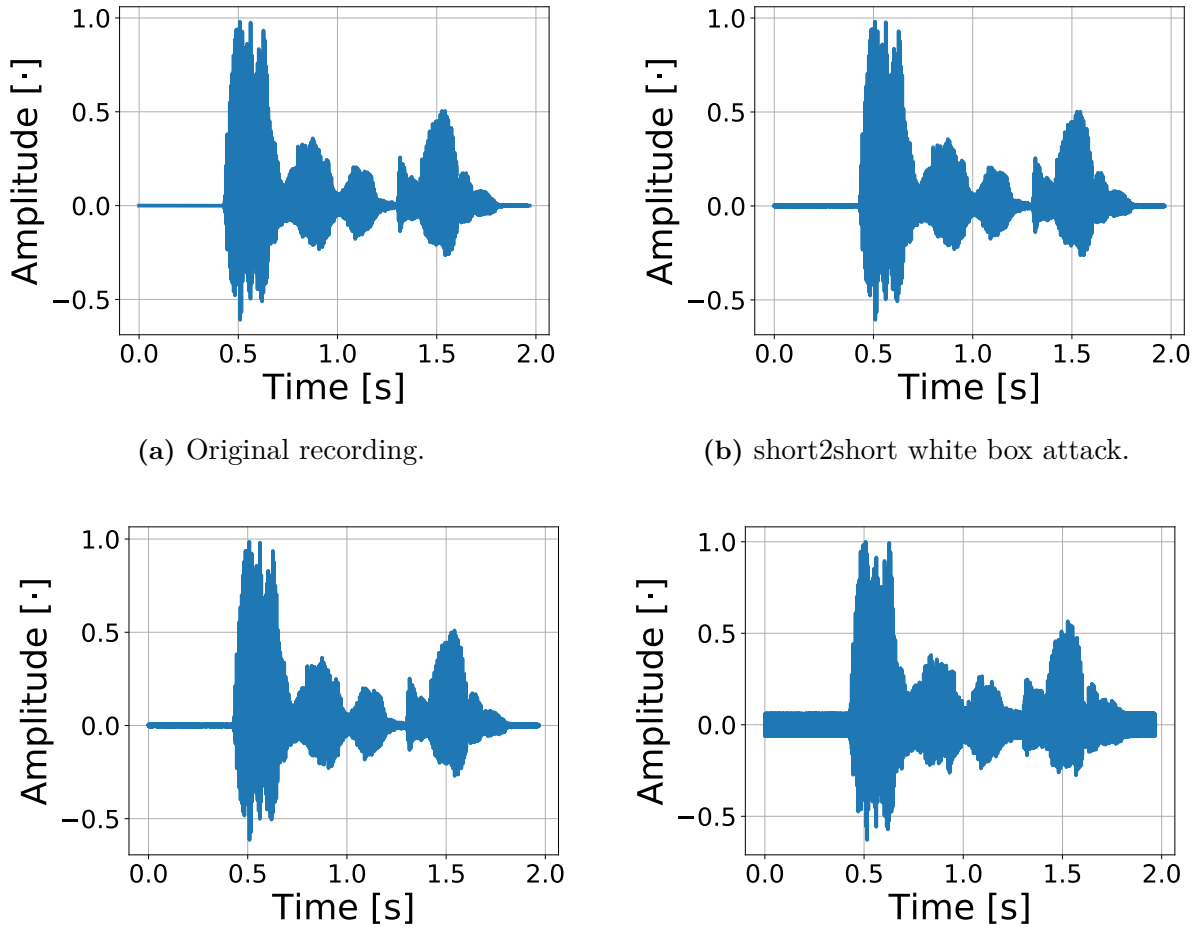
White box attacks, dataset A, contains three categories, short, medium and long. These categories are related to the length of the recorded signal. Short recordings are between 1 and 2 seconds, medium recordings are between 3 and 4 seconds, and long recordings are between 6 and 7 seconds. Each category contains three types of adversarial attacks, these are short, medium and long attacks. This means the adversarial noise has either a short, medium, or long target sentence. The following targets are,

- **Short**: Open all doors.
- **Medium**: Switch off wifi connection.
- **Long**: I need a reservation for sixteen people at the seafood restaurant down the street.

Short original recordings with related target are called short2short, short2medium and short2long. Medium and long recordings have the same naming scheme.

The white box dataset named dataset A consists of 900 normal and 900 adversarial examples. The normal examples contains 300 examples from each category, short, medium and long. Note, that none of the normal examples are the original examples used to generate the adversarial examples. The 900 adversarial examples are generated from 300 original examples, where 100 examples are from each category, short, medium and long. Each example is attacked with the three targets, yielding 900 adversarial examples.

In Figure 3.3 a short original recording with its corresponding short, medium, and long targeted white box attacks are shown.




**(a)** Original recording.

**(b)** short2short white box attack.

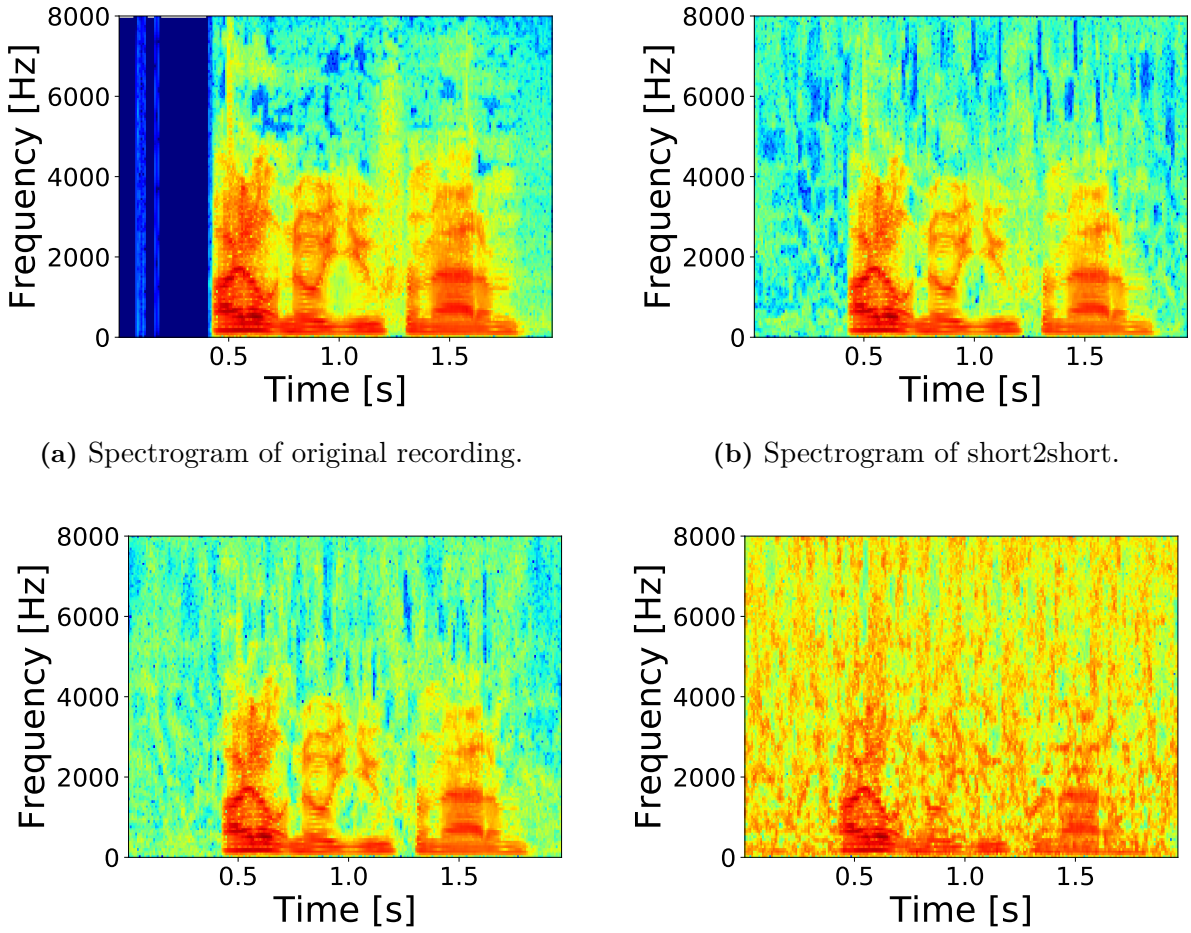
**(c)** short2medium white box attack..

**(d)** short2long white box attack.

**Figure 3.3:** The original recording and the corresponding short, medium and long targeted white box attacks.

By visually comparing the four figures in Figure 3.3, it is possible to see a difference in the general noise level. A distinctive difference can be seen between all other figures and the short2long Figure 3.3d. In general it has been found that the longer the target the larger amplitude of the adversarial noise.

A more clear visual difference between the original recording, short2short, and short2medium can be found when inspecting the spectrograms in Figure 3.4.




**(a)** Spectrogram of original recording.

**(b)** Spectrogram of short2short.

**(c)** Spectrogram of short2medium.

**(d)** Spectrogram of short2long.

**Figure 3.4:** Spectrogram of the original recording and the corresponding short, medium and long targeted white box attacks.

To investigate this further the original recording is subtracted from the adversarial example, resulting in the adversarial noise. The Signal-to-Noise Ratio (SNR) is found by subtracting the Root-Mean-Square (RMS) in dB of the adversarial noise from the RMS of the original recording in dB. This is done for all white box categories, i.e. all recording lengths and targets lengths. The SNRs can be seen Table 3.1.

| | Orig. L. | Orig. M. | Orig. S. |
|---|---|---|---|
| Target L. | 25.25 | 19.66 | 3.40 |
| Target M. | 30.20 | 27.76 | 25.07 |
| Target S. | 33.02 | 31.17 | 30.26 |

**Table 3.1:** Signal-to-Noise Ratios of various input lengths with various target sentence lengths.

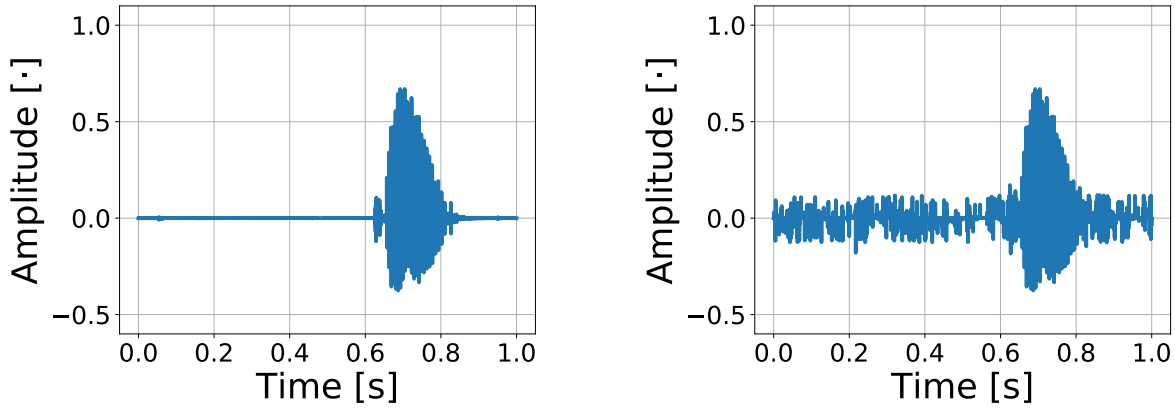
Table 3.1 corresponds well with what was seen in Figure 3.3. When the original signal is short with a long target, short2long, the adversarial noise is high compared to a long original recording with a short target, long2short.

## Black Box Attacks

The black box attacks, dataset B, are split into two main categories: Adversarial and normal. Each category consist of 10 subcategories. These subcategories are "down", "go", "left", "no", "off", "on", "right", "stop", "up" and "yes". In the main category normal, each of the 10 subcategories contains the related recordings, where the name of the subcategory corresponds to what is said in the recording. In the main category adversarial the subcategory is the target. In each target category is 10 more subcategories which corresponds to the specific original recording with the aforementioned subcategory as their target.

The black box dataset named dataset B consists of 1800 normal examples and 1800 adversarial examples. The 1800 normal examples consist of 180 examples for each command. The 1800 adversarial examples are made from 20 examples from the 10 commands, with targets from the 9 other commands. The files in the black box dataset are all approximately one second long.

An example of a original recording of the word 'go' and an adversarial attack on the word 'go' with the target 'down' can be seen in Figure 3.5.

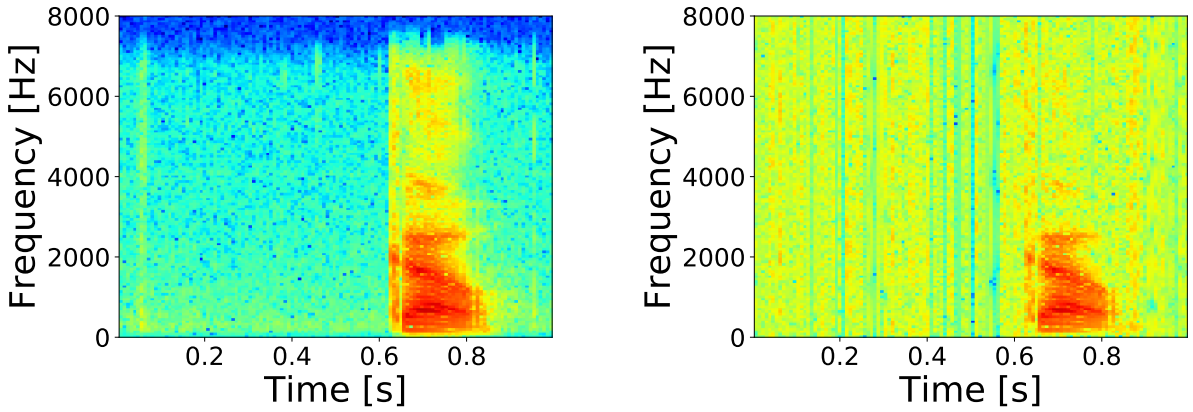



**(a)** Original recording 'go'.

**(b)** Adversarial example target 'down'.

**Figure 3.5:** Original recording of the word 'go' and adversarial example with target 'down' on the aforementioned original example 'go'.

From Figure 3.5 it can be seen that the adversarial attack increases the general noise level across the recording. A spectrogram of the aforementioned original examples and adversarial example can be seen in Figure 3.6.




**(a)** Spectrogram of original recording 'go'

**(b)** Spectrogram of adversarial example.

**Figure 3.6:** Spectrogram of the original recording of the word 'go' and adversarial example with target 'down' on the aforementioned original example 'go'.

The average adversarial noise between all original recordings and their targeted adversarial examples is 10.35 dB. Comparing this with Table 3.1 shows a worse original recording to adversarial noise ratio for all categories except the short2long category. However, it should be noted that a direct comparison should not be made. This is because the black box attacks and their related original recordings are one word and not a sentence like the white box attacks. In addition, the target length and original recording length is not varied either.

When visually comparing the adversarial noise of Figure 3.5b and Figure 3.3d a difference is observed. In Figure 3.3d the adversarial noise is more dense than in Figure 3.5b. This is only noted and why it occurs is not investigated further. It should also be noted that it is not necessarily advantageous to compare the black box attacks and the white box attacks, due to their differences.

## Considerations

The white box and black box spectrograms, in Figure 3.4 and Figure 3.6 respectively, show that the adversarial attacks add noise across all frequencies. In general the amplitude is highest at lower frequencies, in both the original and adversarial examples. The high amplitude areas contain mostly components from the spoken word or sentence. Most of the speech components occur in the lower frequencies, which yields a higher general adversarial noise change in the higher frequencies than in the lower frequencies. This supports the use of feature extraction methods which has higher resolution at higher frequencies when detecting if an example is adversarial or not.

By further visual inspection of Figure 3.4 it can be seen that a significant change occurs from 0s to 0.5s, when comparing the original example and an adversarial example. In

general, as can be seen in Figure 3.6 a significant change in amplitude occurs when speech is present compared to when it is not. This can also be seen in the time domain presentation, Figure 3.5b and Figure 3.3d. It could be interesting to see how the silent and speech parts of an example contribute to the performance of a NN together with different feature extracting methods.

It should be noted that the method used for dataset A and dataset B when generating the adversarial examples can result in a bias. The adversarial examples from A are generated from 100 original examples from each short, medium and long category. This results in three adversarial examples with three different targets generated from the same original example. This results in a pattern which could create a bias. A similar scenario occurs for dataset B. However, here there are 9 files instead of the 3 original examples with different targets. The dataset A and B are generated by a third party and the possibility of a bias is therefore only noted and not resolved.

# 4 | Speech Processing

The purpose of this chapter is to investigate some feature extraction methods and signal representations for a speech recognition task. The feature extraction methods that will be presented are the Short-Time Fourier Transform (STFT) feature extraction method and the four cepstrum based feature extraction methods MFCC, Inverse Mel-Frequency Cepstral Coefficients (IMFCC), Gammatone Frequency Ceptral Coefficients (GFCC) and Inverse Gammatone Frequency Ceptral Coefficients (IGFCC). The MFCC and GFCC feature extraction methods are investigated as these both show promising results in speech recognition tasks [34, Chapter 6] [35]. The feature extraction methods IMFCC and IGFCC are also investigated based on the findings in section 3.2.

Let $x[n]$ be an infinite discrete time signal containing speech to be recognized. It is unclear how to input this infinite sequence to a classifier with a fixed input size which motivates the need to extract features from the signal. A simple way to obtain a finite length sequence is to segment the infinite sequence into an infinite series of finite sequences as

$$x[n] \rightarrow x_r[m] = x[m + rR], \quad m = 0, 1, ..., M - 1 \tag{4.1}$$

where $r$ is the shift index, $R$ is the shift between segments, and $M$ is the segment size [36, Chapter 11]. The choice of $M$ is task dependent based on the desired time resolution. There are three overall choices for $R$. $R > M$ results in a loss of information as some samples are discarded. Selecting $R = M$ results in non-overlapping sequences and is a necessity for no loss of information. With $R < M$ some samples are overrepresented and could be useful as the arbitrary segmentation place could otherwise split an important speech part into different segments. By overrepresenting some samples, the edge case problems can be mitigated but at a cost of having unnecessary data samples for a full signal representation.

The following feature extraction methods can be grouped into methods that have no loss of information and methods that do. Methods with no loss of information is able to perfectly reconstruct the original input and is said to have a Perfect Reconstruction (PR) property. If the lossy methods can accurately discard unnecessary samples, they would be advantageous as the resulting reduction in samples means a reduction in processing time for the ANN.

## 4.1 Short-Time Fourier Transform

The STFT [36, Chapter 11] transforms the series of sequences into frequency domain by applying the Discrete Fourier Transform (DFT) to each sequence as

$$x_r[m] \leftrightarrow X_r[k] = \sum_{m=0}^{M-1} x_r[m]w[m]\mathrm{e}^{-j2\pi\frac{mk}{K}}, \quad k = 0, 1, ..., K - 1 \tag{4.2}$$

where $K = M$ is chosen for PR. The window function $w[m]$ can be chosen to reduce edge case problems when applying the DFT [36, Chapter 11]. The STFT is PR if $R \leq M$ and $w[m] \neq 0$ [36, Chapter 11].

## 4.2 Cepstrum based Feature Extraction

The cepstral coefficients [34, Chapter 6] is the inverse Fourier transform of the logarithm of the STFT and is computed as

$$c_r[l] = \sum_{k=0}^{M-1} \ln |X_r[k]| \, \mathrm{e}^{j2\pi \frac{kl}{M}}, \quad l = 0, 1, ..., L-1 \tag{4.3}$$

where $L$ is the number of cepstrum coefficients. It only uses the magnitude of the STFT and thus loses PR but can be expanded to include the phase to maintain PR [34, Chapter 6]. The power cepstrum can be found by using the power spectrum $P_r$ as

$$P_r[k] = |X_r[k]|^2. \tag{4.4}$$

instead of $X_r$ in Equation (4.3).

There are a few different cepstrum based feature extraction methods that take a basis in the power spectrum and apply a filterbank before computing the cepstral coefficients [34, Chapter 6]. The logarithm of the filtered power spectrum is computed as

$$\lambda_r[s] = \ln \left( \sum_{k=0}^{M-1} H_s[k] P_r[k] \right) \tag{4.5}$$

where the filterbank $H_s$ can be chosen [34, Chapter 6]. Different filterbanks will result in different cepstral coefficients, such as MFCC, IMFCC, GFCC and IGFCC.

### Mel-Frequency Cepstral Coefficients

The MFCC has shown to give useful features for tasks such as speech recognition [34, Chapter 6]. The Mel-Frequency filterbank is modeled after the human auditory system as a series of triangular filters. An example of a filterbank used for calculating MFCC is seen in Figure 4.1.

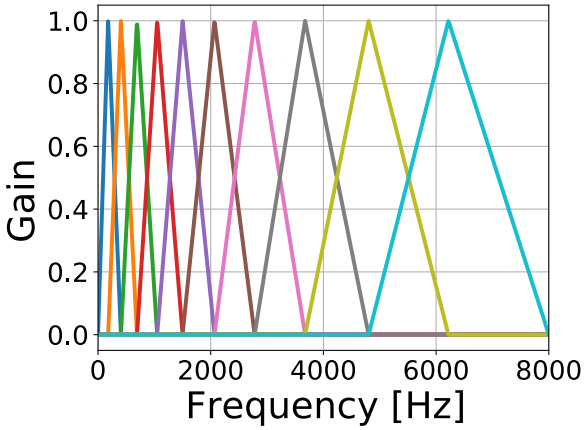



**Figure 4.1:** An example of a filterbank used to generate the MFCC.

The filters are found as

$$H_s[k] = \begin{cases} 0, & k < f[s-1] \\ \frac{k-f[s-1]}{f[s]-f[s-1]}, & f[s-1] \leq k \leq f[s] \\ \frac{f[s+1]-k}{f[s+1]-f[s]}, & f[s] \leq k \leq f[s+1] \\ 0, & k > f[s+1] \end{cases} \tag{4.6}$$

where $f[s]$ is a sequence frequencies uniformly spaced in the mel-scale [34, Chapter 6]. The mel-scale is defined as Equation (4.7a). It contains $S+2$ bins starting at $f_{\min}$ and ending at $f_{\max}$ in Hz. The sequence is found as

$$B(f) = 1125 \ln\left(1 + \frac{f}{700}\right) \tag{4.7a}$$

$$B^{-1}(b) = 700\left(\mathrm{e}^{b/1125} - 1\right) \tag{4.7b}$$

$$f[s] = \left(\frac{M}{f_s}\right) B^{-1}\left(B(f_{\min}) + (s+1)\frac{B(f_{\max}) - B(f_{\min})}{S+1}\right) \tag{4.7c}$$

where $f_s$ is the sampling frequency in Hz. The number of filters $S$ can be chosen depending on the task but should be less than $M$. $f_{\min}$ and $f_{\max}$ can also be chosen but can not be respectively smaller or larger than 0 or $f_s/2$.

The MFCC can then be computed by applying the Discrete Cosine Transform (DCT) as

$$c_r[l] = \sum_{s=0}^{S-1} \lambda_r[s] \cos\left(\frac{(s+\frac{1}{2})l\pi}{S}\right), \quad l = 0, 1, ..., L-1 \tag{4.8}$$

where $L$ determines the number of coefficients to compute and can not exceed $S$. The MFCC thus depends on the choice of $L$, $S$, $f_{\min}$, and $f_{\max}$.

## Inverse-MFCC

The MFCC is modelled after the human auditory system, thus it has more filters located at lower frequencies than at higher frequencies. The filterbank used for calculating the IMFCCs is found by reversing the MFCC filterbank. An example of a IMFCC filterbank with 10 filters is seen in Figure 4.2. The reason why it is wanted to look into IMFCC is found in Figure 3.4 and Figure 3.6. Here a higher general noise change is found at higher frequencies when comparing an adversarial example with its corresponding original example. Therefore it is of interest to investigate if a higher concentration of filters at higher frequencies, compared to MFCC, will improve the performance of the model.

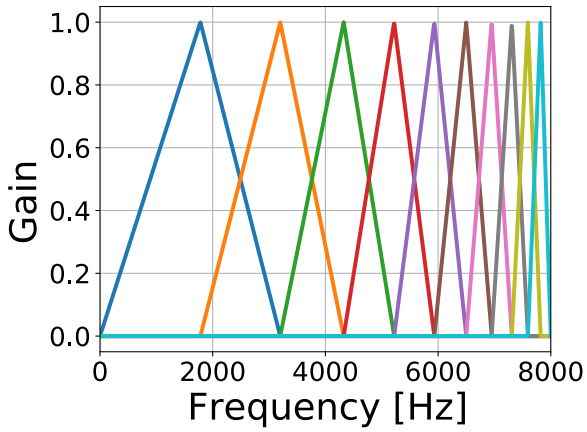



**Figure 4.2:** An example of a filterbank used to generate the IMFCC.

## Gammatone-Frequency Cepstral Coefficients

Another cepstral based feature extraction is the GFCC which is modelled after the human cochlear filtering mechanism and which has shown promising performances in e.g. speaker recognition tasks [37] and in speech recognition tasks [35]. This filtering mechanism is modelled by a filterbank of gammatone filters. These gammatone filters have the transfer function seen in Equation (4.9) [35].

$$H(f) = \frac{(N-1)!}{(\alpha + j(2\pi f - 2\pi f_c))^N} \tag{4.9}$$

Where $N$ is the filter order which is typically set to 4, $\alpha$ is the bandwidth, and $f_c$ is the center frequency. The number of center frequencies is equal to the number of filters in the filterbank and they are calculated as seen in Equation (4.10) [35].

$$f_c(s) = -C + \mathrm{e}^{s \cdot \log(\frac{f_{\min}+C}{f_{\max}+C})/S} \cdot (f_{\max} + C) \tag{4.10}$$

Where $S$ is the number of filters and $1 \leq s \leq S$. $C = 228.83$ and $f_{\min}$ and $f_{\max}$ are the minimum and maximum cutoff frequencies of the filterbank. The bandwidth $\alpha$ in

Equation (4.9) is calculated based on the Equivalent Rectangular Bandwidth (ERB) as seen in Equation (4.11).

$$\alpha(f_c) = \frac{f_c}{9.26} + 24.7 \tag{4.11}$$

Once the gammatone filterbank is generated the GFCC is calculated the same way as the MFCC using Equation (4.5) and Equation (4.8).

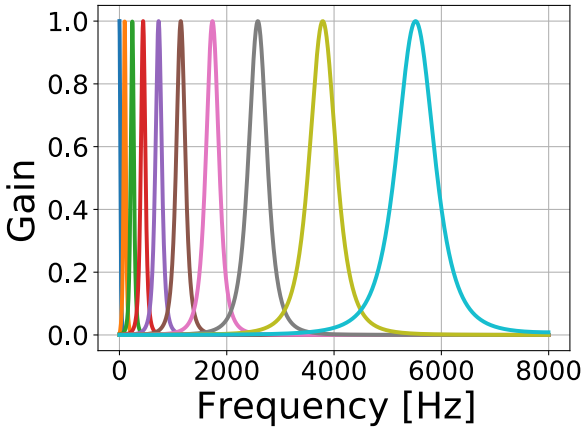



**Figure 4.3:** An example of a filterbank used to generate the GFCC.

## Inverse-GFCC

As seen in Figure 4.3 the gammatone filterbank is similar to the filterbank used for MFCC Figure 4.1 as they both have more filters in the lower frequencies than in the higher frequencies. In Figure 3.4 and Figure 3.6 it is seen that the relative change when comparing a clean sample to its corresponding adversarial example is larger at higher frequencies. This observation justifies the use of IMFCC and therefore it also justifies looking into the IGFCC. The IGFCC are calculated using a reversed gammatone filterbank. An example of a filterbank used for IGFCC is seen in Figure 4.4.

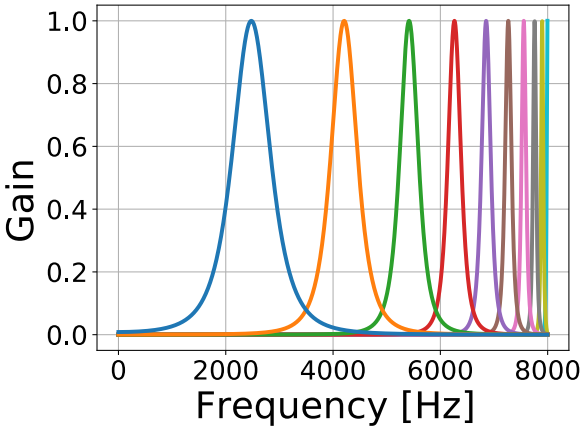



**Figure 4.4:** An example of a filterbank used to generate the IGFCC.

Five feature extraction methods have now been presented; The STFT along with the four cepstrum based feature extraction methods MFCC, IMFCC, GFCC and IGFCC.

# 5 | Evaluation Metrics

It is desired to be able to evaluate the performance of a model. Different evaluation metrics can be used for different classification problems. In this project it is desired to detect if an example is adversarial or not, which is a binary classification problem. The following section describes some different metrics for evaluating the performance of binary classifiers. Furthermore a description is made of how a 95 percent confidence interval is used to make a more accurate evaluation of a model's performance.

If only accuracy is used to evaluate the model, problems can arise when having an unbalanced amount of examples. This is the case if more adversarial examples are used than non adversarial examples and vice versa. It would therefore be possible for a model to always make the same prediction and get a good accuracy. The accuracy will then depend on the balance of the examples. This is called the accuracy paradox and is why accuracy should be used with caution. By using various metrics, different information about the model's performance can be extracted. This can give an increased insight into the model performance. [38]

The confusion matrix is commonly used for binary classification as a step to evaluate the performance of a model. For a binary classification case the confusion matrix is as shown in Table 5.1.

| | Adversarial | Normal |
|---|---|---|
| Predicted adversarial | True positive | False positive |
| Predicted normal | False negative | True negative |

**Table 5.1:** Confusion matrix for binary classification, predicting if an example is adversarial or not.

Table 5.1 shows if an example is predicted to be an adversarial example or a normal example and if that example is actually adversarial or normal. True positive (TP) is when an example is adversarial and is predicted to be adversarial. True negative (TN) is when an example is normal and predicted to be normal. False negative (FN) is when an example is adversarial and is predicted to be normal, and finally, False positive (FP) is when an example is normal and is predicted to be adversarial. Note, to achieve a 100 % accuracy FN and FP should be zero, resulting in values only in the diagonal. If either TP or TN is zero it can indicate multiple things: The model always predicts the same, the examples are only of one kind or the model systematically misclassify that type of example.

For some problems, the cost of getting an FP or an FN is not the same. For some problems such as facial recognition for unlocking systems, it is crucial that the system only unlocks

when the right person attempts to unlock the system. Here the cost of an FP is a security breach and unacceptable while an FN would simply mean a person with security clearance need to attempt again or use another method of entry. For other systems like a fire alarm system, there is a higher cost associated with an FN as this would mean a failure to alert people when there is a fire. Here the cost of an FP is not as severe. The number of false positives or negatives is however not sufficient to evaluate the performance of a model. It could be that different networks have similar performance w.r.t. the most costly false result, but it is still desired to get a proper evaluation of the model in order to decide which is better.

The raw number of misclassifications a model makes is not very telling of its performance. Therefore the rate of which false positives/negatives occur will be used instead and are defined as

$$\mathrm{FP}_{\mathrm{rate}} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}}, \quad \mathrm{FN}_{\mathrm{rate}} = \frac{\mathrm{FN}}{\mathrm{FN} + \mathrm{TP}}. \tag{5.1}$$

The model outputs a number between 0 and 1 that can be thought of as the probability that a given input is adversarial. In order for the network to predict that the input is adversarial the output $y$ has to be larger than a threshold $p$ to determine the input as adversarial. A threshold of $p = 0.5$ is not necessarily the optimal choice as the model could potentially have a bias or given that FPs and FNs are not necessarily equally costly. The model can then be evaluated by finding the false positive and false negative rates for different $p$ values and plotting the false negative rate over the false positive rate. If one model has a higher false negative rate for all false positive rates it is systematically worse than the other. However both models can have a lower false negative rate but at different false positive rates. The curve will thus show the model with the best performance at any given rate and allow the designer to choose the threshold $p$ in order to achieve the desired trade-off between the rates of FPs and FNs.

## 5.1 Confidence Interval

The above mentioned evaluation metrics will give an idea if a specific model performs well or not. As described in Chapter 2 two models with identical architectures can converge to different optimums depending on e.g. the initial weights, the data, and how the hyper-parameters are set. Therefore by only evaluating on one model it is possible to evaluate the model architecture based on edge cases. To help this issue the confidence interval can be calculated combining results from training the same model several times. For each training the model will have the same architecture but differ in the initial weights. The confidence interval is useful when neither the mean $\mu$ or the variance $\sigma^2$ of a distribution is known. In this case it is sometimes more valuable to specify with a certain confidence in which interval $\mu$ is located rather than estimating an exact location of $\mu$ [39, Chapter 7]. In the case of evaluating a neural network the 95 percent confidence interval can tell with 95 percent confidence in which interval the mean accuracy of the model is located. As training a model is in many cases a time consuming task it is not certain that a high

number of samples is available to calculate the confidence interval of the models accuracy. If this is the case the $100(1-\alpha)$ confidence interval can still be calculated using the t-distribution as seen in Equation (5.2) [39, Chapter 8].

$$\mu \in \left( \bar{x} - t_{\alpha/2,n-1}\frac{s}{\sqrt{n}},\ \bar{x} + t_{\alpha/2,n-1}\frac{s}{\sqrt{n}} \right) \tag{5.2}$$

Where $\bar{x}$ is the sample mean, $s$ is the sample standard deviation, $n$ is the number of samples, and $t$ is a random variable dependent on the confidence interval and $n$. By training a model several times with different initial weights and calculating the 95 percent confidence interval it is possible to make a more accurate evaluation of a model's performance.

# 6 | Analysis Conclusion and Problem Description

The areas described in the initial problem statement have through the analysis been researched. The main concepts of ANNs have been described along with some of the possible challenges associated with ANNs. The neural network type CNN has been investigated and some of its adjustable parameters have been presented. The concept of adversarial examples has been described and some examples have been presented of how they pose a threat to state-of-the-art neural networks. Some methods for generating adversarial examples in the audio domain have been presented. The dataset used in the project have been investigated in order to have a clear understanding of how data can differ and which properties hold for the different types of data in the dataset. The five feature extraction methods STFT, MFCC, IMFCC, GFCC and IGFCC have been investigated with the purpose of being able to test different types of features as input for a neural network. Finally, different evaluation metrics have been presented. These are studied to be able to evaluate a neural network correctly and not be mislead by too simple evaluation metrics.

Based on the analysis a problem description can be formulated.

## 6.1 Problem Description

The aim of the project is to examine the detection of adversarial examples using different feature extraction methods with the datasets used by [12]. This will be done using a baseline CNN model with the same architecture as used in [12] and train it to be able to separate adversarial examples from normal examples. This will be done for the white box (A) dataset, the black box (B) dataset and the datasets combined called AB. The features described in Chapter 4 will be extracted from the datasets and used as input for the baseline model. This enables the possibility of examining how these features affect the performance of the model. In section 3.2 it is found that the relative change across all frequencies is higher for the segments where no speech is present. It is therefore investigated how the baseline model performs when the data used for training and testing is split up into speech and nonspeech. Furthermore, the datasets described in section 3.2 represents a limited and ideal case. The robustness of the baseline model in regards to noise is therefore investigated. This is done by evaluating the baseline model using different amounts and types of noise present in all or either training, validation and test sets.

This gives rise to the following problem statement:

**How do different feature extraction methods affect the performance of a CNN when detecting adversarial examples for speech recognition, how robust are they to noise, and is it the characteristic from speech or nonspeech that is most important when classifying?**

The following list should give an overview into what will be done in the following report part.

1. Chapter 7: Choices and parameters regarding preparation and handling of data.
2. Chapter 8: Definition, Implementation, training and evaluation of the baseline model which is used in the project. The baseline is trained individually on dataset A, B, and AB, and tested individually on dataset A and B, for the five feature extraction methods, STFT, MFCC, IMFCC, GFCC, IGFCC.
3. Chapter 9: Investigates the influence of speech and nonspeech segments in Dataset A. Both when it is trained individually on A, speech and nonspeech and tested individually on speech and nonspeech segments, for the five feature extraction methods, STFT, MFCC, IMFCC, GFCC, IGFCC.
4. Chapter 10: Investigates the model's and the different feature extraction methods' robustness to noise. This is done when noise is present in all or either the training, validation and test set. Different noise types and noise volumes are used, for the five feature extraction methods, STFT, MFCC, IMFCC, GFCC, IGFCC.

# Part II

# Implementation and Evaluation

# 7 | Data Preparation

In the current chapter certain aspects and choices will be explained. It is made clear why the examples are split up into blocks instead of using zero padding as in [12], together with a review of the used parameters for the feature extraction and the general data setup. It is also explained how a dataset is split into training, validation and test sets.

It was found that some of the short2long examples in dataset A were misclassified by the model as normal and not adversarial. The short2long examples should be the 'easiest' examples to classify as they contain the highest amplitude of adversarial noise, see section 3.2. These examples was investigated further and it was found that some of the files that are labeled as adversarial are in fact normal examples. Due to this discovery the data set used was inspected. It was found that 22 short2long examples were mislabeled as adversarial examples. These 22 adversarial examples are removed from dataset A. Dataset A thus contains 878 adversarial examples.

In Figure 7.1 an illustration is seen of the order in which the data preparation is made.

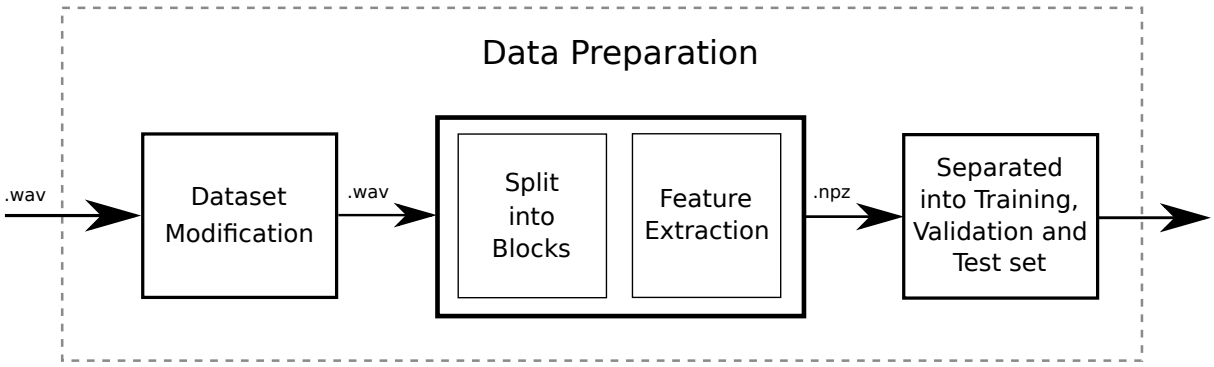



**Figure 7.1:** Illustration of the data preparation.

In Figure 7.1 it is seen how a dataset is handled before it can be used for training the model. The datasets contains .wav files which are at first modified when investigating the influence of speech and nonspeech segments, and the robustness to noise. The dataset is not modified for the investigation in Chapter 8, where the model and feature extraction methods are evaluated. After the possible dataset modifications the dataset still consists of .wav files. These are then split into blocks and the five features described in Chapter 4 are extracted from each block before storing the data as .npz files. The data in the .npz files are then split into training, validation and test sets before training the model.

## 7.1 Zero Padding and Block Separation

The performance of a neural network is dependent on what kind of data is used as input to the network. In section 3.2 the dataset for this project is described, where it is found that the dataset consists of examples with various lenghts. As this is the case some consideration must be made of how to handle this, as ANNs in general are not suited for input of various sizes. One way of solving this issue is by using zero padding, which is what is done in the reference paper [12]. In the reference paper the longest example in the dataset is found and all other examples are then padded with zeros to make them the same length as the longest example. Another approach is to split up each example into blocks of equal size like in [40]. The result of this will be that some examples will produce more blocks than others. In this project it is chosen to split each example into blocks and in Appendix A the full description is found of how the examples are split into blocks. The reason why it is chosen not to use zero padding is the fact that when using zero padding some demands are made to the dataset. If the dataset is not balanced regarding the length of examples, more zeros will be present in one of the categories, which might affect what is learned by the NN. In Figure 7.2 an example is seen of how zero padding can affect a CNN.

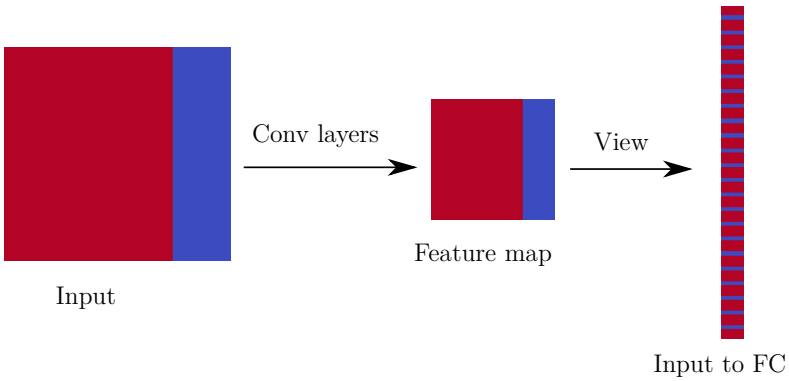



**Figure 7.2:** Example of how zero padding can affect a CNN. Relevant data (red), zero padding (blue)

Figure 7.2 shows what happens to an input containing relevant data (red) and zero padding (blue) through a CNN until the input layer of the fully connected network. It is seen that a significant amount of zeros are still present after the input has been through the convolutional layers (Feature map). When the output feature map of the convolutional layers is reshaped to fit the fully connected network the zeros are still present (Input to FC). The amount of zeros in the input to the fully connected network will change if the relative amount of relevant data in the input changes. However the location of the zeros in the input to the fully connected network will not change. This is a problem if the dataset is not balanced because the model could simply assign large weights to the input nodes where zeros are present for data with one label and where zeros are not present for data with a different label.

## 7.2 Feature Extraction Parameters and Data Setup

In this project five types of features will be extracted from the data set. These are the MFCC, IMFCC, GFCC, IGFCC, and STFT features, which are described in Chapter 4. The implementations of the four cepstrum based feature extraction methods; MFCC, IMFCC, GFCC and IGFCC are not developed in this project. The implementations of these four feature extraction are made in [41] for MATLAB. The implementations of the GFCC and IGFCC filterbanks are described in [42]. In this project changes are only made to the MATLAB scripts from [41] to make them compatible with the feature extraction scripts in Appendix A, Listing A.2.

Several parameters can be set for the feature extraction methods. In this project the parameters for the feature extractions are set as in Table 7.1 where the number of filters is irrelevant for the STFT.

**Table 7.1:** Feature extraction parameters.

| Parameter | Value |
|---|---|
| Window size [ms] | 32 |
| Window step size [ms] | 16 |
| Filters | 20 |
| Block size [ms] | 512 |
| Block step size [ms] | 512 |
| Sample rate | 16000 |

The window size and step size determines $M$ and $R$ in Equation (4.1). These are given in samples and not milliseconds but can be found using the sample rate. The window size determines the spacing in frequency between bins in the DFT since $K$ was set equal to $M$ for PR. A larger value corresponds to a better frequency resolution but at a cost of increased computation time for all subsequent operations. By using a window size of 32 ms, the frequency spectrum is represented in 31.25 Hz intervals given the sample rate of 16000. It is difficult to say what the optimal window size should be. The window step size determines the time resolution. Likewise it is also difficult to say what the optimal value should be however it was chosen to use a smaller value than the window size to reduce edge case problems as described in Chapter 4. With $M$, $K$ and $R$ determined it is possible to compute the STFT features.

The number of filters is the variable $S$ in Equation (4.8) and the number of cepstral coefficients $L$ is set to be equal to $S$. Here $f_{\text{min}} = 0$ and $f_{\text{max}} = 0.5f_s$ where $f_s$ is the sample rate. With these variables defined, the cepstrum coefficients can then be computed.

The block size and block step size in Table 7.1 is not a part of the feature extraction methods but is used to split the raw input file before it is used as input to the feature extraction methods. With the parameters set as in Table 7.1 the raw input audio file is split into a number of blocks of $\frac{16000}{1000} \cdot 512 = 8192$ samples. Features are then extracted from each of these blocks. The output of each feature extraction are then a feature map of size (filters $\times \frac{\text{block size}}{\text{window step size}}$). An example of such a feature map is seen in Figure 7.3.

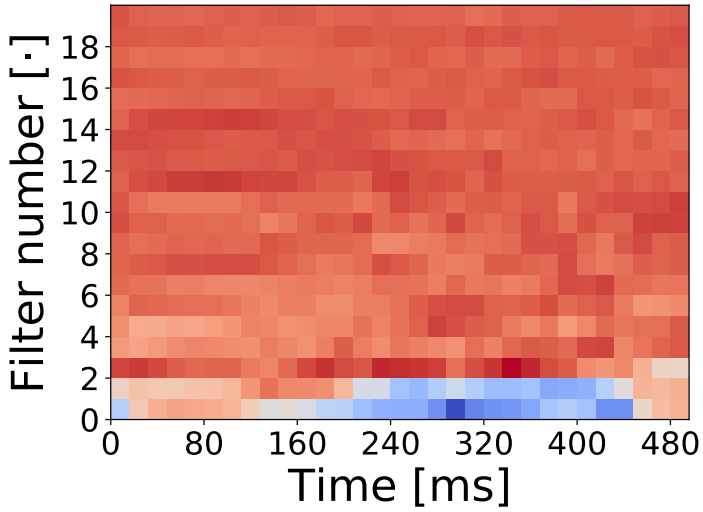



**Figure 7.3:** Example of a feature map from GFCC.

As seen in Figure 7.1 the calculated features are then saved as .npz files with the shape (number of feature maps $\times$ filters $\times \frac{\text{block size}}{\text{window step size}}$). Once the .npz files are generated they are ready as input for the NN where the split into training, validation and test sets. Each npz file contain three categories of data: The feature coefficient, their label, and the name of the file they are made from.

## 7.3 Splitting into Training, Validation and Test Sets

As stated in section 2.3 it can be beneficial to split the dataset into a training set $\mathcal{T}$, a validation set $\mathcal{V}$ and a test $\mathcal{S}$ when working with an ANN model. In this project it is chosen to use 70 % of a dataset for training, 10 % for validation, and 20 % for testing. It is important that the data present in one of the sets will not be present in the two other sets. If this is not the case a model can be tested on data which was used for training, thus corrupting the evaluation of the model. The function for splitting data into the three sets is the TTSplit function from the sklearn.model_selection package in Python. This function also preserves the ratio of adversarial to normal examples such that if the original dataset is balanced, each of $\mathcal{T}$, $\mathcal{V}$, and $\mathcal{S}$ are also balanced. Before the data is split, a random seed is set to zero. This is done because in some cases it is desired to only modify the test set, e.g. training on A and testing on A with noise. As seen in Figure 7.1 these dataset

modifications produce new .wav files and it is therefore important that the correct 20 % of these files are used for the test set. It is therefore important that the order of the files before splitting into training, validation and test sets is the same for each test.

It has now been explained how and why data is split into blocks, how features are extracted, and how data is split into training, validation and test sets.

# 8 | Evaluation and Comparison of the Feature Extraction Methods

In this chapter the implementation of the baseline model is described before evaluating the performance of the model and the different feature extraction methods. The evaluation of the baseline model's performance is done by training and evaluating on the datasets, described in section 3.2, with the different feature extraction methods, described in Chapter 4.

## 8.1 Baseline Architecture

As evident by Chapter 2 there are many different ways to design the architecture of an ANN. There is not an analytic path to design the network architecture for a certain specification but only empirical evidence that certain structures generally outperform others [13]. The architecture of the baseline model is therefore chosen to be the one from [12]. The architecture of the baseline model is shown on Figure 8.1.

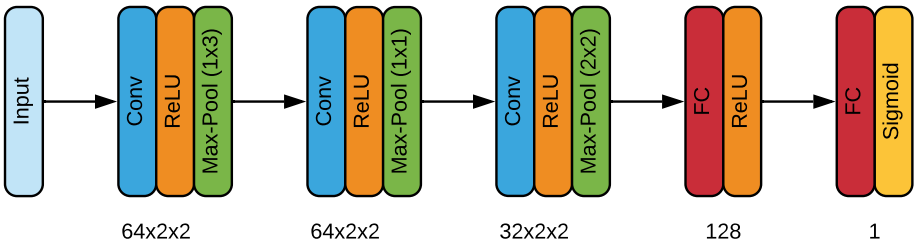



**Figure 8.1:** Diagram of the baseline model architecture.

As seen in Figure 8.1 the CNN consists of three convolutional layers which all use the ReLU AF. Each convolutional layer is followed by a pooling layer where max pooling is used in all three cases. The first pooling layer has a pool size of $(1 \times 3)$, the second has a pool size of $(1 \times 1)$ and the third has a pool size of $(2 \times 2)$. These layers are followed by two fully connected layers where the second is the output layer. The first fully connected layer has 128 neurons and the output layer produces one output using the Sigmoid AF.

The variable length inputs to this model are split into blocks as described in Chapter 7 as the model can only handle fixed size inputs.

## 8.2 Baseline Implementation

In the following section the implementation of the baseline model in PyTorch is reviewed. Only segments of the code necessary to implement a fully working CNN are described. These segments are the main model module and its training procedure.

During development, a focus was put on creating a versatile Python script that could handle the different types of features with minimal changes. The discussion and implementation of this versatility is found in Appendix C.

In Listing 8.1 the class Detection() can be seen. This is the code segment containing the model.

**Listing 8.1:** Implementation of the baseline model in PyTorch.

```
1  class Detection(nn.Module):
2      def __init__(self, in_h, in_w):
3          super(Detection, self).__init__()
4          self.conv1 = nn.Conv2d(1, 64, kernel_size=(2, 2))
5          self.conv2 = nn.Conv2d(64, 64, kernel_size=(2, 2))
6          self.conv3 = nn.Conv2d(64, 32, kernel_size=(2, 2))
7
8          self.nfc = 32 * int((((in_h-2+1)/1-2+1)/1-2+1)/2) * int((((in_w-2+1)/3-2+1)/1-2+1)/2)
9
10         self.fc1 = nn.Linear(self.nfc, 128)
11         self.fc2 = nn.Linear(128, 1)
12         # Weight initializations
13         nn.init.kaiming_uniform_(self.conv1.weight, nonlinearity='relu')    # He
14         nn.init.kaiming_uniform_(self.conv2.weight, nonlinearity='relu')    # He
15         nn.init.kaiming_uniform_(self.conv3.weight, nonlinearity='relu')    # He
16         nn.init.kaiming_uniform_(self.fc1.weight, nonlinearity='relu')  # He
17         nn.init.xavier_uniform_(self.fc2.weight)
18
19     def forward(self, x):
20         x = F.relu(F.max_pool2d(self.conv1(x), (1, 3)))
21         x = F.relu(F.max_pool2d(self.conv2(x), (1, 1)))
22         x = F.relu(F.max_pool2d(self.conv3(x), (2, 2)))
23         x = x.view(-1, self.nfc)
24         x = F.relu(self.fc1(x))
25         x = self.fc2(x)
26         return x
```

Detection() consist of the two functions, __init__() and forward(). The __init__() is used to specify parameters such as input, hidden and output layers. The function forward() defines how the data is propagated through the layers when the forward pass is performed.

__init__() takes the parameters in_h and in_w, which is the height and width of each example. In __init__() three convolution layers are created, conv1, conv2 and conv3 all with a kernel size of 2x2. conv1 takes 1 channel as input and outputs 64 new channels, i.e. creating feature maps of depth 64. conv2 takes in the 64 channels and output 64, and conv3 takes in 64 channels and outputs 32 channels.

The variable nfc contains the size of a 1-dimensional vector used to contain the input nodes for the fully connected layer. The size is calculated by using Equation (2.18). The stride is set to 1, there is no zero-padding and $F_w$ and $F_h$, width and height of the feature map, is multiplied together. Two fully connected layers are then defined. fc1 which has 128 ouputs and the output layer fc2 has just a single output, creating the desired binary classification.

As described in Chapter 2 it is necessary to initialize the weights and biases in the model. As written in Figure 2.3 He initialization is typically used for ReLU activation functions. Furthermore, Xavier initialization can be and is typically used for sigmoid activation functions [20].

The function forward() takes x as input, where x is the data split into batches. The data x is propagated through the layers, with the output from the last layer being returned from the function. Note after the convolution layer conv3 a function view() is applied to reshape the data enabling it to be used in a fully connected layer.

An instance of the class Detection is required together with a loss function and optimizer, this can be seen in Listing 8.3.

**Listing 8.2:** Choice of Loss function, optimizer and instance of the class Detection created.

```
# Creating instance of model
net = models.Detection(x.shape[2], x.shape[3])

# Loss function - Binary Cross Entropy
criterion = torch.nn.BCEWithLogitsLoss()

# Optimizer
optimizer = torch.optim.Adam(net.parameters(), lr=lr, betas=(beta1, beta2))
```

The class instance is named net. The input x is a tensor in $\mathbb{R}^{N\times C\times H\times W}$ where $N$ is number of examples, $C$ is the number of channels, $H$ is the height of the example and $W$ is the width. To calculate the loss a function for binary cross entropy is used. This is used when binary classification is desired and is shown in Equation (2.10). Finally the Adam optimizer is used. Adam's first passed parameter is the parameter which should be optimized. The second parameter is the learning rate, described in Chapter 2. The last parameter controls the decay rates of the exponential moving average of the gradient and the squared gradient. However, this will not be elaborated or studied, the beta values, beta1 and beta2, are therefore set to default values.

The procedure to train the CNN using mini-batch gradient descent can be seen in Listing 8.3.

**Listing 8.3:** Training procedure for initial CNN.

```
for epoch in range(num_epochs):
  idxminibatches = np.random.permutation(NB) # shuffling
```

```
3   for j in range(NB):
4     i = idxminibatches[j] # index of minibatch

6     # Extract i-th minibatch from x and d
7     idxsmp = np.arange(B*i, min(B*(i+1), N)) # indices of samples for i-th minibatch
8     inputs = x[idxsmp]
9     labels = d[idxsmp]

11    # Initialize the gradients to zero
12    optimizer.zero_grad()

14    # Forward propagation
15    outputs = net(inputs)

17    # Error evaluation
18    loss = criterion(outputs, labels)

20    # Back propagation
21    loss.backward()

23    # Parameter update
24    optimizer.step()
```

The training contains two loops. The outer loop iterates through each epoch until the set amount of epochs, called num_epochs. The inner loop iterates over batches containing examples, where NB is the calculated number of batches in the training set with a certain set batch size. idxminibatches contains indexes of all batches which has been randomly shuffled. In each iteration of the inner loop an index from idxminibatches is used.

idxsmp contains an extracted batch consisting of all example indices from that batch, where the batch index corresponds to the index extracted from idxminibatches. As the batch size, B, is not necessarily a multiple of the total amount of examples, it is possible that the last batch contains fewer examples than the set batch size, B. Hence why min() in Listing 8.3 line 7 is necessary.

The gradients of the Adam optimizer are zeroed by using the zero_grad() operation. Line 15 calls the function forward() from Listing 8.1. Note when calling net(inputs) the function forward() is automatically executed. The loss is calculated by passing the outputs and labels to the function BCEWithLogitsLoss() from Listing 8.2. Then back-propagation is performed calculating the gradients with respects to the weights and finally the optimizer.step(), which is the Adam optimizer, updates the weights in the CNN.

An overview of the hyperparameters used for the baseline model can be seen in Table 8.1.

| | |
|---|---|
| Number of epochs | 50 |
| Batch size | 64 |
| Learning rate $\gamma$ | 1.0e−4 |
| $\beta_1$ | 0.9 |
| $\beta_2$ | 0.999 |
| Optimizer | Adam |

**Table 8.1:** Hyperparameters for Baseline model.

## 8.3 Evaluation and Comparison of the Feature Extraction Methods - Per Block

To evaluate the baseline model with the available data sets and feature extraction methods a series tests are performed. Two datasets are available for this project as described in section 3.2: A dataset with white box attacks, and a dataset with black box attacks. The dataset with white box attacks and black box attacks will be referred to as dataset A and dataset B, respectively. The combination of the two datasets is referred to as dataset AB. Both datasets are split into subsets with the following distribution: 70% is the training set $\mathcal{T}$, 20% is the test set $\mathcal{S}$, and 10% is the validation set $\mathcal{V}$. The training set for dataset A is $\mathcal{T}_A$ while the test set for dataset B is $\mathcal{S}_B$ and so forth.

Each file in the dataset is split into segments as described in Chapter 7. Five methods are used to extract different features from the segments as described in Chapter 4. These methods are the STFT, MFCC, IMFCC, GFCC and IGFCC. This results in five different models each trained with $\mathcal{T}_A$, $\mathcal{T}_B$ and $\mathcal{T}_{AB}$. Each model is trained 10 times with different random initial weights. This enables the computation of the 95 percent confidence interval of the performance w.r.t both model architecture and feature extraction method. All models and all initializations are tested on both $\mathcal{S}_A$ and $\mathcal{S}_B$. The model is evaluated based on the output of each block, referred to as *per block evaluation*. All tests are shown in Appendix D, Appendix E, and Appendix F.

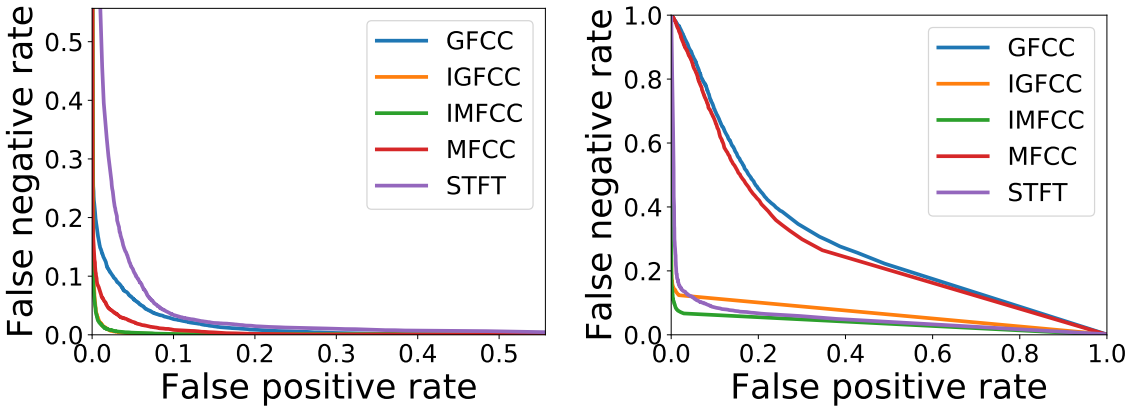
The accuracy of all models are shown in Table 8.2.

**Table 8.2:** Accuracy and confidence intervals in percent of the baseline model with different training sets, test sets, and feature extraction methods. $\mathcal{T}_A \rightarrow \mathcal{S}_A$ means the model was trained with the white box training set $\mathcal{T}_A$ and tested on the white box testing set $\mathcal{S}_A$, and so on.

| | STFT | MFCC | IMFCC | GFCC | IGFCC |
|---|---|---|---|---|---|
| $\mathcal{T}_A \rightarrow \mathcal{S}_A$ | 93.39 ± 0.60 | 96.73 ± 0.10 | 98.55 ± 0.10 | 94.39 ± 0.24 | 98.54 ± 0.14 |
| $\mathcal{T}_A \rightarrow \mathcal{S}_B$ | 91.67 ± 1.59 | 53.82 ± 3.11 | 93.18 ± 0.83 | 56.45 ± 2.44 | 89.82 ± 1.02 |
| $\mathcal{T}_B \rightarrow \mathcal{S}_A$ | 64.74 ± 1.13 | 57.96 ± 0.76 | 86.85 ± 1.44 | 58.58 ± 1.29 | 85.26 ± 2.48 |
| $\mathcal{T}_B \rightarrow \mathcal{S}_B$ | 99.40 ± 0.17 | 99.06 ± 0.14 | 99.75 ± 0.07 | 98.89 ± 0.24 | 99.72 ± 0.09 |
| $\mathcal{T}_{AB} \rightarrow \mathcal{S}_A$ | 92.89 ± 0.81 | 95.63 ± 0.35 | 98.15 ± 0.22 | 93.15 ± 0.48 | 98.00 ± 0.34 |
| $\mathcal{T}_{AB} \rightarrow \mathcal{S}_B$ | 99.42 ± 0.09 | 98.22 ± 0.27 | 99.56 ± 0.12 | 97.68 ± 0.31 | 99.46 ± 0.32 |

Table 8.2 shows that IMFCC and IGFCC in general performs best of the five feature extraction methods. It is seen that the IMFCC and IGFCC features reach accuracies above $98.00 \pm 0.34$ when the adversarial attack type in the test set is represented in the training set. It is also seen that all feature extraction methods reach accuracies above $92.89 \pm 0.81$ when the adversarial attack type in the test set is represented in the training set. When the adversarial attack type in the test set is not represented in the training set then the IMFCC and IGFCC features reach accuracies above $85.26 \pm 2.48$. In the case of $\mathcal{T}_B \rightarrow \mathcal{S}_A$ it can be seen that STFT, MFCC and GFCC performs significantly worse compared to the IMFCC and IGFCC. In this case the highest accuracy reached for the three feature extraction methods is $64.74 \pm 1.13$. From section 3.2 it was expected that IMFCC and IGFCC would perform better than MFCC and GFCC due to a higher resolution at higher frequencies. The results in Table 8.2 are consistent with this expectation.

As many of the accuracies between the feature extraction methods are close, it would be interesting to take a look at whether there is a trade-off between the number of false positives and false negatives. The accuracy measure does not report this but it can be inspected by plotting the false negative rate over the false positive rate as shown on Figure 8.2 for $\mathcal{T}_A \rightarrow \mathcal{S}_A$ and $\mathcal{T}_A \rightarrow \mathcal{S}_B$.




**(a)** $\mathcal{T}_A$ and $\mathcal{S}_A$.

**(b)** $\mathcal{T}_A$ and $\mathcal{S}_B$.

**Figure 8.2:** Mean false negative and false positive rates of the baseline implementation trained with $\mathcal{T}_A$ and evaluated on $\mathcal{S}_A$ and $\mathcal{S}_B$ for different feature extraction methods.

Figure 8.2a shows the same tendency as the accuracies for $\mathcal{T}_A \rightarrow \mathcal{S}_A$ since there are no intersections between the different curves. The STFT has the lowest accuracy for $\mathcal{T}_A \rightarrow \mathcal{S}_A$ and on Figure 8.2a it also has the highest false negative rate for all false positive rates. The GFCC has the second lowest accuracy and also has the second worst false negative rate for all false positive rates, and so on. This means that the accuracies for $\mathcal{T}_A \rightarrow \mathcal{S}_A$ are sufficient to compare each feature extraction method. In the case $\mathcal{T}_A \rightarrow \mathcal{S}_B$ it is seen in Figure 8.2b that there is an overlap for the STFT and the IGFCC feature extraction methods. This means that there is a tradeoff if either the false negative rate or the false positive rate is more important than the other. The accuracies in Table 8.2 are sufficient for comparing each feature extraction method, except for $\mathcal{T}_A \rightarrow \mathcal{S}_B$ where there is a slight trade-off between the STFT and IGFCC. This is seen in the false negative and false positive figures in Appendix D, Appendix E and Appendix F. The validation accuracies for the models trained with $\mathcal{T}_A$, $\mathcal{T}_B$ and $\mathcal{T}_{AB}$ can be found in Appendix D, Appendix E and Appendix F. All validation accuracies tend to converge without any overfitting.

The MFCC results are different from the results in [12]. The reason for this is not known, however it could be caused by the different implementation strategies. As written in Chapter 7 it was decided not to use zero-padding due to the bias it can apply if the distribution of file lengths for normal examples and for adversarial examples are not exactly the same. In addition to this, 20 filters for MFCC is used instead of the 40 used in [12] and the hyper parameters are not exactly the same.

In Appendix G a simple investigation is made of which blocks are misclassified. Here it was found that in the case $\mathcal{T}_A \rightarrow \mathcal{S}_A$ only a few of the networks misclassified the majority of the blocks from the same file. This means that the majority of the blocks of e.g. an adversarial file are correctly classified as adversarial. Thus it is possible to increase accuracy on a per file basis by processing the combined predictions of all blocks in each

file. Hence increase performance for $\mathcal{T}_A \rightarrow \mathcal{S}_A$ and maybe the others too.

## 8.4 Evaluation and Comparison of the Feature Extraction Methods - Per File

Given that the network only misclassify one or two blocks out of the many that constitutes a single file, it should be possible to modify the system in such a way that it makes a single prediction per file with no misclassifications. There are a couple of options that will be discussed.

One option is to continue splitting each file into blocks and make a prediction for each block. And then afterwards gather the predictions of all blocks that originate from the same file and combine them in a manner that reduces the number of misclassifications. There are many ways to do so but a simple method would be to average the predictions of all blocks in each file and consider the result as the prediction for each file. Another way is to consider an adversarial example absolutely detrimental and a must to be detected. With this in mind the prediction could also be to say that the file is adversarial if a single block in each file is predicted as adversarial. This method would vastly reduce the number of false negatives that could occur but at a cost of increasing the false positives.

Another option is to feed the files of varied lengths into the network directly and modify the network in such a manner that it can take variable size inputs. There are two distinctly different ways to achieve this. One is to use the Recurrent Neural Network (RNN). Another is to still use a CNN but apply a layer with adaptive pooling. Adaptive pooling is an operation where the pooling window size is dependent on the input size but has a fixed output size. Thus making it possible to give an input with variable size (one file) and get a single output from the network. If the adaptive pooling is an average pool then it is similar to the first option of averaging the outputs of each block but has the added flexibility that it can be used anywhere in the network and not just at the very end.

Both adaptive pooling and RNN have the issue that it is not possible to use batches of samples as this would require tensors with variable column widths which is not supported in any current popular deep learning library (PyTorch, Tensorflow, or Scikit-learn). The options are to use a batch size of 1 i.e. stochastic gradient descent or to fix the input sizes in some manner either by padding with zeros or clipping/splitting the data. The last of which seems very similar to the current implementation of splitting the data into blocks of fixed size. While being forced to use stochastic gradient descent to have a variable input size is not a disaster, it is certainly not ideal either.

Therefore the option to split the data into blocks and then averaging the predictions of each block into a single prediction for each file will be used. This is based on the fact that it is both simpler to implement and the fact that the baseline implementation using these blocks with mini-batch gradient descent has a proven ability to converge and produce

results with a near perfect accuracy. This is not a guarantee with a RNN or CNN with adaptive pooling that need to use stochastic gradient descent.

A test similar to section 8.3 is made with the only exception of evaluating the predictions on a per file basis, referred to as *per file evaluation*. Results from the tests are shown in Table 8.3.

**Table 8.3:** Accuracy in percent of the baseline model evaluated on a per file basis with different training sets, test sets, and feature extraction methods. $\mathcal{T}_A \rightarrow \mathcal{S}_A$ means the model was trained with the white box training set $\mathcal{T}_A$ and tested on the white box testing set $\mathcal{S}_A$, etc.

| | STFT | MFCC | IMFCC | GFCC | IGFCC |
|---|---|---|---|---|---|
| $\mathcal{T}_A \rightarrow \mathcal{S}_A$ | 97.58 ± 0.36 | 99.80 ± 0.19 | 99.83 ± 0.14 | 99.44 ± 0.21 | 99.75 ± 0.15 |
| $\mathcal{T}_A \rightarrow \mathcal{S}_B$ | 92.78 ± 2.13 | 52.03 ± 2.37 | 94.15 ± 0.84 | 53.89 ± 2.46 | 90.65 ± 1.02 |
| $\mathcal{T}_B \rightarrow \mathcal{S}_A$ | 65.84 ± 2.29 | 58.17 ± 0.98 | 93.65 ± 1.59 | 58.71 ± 1.62 | 92.20 ± 2.94 |
| $\mathcal{T}_B \rightarrow \mathcal{S}_B$ | 99.58 ± 0.14 | 99.72 ± 0.20 | 99.99 ± 0.03 | 99.66 ± 0.23 | 99.97 ± 0.05 |
| $\mathcal{T}_{AB} \rightarrow \mathcal{S}_A$ | 97.36 ± 0.46 | 99.75 ± 0.20 | 99.63 ± 0.21 | 98.96 ± 0.37 | 99.72 ± 0.19 |
| $\mathcal{T}_{AB} \rightarrow \mathcal{S}_B$ | 99.65 ± 0.07 | 99.21 ± 0.33 | 99.89 ± 0.08 | 98.96 ± 0.24 | 99.75 ± 0.22 |

Comparing Table 8.3 with Table 8.2 an increase in general performance can be seen. Also in the case of $\mathcal{T}_B \rightarrow \mathcal{S}_A$ an increase in performance is seen across all feature extraction methods. In the case of $\mathcal{T}_A \rightarrow \mathcal{S}_B$ a small decrease in performance is seen for the feature extraction methods MFCC and GFCC but this is not unexpected as their accuracies in Table 8.2 are similar to random guessing . The feature extraction method IMFCC is the best performing. For all cases the IMFCC feature reaches accuracies higher than $93.65 \pm 1.59$. This is a significant increase compared to the per block based evaluation where the IMFCC feature only reached an accuracy of $86.85 \pm 1.44$ in the case $\mathcal{T}_B \rightarrow \mathcal{S}_A$. This is important to notice as the per file based evaluation makes the baseline model, using IMFCC features, more robust to unseen adversarial attack types. The IGFCC feature extraction method is close to performing as well as the IMFCC. The largest difference of the two is in the case $\mathcal{T}_A \rightarrow \mathcal{S}_B$ where the IMFCC features reach an accuracy of $94.15 \pm 0.84$ and the IGFCC features only reach an accuracy of $90.65 \pm 1.02$. As for the per block based evaluation, MFCC and GFCC features have very low accuracies when the adversarial attack type in the test set is not represented in the training set. It is although seen that all five feature extraction methods reach accuracies higher than $97.36 \pm 0.46$ when the adversarial attack type in the test set is represented in the training set. This observation suggests that the model with the different feature extraction methods is able to detect adversarial examples for speech recognition when the type of attack is represented in the training set.

## Conclusion of the Evaluation and Comparison of the Feature Extraction Methods

It has now been investigated if the model and features are able to detect adversarial examples for speech recognition. Dataset A, dataset B and the combined dataset AB have been used for training the baseline model, and dataset A and B have been used for evaluation. The feature extraction methods described in Chapter 4 have individually been used to investigate how they affect the performance of the baseline model. The influence of using per block based evaluation and per file based evaluation has also been investigated. From this investigation it is found that the baseline model is able to detect adversarial examples in the audio domain, especially when using the IMFCC and IGFCC features, and using per file based evaluation. This can although only be said for the specific cases investigated in this chapter. In order to make broader statements it is necessary to further investigate the behaviour of the baseline model and the feature extraction methods. In the next chapter the separation of speech and nonspeech segments is investigated to find the contribution of each segment to the overall performance of the baseline model for each feature. This should give a better understanding into how the baseline model detects adversarial examples.

# 9 | Speech and Nonspeech

This chapter deals with an examination of the influence of the two segments, speech and nonspeech, for white box attacks only. To find this influence two overall tests are performed. One where the baseline model described in Chapter 8 is trained on white box data $\mathcal{T}_A$. The other test is where the baseline model is individually trained on the speech segments and the nonspeech segments of the white box dataset. In both test the speech and nonspeech segments are used individually as test sets. Performance for all feature extraction methods are found for the two tests.

## 9.1 Voice Activity Detection

In order to split each example in dataset A into speech and nonspeech segments it is necesarry to know when speech is present in each example. Voice Activity Detection (VAD) is a method used to detect when speech is present in an audio file. For this project the VAD method developed in [43] for python (rVAD_fast.py) will be used. Figure 9.1 shows an example of where the VAD method is used to detect when speech is present.

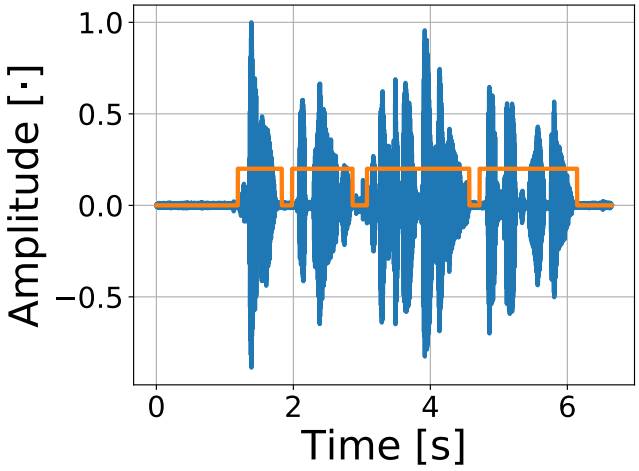



**Figure 9.1:** Example of detecting speech using VAD method [43]. Blue: example, orange: VAD

As it is seen in Figure 9.1 this VAD is able to detect when speech is present in an example. When the orange VAD line is not 0, speech is present. Although this VAD method is robust to noise it is chosen to only detect speech in the normal samples and not in the adversarial examples. This can be done because, as mentioned in section 3.2, the adversarial examples are generated from 300 original examples which are accessible. Thus it is possible to find the indices of the original examples where speech is present, and then use these indices to find where speech is present in the corresponding adversarial examples. How the speech segments are found in dataset A can be seen in Listing 9.1.

**Listing 9.1:** Code snippet of how speech regions are found in dataset A.

```
1  for n in range(n_clean_files):
2      print("file: " + str(n)+ " of: " +str(n_clean_files) + "-" + clean_files[n])
3      original = 0 # determines if current file is an original
4      clean_file = clean_files[n][-10:-4] # get clean file name
5      file_split = clean_files[n].split('/') # split clean file names

7      vad = get_rvad(clean_files[n]) # find where speech is present

9      out_name_speech = out_dir_speech+file_split[3][:-4]+"_speech.wav" # Where to store the
           speech file if clean file is not original
10     out_name_silence = out_dir_silence+file_split[3][:-4]+"_silence.wav" # Where to store the
           nonspeech file if clean file is not original

12     for m in range(n_files):
13         if clean_file in file_names[m] and "adv" in file_names[m]: # clean file is original
14             original = 1
15             file_split = file_names[m].split('/')
16             out_name_speech = out_dir_speech+file_split[3][:-4]+"_speech.wav" # where to store
                   speech of adversarial file
17             out_name_silence = out_dir_silence+file_split[3][:-4]+"_silence.wav" # where to
                   store nonspeech of adversarial file
18             SnS_split(vad, file_names[m], out_name_speech, out_name_silence) # split
                   adversarial example into speech and nonspeech



21     if original == 0:
22         SnS_split(vad, clean_files[n], out_name_speech, out_name_silence) # split normal
               example into speech and nonspeech
```

In Listing 9.1 the outer loop runs trough all the normal examples in the dataset, including the original examples. Line 7 finds the indices when speech is present in the example and line 12 to 18 handles if the current normal example has corresponding adversarial examples. If this is the case the current normal example is an original example and will therefore not be split into speech and nonspeech segments. In Listing 9.2 a code snippet is seen of the SnS_split() function which is used in Listing 9.1.

**Listing 9.2:** Code snippet of how speech regions are found in entire dataset.

```
1  def SnS_split(vad, file_name, out_dir_speech, out_dir_silence):
2      sr, data = scipy.io.wavfile.read(file_name) #load file
3      max_sample = np.max(data)
4      data = data/max_sample #normalize
5      speech = []
6      silence = []
7      for n in range(len(data)):
8          if vad[n] == 1: # if speech is present at index n
9              speech.append(data[n])
10         else:
11             silence.append(data[n])

13     scipy.io.wavfile.write(out_dir_speech, sr,
14         (np.asarray(speech)*max_sample).astype(np.int16)) #save speech wav file
15     scipy.io.wavfile.write(out_dir_silence, sr,
```

```
16          (np.asarray(silence)*max_sample).astype(np.int16)) #save nonspeech wav file
```

The dataset has been split into speech and nonspeech segments and it is now possible make speech and nonspeech training, validation and test sets. The performance of the baseline model with the five feature extraction methods is evaluated on the two test sets for speech and nonspeech, $\mathcal{S}_S$ and $\mathcal{S}_{NS}$.

## 9.2 Training A - Testing Speech and Nonspeech

In the following section the performance is found for testing on the two segments, speech and nonspeech. The model used are the baseline model, found in Chapter 8 trained with $\mathcal{T}_A$. Dataset A is split into subsets with the following distribution: 70% is the training set $\mathcal{T}$, 20% is the test set $\mathcal{S}$, and 10% is the validation set $\mathcal{V}$. The performance is evaluated for the speech segments in $\mathcal{S}_A$ called $\mathcal{S}_S$ and for the nonspeech segments in $\mathcal{S}_A$ called $\mathcal{S}_{NS}$.

The performance for the five feature extraction methods are found; STFT, MFCC, IMFCC, GFCC and IGFCC. This results in five different models trained with $\mathcal{T}_A$, one for each feature extraction method. Each model is trained 10 times with different random initial weights. This enables the computation of the 95 percent confidence interval of the performance w.r.t both model architecture and feature extraction method. All models and all initializations are tested on $\mathcal{S}_S$ for speech and $\mathcal{S}_{NS}$ for nonspeech. For this test per block based evaluation is used because it is not guaranteed that each file will contain equal amounts of speech and nonspeech. It is observed that speech is in general more present in the examples, thus each speech file will have more blocks to base an averaged evaluation on. The full description of the test is seen in Appendix H.

For the feature extraction methods STFT, MFCC and GFCC, it is expected that the performance is significantly better for testing on nonspeech than for testing on speech. This is due to the speech segments high amplitude component in the low frequency area as seen in section 3.2. However for the feature extraction methods IMFCC and IGFCC, it is expected that the performance for nonspeech is only slightly higher than for speech as there appear fewer high amplitude speech components at higher frequencies.

The accuracies from the aforementioned tests can be seen in Table 9.1, together with a comparison of $\mathcal{T}_A \rightarrow \mathcal{S}_A$ from Table 8.2. These are sufficient to compare the different feature extraction methods as shown by the plots in Appendix H.

**Table 9.1:** Accuracy in percent of the baseline model with different training sets, test sets, and feature extraction methods. $\mathcal{T}_A \rightarrow \mathcal{S}_S$ means the model was trained with the white box training set $\mathcal{T}_A$ and tested on the white box testing set only speech $\mathcal{S}_S$ only nonspeech $\mathcal{S}_{NS}$.

| | STFT | MFCC | IMFCC | GFCC | IGFCC |
|---|---|---|---|---|---|
| $\mathcal{T}_A \rightarrow \mathcal{S}_A$ | 93.39 ± 0.60 | 96.73 ± 0.10 | 98.55 ± 0.10 | 94.39 ± 0.24 | 98.54 ± 0.14 |
| $\mathcal{T}_A \rightarrow \mathcal{S}_S$ | 83.88 ± 1.19 | 95.88 ± 0.21 | 97.77 ± 0.19 | 93.29 ± 0.45 | 97.67 ± 0.12 |
| $\mathcal{T}_A \rightarrow \mathcal{S}_{NS}$ | 97.11 ± 0.50 | 98.15 ± 0.23 | 99.55 ± 0.14 | 97.12 ± 0.44 | 99.67 ± 0.14 |

As expected for all feature extraction methods the accuracies for testing on $\mathcal{S}_{NS}$ are higher than for testing on $\mathcal{S}_S$. In section 3.2 it was found that for all adversarial examples an increase in amplitude can be seen around a spoken word or sentence. This is true for all white box attacks. When comparing the original recording with any related adversarial example an increase in noise is seen across all frequencies. However, the highest amplitude is still found around the spoken word or sentence. When visually inspecting the different spectrograms and time domain representations in section 3.2 it is easiest to find a difference between the plots by looking at the areas with no speech component. This is in accordance with what is seen in Table 9.1, that it is easier for the model to detect $\mathcal{S}_{NS}$ than $\mathcal{S}_S$. When comparing to the case $\mathcal{T}_A \rightarrow \mathcal{S}_A$ it is seen that for testing on $\mathcal{S}_S$ the accuracies are lower for all feature extraction methods, but when testing on $\mathcal{S}_{NS}$ the accuracies are higher for all feature extraction methods. This also fits with what is expected as $\mathcal{S}_A$ contains both speech and nonspeech segments.

From the spectrograms in section 3.2 it was also found that the highest amplitudes from the speech component are found in the lower frequencies for both normal and adversarial examples. The general noise change between a normal example and an adversarial example are therefore mostly seen in the higher frequencies. It therefore corresponds well with what is seen in the test, namely that for testing on $\mathcal{S}_S$ the IMFCC and IGFCC performs better than MFCC and GFCC. In Table 9.1 all the cepstrum based feature extraction methods for both $\mathcal{S}_S$ and $\mathcal{S}_{NS}$ reaches higher accuracies than $93.29 \pm 0.45$. For the IMFCC and IGFCC features the baseline model reaches accuracies higher than $97.67 \pm 0.12$ for both $\mathcal{S}_S$ and $\mathcal{S}_{NS}$. As can be seen from Table 9.1 it is in general still possible to detect adversarial examples even if only speech or nonspeech is present. This is especially true for the IMFCC and IGFCC features.

It was attempted to perform a similar test using the black box dataset described in section 3.2. This was although not trivial as the files in the black box dataset are significantly shorter than most in the white box dataset. To solve this it was attemped to reduce the block size and block step size in Table 7.1 from 512 ms to 128 ms. This however conflicted with the architecture of the baseline mode described in Chapter 8. After the convolution in the third convolutional layer in the baseline model the feature maps become vectors when the block size is 128 ms. Therefore it was not possible to perform the last $2 \times 2$

convolutional layer. It was therefore decided to only use the white box dataset when investigating the influence of the speech and nonspeech segments.

It has now been investigated how the performance of the baseline model trained with $\mathcal{T}_A$ for the five feature extraction methods is affected when tested on only speech and nonspeech segments of dataset A. It is now further investigated how it affects the performance of the baseline model and the five feature extraction methods when trained and tested individually on the speech and nonspeech segments of dataset A.

## 9.3 Training Speech and Nonspeech - Testing Speech and Nonspeech

The following section contains further investigation and performance evaluation of the speech and nonspeech segments. This is to determine the ability to detect if a speech or nonspeech segment is adversarial using the same and the opposite segment for training. This is done by training on speech or nonspeech and evaluate performance on two test sets, one set is the opposite segment and one is the same segment. If the model is trained with speech, then the performance is found for speech and nonspeech. This results in two models, one trained with speech and one trained with nonspeech. The performance of the two models will be evaluated with the five feature extraction methods. The same procedure as in section 9.2 (Appendix H) is used in the following evaluation. This test is also performed using per block based evaluation.

The accuracies from the performed test, can be seen in Table 9.2, together with a comparison of $\mathcal{T}_A \rightarrow \mathcal{S}_A$ from Table 8.2.

**Table 9.2:** Accuracy in percent of the baseline model with different training sets, test sets, and feature extraction methods. $\mathcal{T}_{SN} \rightarrow \mathcal{S}_S$ means the model was trained with the white box training set segmented into nonspeech $\mathcal{T}_{SN}$ and tested on the white box testing set segmented into only speech $\mathcal{S}_S$.

| | STFT | MFCC | IMFCC | GFCC | IGFCC |
|---|---|---|---|---|---|
| $\mathcal{T}_A \rightarrow \mathcal{S}_A$ | 93.39 ± 0.60 | 96.73 ± 0.10 | 98.55 ± 0.10 | 94.39 ± 0.24 | 98.54 ± 0.14 |
| $\mathcal{T}_S \rightarrow \mathcal{S}_S$ | 90.36 ± 0.55 | 96.09 ± 0.23 | 98.07 ± 0.19 | 93.68 ± 0.34 | 97.93 ± 0.10 |
| $\mathcal{T}_S \rightarrow \mathcal{S}_{NS}$ | 84.05 ± 6.09 | 91.18 ± 1.38 | 98.55 ± 0.40 | 89.07 ± 2.02 | 98.80 ± 0.23 |
| $\mathcal{T}_{NS} \rightarrow \mathcal{S}_{NS}$ | 96.24 ± 0.32 | 98.64 ± 0.23 | 99.71 ± 0.11 | 97.51 ± 0.23 | 99.76 ± 0.13 |
| $\mathcal{T}_{NS} \rightarrow \mathcal{S}_S$ | 61.63 ± 1.39 | 84.80 ± 1.45 | 90.05 ± 0.93 | 79.08 ± 1.10 | 89.98 ± 0.70 |

From Table 9.2 it is evident that the characteristics gained from speech is more important than the characteristics for nonspeech. Both for testing on $\mathcal{S}_S$ and $\mathcal{S}_{NS}$ the training set $\mathcal{T}_S$ makes the baseline model gain higher accuracies for all features than for training

on $\mathcal{T}_{NS}$. For training on $\mathcal{T}_S$ the IMFCC and IGFCC features make the baseline model reach accuracies higher than $97.93 \pm 0.10$ for both testing on $\mathcal{S}_S$ and $\mathcal{S}_{NS}$. For training on $\mathcal{T}_{NS}$ the same two features only makes the baseline model reach accuracies higher than $89.98 \pm 0.70$. The main observation for arguing that the characteristics gained from speech is more important than the characteristics for nonspeech is seen in the cases when the baseline model is trained on one type of segments and tested on the opposite. For $\mathcal{T}_S \rightarrow \mathcal{S}_{NS}$ the IMFCC and IGFCC features makes the baseline model reach accuracies higher than $98.55 \pm 0.40$. For $\mathcal{T}_{NS} \rightarrow \mathcal{S}_S$ the best performing feature is the IMFCC which only reaches an accuracy of $90.05 \pm 0.93$ It is also possible to see that IMFCC and IGFCC in general performs best in all cases.

In the case $\mathcal{T}_{NS} \rightarrow \mathcal{S}_S$ it can easily be seen that IMFCC and IGFCC performs best, however overlapping each other. In the case of $\mathcal{T}_S \rightarrow \mathcal{S}_{NS}$ it is difficult to see IMFCC and IGFCC as they perform very well and superior to the three other feature extraction methods.

In both Table 9.1 and Table 9.2 the STFT feature has the worst performance both for testing on $\mathcal{S}_S$ and $\mathcal{S}_{NS}$. Here its performance is significantly worse than all other feature extraction methods only reaching an accuracy of $61.63 \pm 1.39$. This can be due to several things. It is possible that the characteristics from nonspeech does not translate well to speech. However for the STFT, $\mathcal{T}_S \rightarrow \mathcal{S}_{NS}$ is superior by 27 % compared to $\mathcal{T}_{NS} \rightarrow \mathcal{S}_S$, which would suggest there is some similarity in the adversarial noise for nonspeech and speech. It could be due to the amount of data available and the removal of too small blocks as described in Appendix A. After splitting the speech and nonspeech segments into blocks as described in Chapter 7, 8779 blocks are made for speech and 4145 blocks for nonspeech. Furthermore, 335 blocks are removed from nonspeech as they are not large enough to constitute a certain block size, see Table 7.1. This difference in available data could results in a difference in performance between $\mathcal{T}_S \rightarrow \mathcal{S}_{NS}$ and $\mathcal{T}_{NS} \rightarrow \mathcal{S}_S$. The STFT is especially susceptible to an insufficient amount of data due to its much higher frequency resolution compared to the other feature extraction methods, see Chapter 7.

## Speech and Nonspeech Conclusion

It has now been investigated how the performance of the baseline model trained with $\mathcal{T}_A$, $\mathcal{T}_S$ and $\mathcal{T}_{NS}$ for the five feature extraction methods is affected when tested on only speech and nonspeech segments of dataset A. It was found that for these tests the two feature extraction methods IMFCC and IGFCC are the best performing and that the STFT feature extraction method is the worst performing. The baseline model reached higher accuracies when testing on nonspeech segments than on speech segments for all feature extraction methods and all training sets. This fits with the observation in section 3.2 from where it was expected that the nonspeech segments would be easier to classify. It was also observed that the information gained from training on speech segments generalizes better than when training on nonspeech segments. For the IMFCC and IGFCC features the accuracies for training on $\mathcal{T}_S$ were similar as for training on $\mathcal{T}_A$ which contains both

speech and nonspeech segments. It is now of interest to investigate the baseline models and the five feature extraction methods' robustness to noise. This is done in Chapter 10.

# 10 | Robustness to Noise

The purpose of this chapter is to investigate the model's and the different feature extractions' robustness to noise. It is shown in Chapter 8 that the model can accurately predict whether adversarial noise is present or not. However, there is no guarantee that the model has learned some characteristics that are specific to adversarial noise or if the presence of noise would completely inhibit its ability to detect adversarial examples. That would be a flaw in the model and motivates an investigation of the robustness to noise.

In order to get a complete and accurate evaluation of the robustness to noise, it is necessary to test for all use-case scenarios: All the types of noise that could occur at all the different volumes they could occur at. However, it is impossible to know all types of noise that could occur and their volume. It is necessary to take a subset of noise types and volumes that ideally should reflect the actual distribution. Thus the common practice for choosing the subset of noise types and volumes to test with will be described. This will ensure the results of the tests accurately represent the real use-case scenario.

To ensure the tests represent a complete view of the real use-case scenario, the following tests will be made:

1. **Clean training - noise testing:** The model is trained without the addition of noise and tested with a specific type of noise added to the white box dataset at different SNRs.
2. **Known single noise type and SNR:** The model is trained with a specific type of noise present at a specific SNR and tested with the same noise type present and at the same SNR.
3. **Known single noise type:** The model is trained with a specific type of noise present at many SNRs and tested with the same noise type present at different SNRs.
4. **Multiple noise types training - known noise type at testing:** The model is trained with many types of noise present at many SNRs and tested with a known noise type present at different SNRs.
5. **Multiple noise types training - unknown noise type at testing:** The model is trained with many types of noise present at many SNRs and tested on an unknown noise type present at different SNRs.

The first determines whether the model with different feature extraction methods has learned characteristics that are specific to the adversarial noise or not. It also shows the robustness when the scenarios for training and testing are vastly different. The next two

tests give insight into the robustness when the use-case scenario is well known and the training and testing scenarios align. The final two tests give insight into the robustness of the model that is trained for many types of noise, both when the actual use-case align with the training scenario and when it does not.

It is accepted that the files with added noise might not be correctly transcribed by DeepSpeech.

## 10.1 Choice of Noise Types and Volumes for Evaluating Noise Robustness

When evaluating a model's robustness to noise, it is important that the evaluation give a complete and accurate representation of the actual use-case scenario. As the use-case scenario for detecting adversarial noise for a speech recognition system has not been defined, it is difficult to choose noise types and volumes that accurately represent the use-case. Therefore the noise types and volumes will be chosen similarly to [44]. Noise from the CHiME 3 dataset [45] will be added to the white box dataset at the following SNRs: 0 dB, 5 dB, 10 dB, 15 dB, and 20dB. The cafeteria noise is always used as the test set and will be denoted as $\mathcal{S}_{0\text{dB}}$, $\mathcal{S}_{5\text{dB}}$, $\mathcal{S}_{10\text{dB}}$, $\mathcal{S}_{15\text{dB}}$, and $\mathcal{S}_{20\text{dB}}$.

The CHiME 3 dataset contains noise signals that are already split into a training file, a validation file and a test file.

The training sets for test 2, 3, 4 and 5 are generated as follows:

- Test 2: The cafeteria noise training file will be added to each file in the white box dataset. A noisy training set is generated for each of the five SNRs: 0 dB, 5 dB, 10 dB, 15 dB, and 20 dB. This training set will be denoted as $\mathcal{T}_{0\text{dB}}$ for 0 dB, etc.
- Test 3: The previous five datasets $\mathcal{T}_{0\text{dB}}$, $\mathcal{T}_{5\text{dB}}$, $\mathcal{T}_{10\text{dB}}$, $\mathcal{T}_{15\text{dB}}$, $\mathcal{T}_{20\text{dB}}$ and the white box training set $\mathcal{T}_A$ are combined into a single training set $\mathcal{T}_{\text{caf}}$.
- Test 4: For each file in the white box dataset, a single noise type is added by uniformly and independently selecting between the six noise types: BUS (bus noise), STR (street noise), PED (pedestrian noise), BBL (babble noise), SSN (speech shaped noise), and CAF (cafeteria noise). This is repeated five times, one for each of the five SNRs: 0dB, 5dB, 10dB, 15dB and 20dB. This way five SNR dependent training sets are generated where in each of them all six noise types are present. The training set $\mathcal{T}_{6\text{noise}}$ is then the combination of these five training sets and dataset A which has added no noise.
- Test 5: The procedure is the exact same as test 4 but excluding the cafeteria noise and is denoted as $\mathcal{T}_{5\text{noise}}$.

The SNR is calculated using a similar method as described in [46]. The RMS value of the signal is calculated using only the speech part of the signal. The speech part is found using the same VAD method [43] as in Chapter 9. Once the RMS value of the signal's speech part is calculated the noise is scaled to have an RMS value that corresponds to the desired SNR. Once the noise is added, the noisy signal is scaled to have the same maximum amplitude as the signal had before adding noise to avoid the case where the addition of noise causes clipping as shown in Listing B.1 in Appendix B.

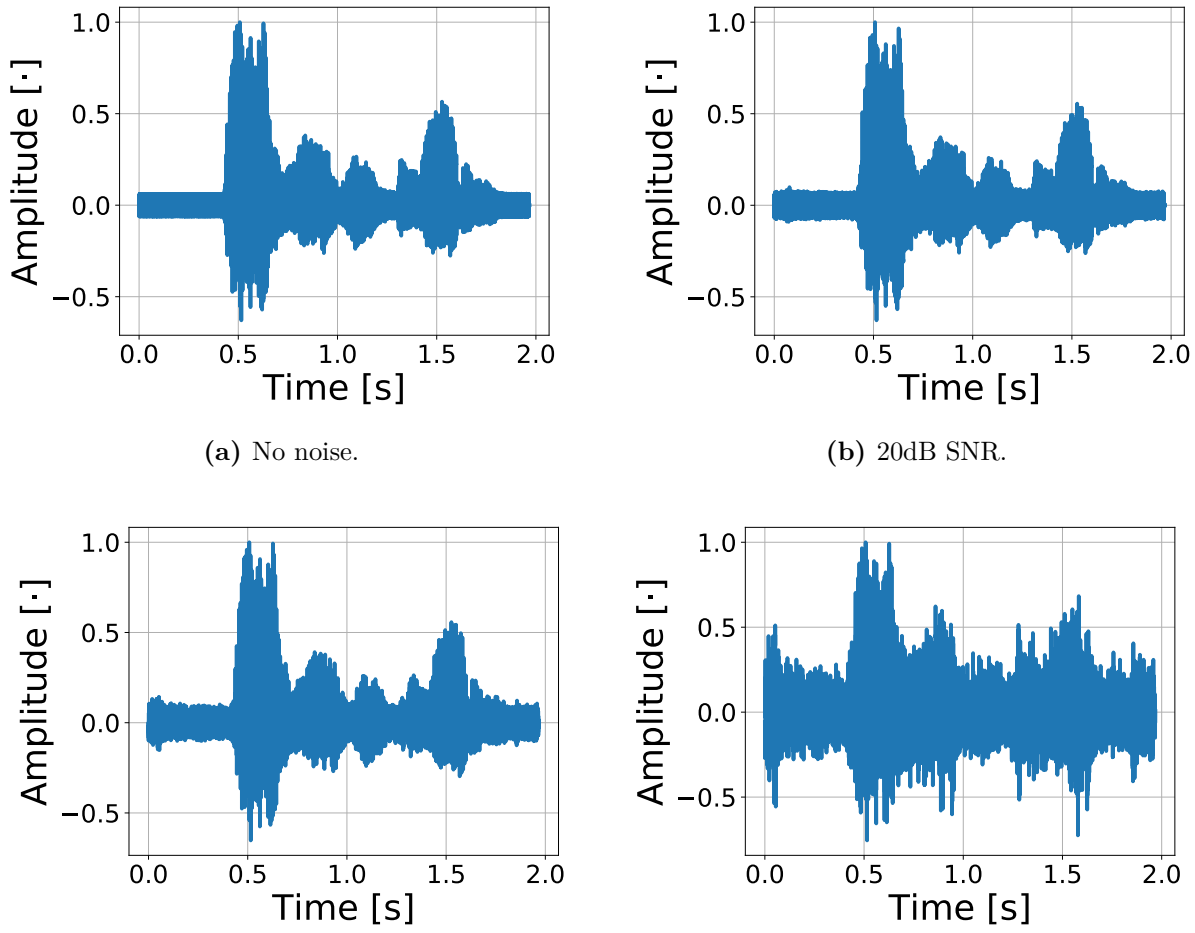
For obtaining the noisy test set (containing white box adversarial attacks with the cafeteria noise added), the exact same method is used for adding the cafeteria noise test file to the white box dataset as for test 2.

As all noise files are longer than the files in the white box dataset, it is necessary to only take a segment of the noise file for adding to a white box file. This is handled the following way: For a given file in the white box dataset, a random start time with uniform distribution for the noise file is chosen. Noise of an equal length to the white box audio file is taken from the random start time and added to the white box audio file. If the combination of the random start time and the white box file length exceeds the length of the cafeteria noise file, the remaining noise is taken from the beginning of the cafeteria noise file. The full description of how noise is added to the dataset is found in Appendix B.

## Analysis of White Box Test Set with Cafeteria Noise

It is desired to investigate the noisy test sets ($\mathcal{S}_{0\text{dB}}$, $\mathcal{S}_{5\text{dB}}$, $\mathcal{S}_{10\text{dB}}$, $\mathcal{S}_{15\text{dB}}$, and $\mathcal{S}_{20\text{dB}}$) to be able to evaluate if the results seem reasonable. It is of interest to investigate how the normal and adversarial files have changed in the test set with the addition of cafeteria noise.

In Figure 10.1 four signals are seen. The signal in Figure 10.1a is an adversarial example without added noise and this signal is present in the other three signals in Figure 10.1b, Figure 10.1c and Figure 10.1d. In Figure 10.1b cafeteria noise is added to the adversarial examples with an SNR of 20dB and for Figure 10.1c and Figure 10.1d the SNRs are 10dB and 0dB. It is seen in Figure 10.1 that the adversarial example is present in all four signals but as the SNR decreases so does the similarity to the adversarial example in Figure 10.1a. The adversarial example in Figure 10.1a is the same adversarial example seen in Figure 3.3d.




**(a)** No noise.

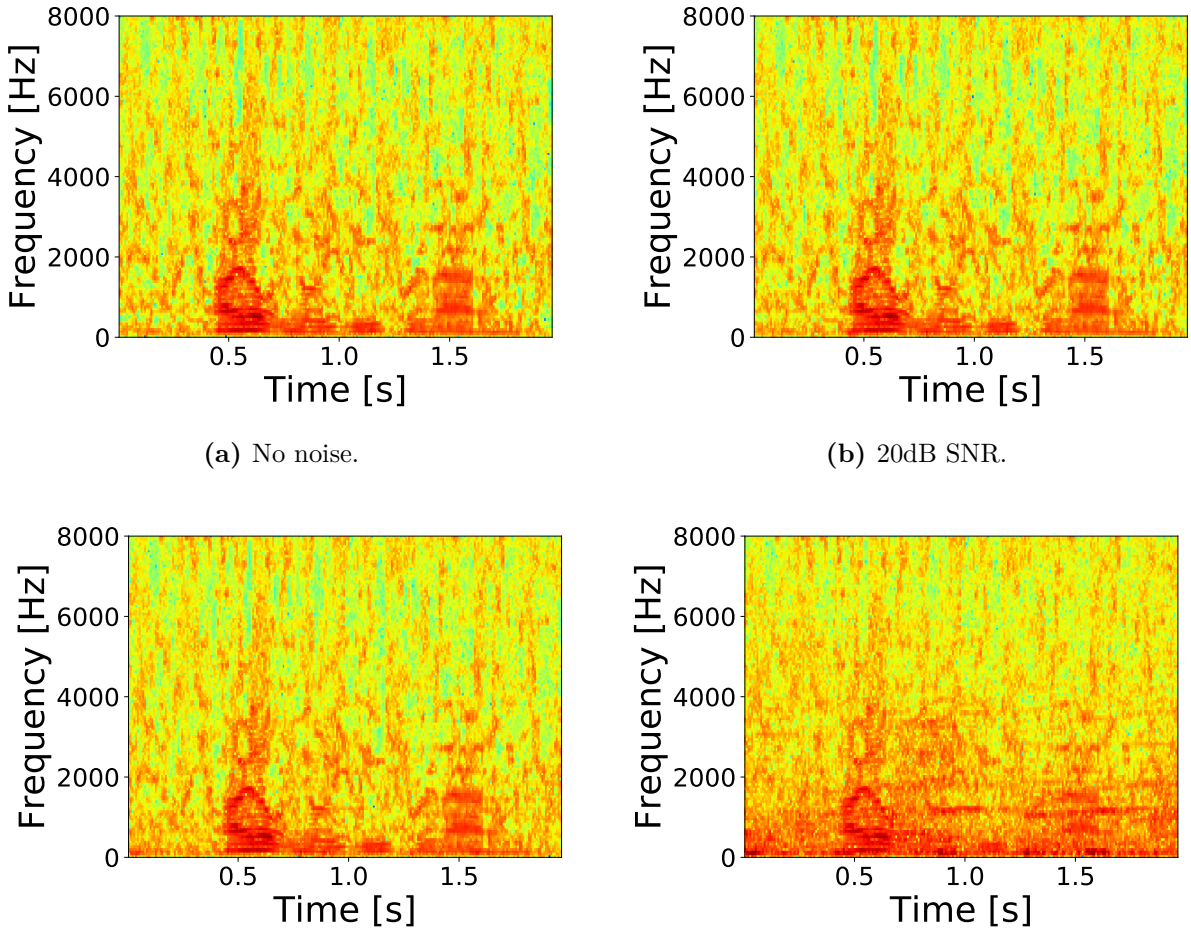
**(b)** 20dB SNR.

**(c)** 10dB SNR.

**(d)** 0dB SNR.

**Figure 10.1:** Time domain representation of an adversarial example with different amounts of additional noise.

It is desired to investigate the spectrograms of the noisy files to gain insight into how the different feature extraction methods will perform when cafeteria noise is present in the test set, see Figure 10.2.




**(a)** No noise.

**(b)** 20dB SNR.

**(c)** 10dB SNR.

**(d)** 0dB SNR.

**Figure 10.2:** Spectrograms of an adversarial example with different amounts of additional noise.

In Figure 10.2 four spectrograms are seen which corresponds to the four signal in Figure 10.1. In the four spectrograms no significant change is seen for the higher frequencies as the SNR decreases. However a change is seen for lower frequencies as the SNR decreases. This indicates that the feature extraction methods MFCC and GFCC might perform worse when cafeteria noise is present in the test set compared to their performance in Table 8.3 for $\mathcal{T}_A \rightarrow \mathcal{S}_A$.

## 10.2 Evaluating the Model and Feature Extraction Methods on Noisy Test Set

It is desired to investigate how it affects the performance of the model and the different feature extraction methods when noise is present only in the test and when noise is present in the training, validation and test set. This is investigated by performing five tests:

1. **Clean training - noise testing:** Train a model using $\mathcal{T}_A$. Test the model using the five test sets $\mathcal{S}_{0\text{dB}}$, $\mathcal{S}_{5\text{dB}}$, $\mathcal{S}_{10\text{dB}}$, $\mathcal{S}_{15\text{dB}}$ and $\mathcal{S}_{20\text{dB}}$ individually. Here the model is trained on a specific case and the actual use-case scenario is different. From this test it is expected that the accuracies will decrease compared to $\mathcal{T}_A \rightarrow \mathcal{S}_A$ in Table 8.3 for all feature extraction methods testing on all SNRs. This is expected as a test sets, such as $\mathcal{S}_{20\text{dB}}$, will contain data which differs more from $\mathcal{T}_A$ compared to the data in $\mathcal{S}_A$.

2. **Known single noise type and SNR:** Train a model for each of the five SNR dependend training sets described in section 10.1: $\mathcal{T}_{0\text{dB}}$, $\mathcal{T}_{5\text{dB}}$, $\mathcal{T}_{10\text{dB}}$, $\mathcal{T}_{15\text{dB}}$ and $\mathcal{T}_{20\text{dB}}$. Test each of the five models using the SNR corresponding test set described in section 10.1: $\mathcal{S}_{0\text{dB}}$, $\mathcal{S}_{5\text{dB}}$, $\mathcal{S}_{10\text{dB}}$, $\mathcal{S}_{15\text{dB}}$ and $\mathcal{S}_{20\text{dB}}$. The model is trained on a specific type and volume of noise and the actual use-case scenario is the exact same specific type and volume. From this test it is expected that all accuracies will increase compared to the results from test 1 although the accuracies might still be lower than for $\mathcal{T}_A \rightarrow \mathcal{S}_A$ in Table 8.3. This is expected as the test set should be better represented in the training set when the two sets have same SNRs.

3. **Known single noise type:** Train one model using the combined training set $\mathcal{T}_{\text{caf}}$ containing the six training sets: $\mathcal{T}_{0\text{dB}}$, $\mathcal{T}_{5\text{dB}}$, $\mathcal{T}_{10\text{dB}}$, $\mathcal{T}_{15\text{dB}}$, $\mathcal{T}_{20\text{dB}}$ and $\mathcal{T}_A$. Test the model using the five test sets $\mathcal{S}_{0\text{dB}}$, $\mathcal{S}_{5\text{dB}}$, $\mathcal{S}_{10\text{dB}}$, $\mathcal{S}_{15\text{dB}}$ and $\mathcal{S}_{20\text{dB}}$ individually. Here the model is trained on a wide set of SNRs and tested on a specific SNR within the set. From this test it is expected that the accuracies will decrease compared to the results from test 2. This is expected as the SNR in the test set will be a smaller percentage of the training set. This behaviour is although difficult to predict as the amount of training data for this test will be six times larger than for all previous tests.

4. **Multiple noise types at training - known noise type at testing:** Train one broad model using the training set $\mathcal{T}_{6\text{noise}}$ that has all noise types described in [44] in the training set. Test the model using the five test sets $\mathcal{S}_{0\text{dB}}$, $\mathcal{S}_{5\text{dB}}$, $\mathcal{S}_{10\text{dB}}$, $\mathcal{S}_{15\text{dB}}$ and $\mathcal{S}_{20\text{dB}}$ individually. This shows the case for attempting to train a general model and the actual use-case is covered in the training set. From this test it is expected that the accuracies will in general decrease compared to the results from test 3. This is expected as the type of noise present in the test set will be less represented in the training set.

5. **Multiple noise types training - unknown noise type at testing:** Train one broad model using the training set $\mathcal{T}_{5\text{noise}}$ that has all noise types described in [44] except for the cafeteria noise in the training set. Test the model using the five test sets $\mathcal{S}_{0\text{dB}}$, $\mathcal{S}_{5\text{dB}}$, $\mathcal{S}_{10\text{dB}}$, $\mathcal{S}_{15\text{dB}}$ and $\mathcal{S}_{20\text{dB}}$ individually. This shows the case for attempting to train a general model and the actual use-case is not covered in the training set. From this test it is expected that that the accuracies will in general decrease compared to the results from test 4. This is expected as the noise type present in the test set will not be present in the training set.

For all five tests it is expected that the accuracy will decrease as the SNR in the test set decreases, i.e. the accuracies will be lower for testing on $\mathcal{S}_{5\text{dB}}$ than for testing on $\mathcal{S}_{10\text{dB}}$. For all five tests it is also expected that the MFCC and GFCC feature extraction methods will have the lowest accuracies. This is because it is seen in Figure 10.2 that by adding cafeteria noise the lower frequencies are affected more than the higher.

For all of these five noise tests the data is split into training, validation and test set the following way: The noise training file is added to the white box dataset and from this dataset 70% is used for training set and 10% is used for validation set. The noise test file is added to the white box dataset and 20% is used for the test set. Each file can only be chosen to one of the three sets, i.e. the model can not be tested on a file which was used for training or validation. In all five tests all models are trained and tested for the five feature extraction methods STFT, MFCC, IMFCC, GFCC, IGFCC. In all five tests all models are trained 10 times to be able to calculate the 95 percent confidence interval as described in Chapter 5. All tests are evaluated using per file based evaluation.

The results from each test will be discussed as the results are presented but the comparison between the different test will be discussed after all results are presented in section 10.8.

## 10.3 Clean Training - Noise Testing

The first evaluation of how noise affects the performance of the baseline model and the feature extraction methods will focus on only adding noise to the test set. This is done to investigate if the baseline model trained with dataset A is still able to detect adversarial examples when noise is added to both the normal examples and the adversarial examples.

In this test the baseline model described in Chapter 8 is trained using $\mathcal{T}_A$ and evaluated on the five noise test sets: $\mathcal{S}_{0\text{dB}}$, $\mathcal{S}_{5\text{dB}}$, $\mathcal{S}_{10\text{dB}}$, $\mathcal{S}_{15\text{dB}}$ and $\mathcal{S}_{20\text{dB}}$. The full test description is found in Appendix I. In Table 10.1 the accuracies are shown when evaluating the baseline model with the five noise test sets and with $\mathcal{S}_A$ for comparisons sake.

**Table 10.1:** Accuracy in percent of the baseline model trained on $\mathcal{T}_A$ with different test sets, and feature extraction methods. $\mathcal{T}_A \rightarrow \mathcal{S}_{0\text{dB}}$ means the model was trained with the white box training set $\mathcal{T}_A$ and tested on the noisy test set $\mathcal{S}_{0\text{dB}}$.

| | STFT | MFCC | IMFCC | GFCC | IGFCC |
|---|---|---|---|---|---|
| $\mathcal{T}_A \rightarrow \mathcal{S}_A$ | 97.58 ± 0.36 | 99.80 ± 0.19 | 99.83 ± 0.14 | 99.44 ± 0.21 | 99.75 ± 0.15 |
| $\mathcal{T}_A \rightarrow \mathcal{S}_{0\text{dB}}$ | 78.57 ± 2.58 | 53.03 ± 1.16 | 64.66 ± 0.93 | 50.84 ± 0.52 | 65.65 ± 0.63 |
| $\mathcal{T}_A \rightarrow \mathcal{S}_{5\text{dB}}$ | 86.91 ± 2.47 | 62.58 ± 2.18 | 70.45 ± 1.45 | 58.48 ± 1.96 | 71.94 ± 1.19 |
| $\mathcal{T}_A \rightarrow \mathcal{S}_{10\text{dB}}$ | 90.67 ± 1.25 | 72.08 ± 2.06 | 79.55 ± 1.53 | 66.07 ± 1.65 | 81.66 ± 0.85 |
| $\mathcal{T}_A \rightarrow \mathcal{S}_{15\text{dB}}$ | 93.65 ± 0.63 | 85.31 ± 0.83 | 89.27 ± 0.87 | 80.42 ± 2.93 | 89.35 ± 0.90 |
| $\mathcal{T}_A \rightarrow \mathcal{S}_{20\text{dB}}$ | 95.06 ± 0.48 | 89.30 ± 1.17 | 93.85 ± 0.81 | 85.90 ± 1.65 | 94.02 ± 0.74 |

In Table 10.1 it is seen that for all five cafeteria noisy test sets the STFT feature extraction method makes the baseline model obtain the highest accuracy. It is also seen that as the SNR increases the two feature extraction methods IMFCC and IGFCC get closer to performing as well as the STFT.

For all noise SNR's the two feature extraction methods MFCC and GFCC obtains the worst performance of the five feature extraction methods as expected. By comparing Table 8.3 and Table 10.1, it is confirmed that the MFCC and GFCC feature extraction methods have the largest decrease in performance as the SNR decreases. Table 10.1 confirms the expectation that when training the baseline model on dataset A and testing on the noise data sets the accuracy decreases as the SNR decreases. It is seen that in the worst case scenario when the SNR is 0dB the best performing feature extraction method STFT, obtains an accuracy of $78.57 \pm 2.58$. For 0dB SNR the other IMFCC and IGFCC feature extraction methods reaches accuracies around 65 % and the MFCC and GFCC reaches accuracies close to random guessing.

## 10.4 Known Single Noise Type and SNR

In this section it is tested how the performance of the baseline model and the different feature extraction methods are affected when the baseline model is trained and tested with cafeteria noise for the five feature extraction methods. How the cafeteria noise is added to the training and the testing set is described in section 10.1. The baseline model is trained individually on the five SNR dependent training sets: $\mathcal{T}_{0\text{dB}}$, $\mathcal{T}_{5\text{dB}}$, $\mathcal{T}_{10\text{dB}}$, $\mathcal{T}_{15\text{dB}}$ and $\mathcal{T}_{20\text{dB}}$. Thus five models are obtained. Each of these models are then tested using the cafeteria noisy test set with the corresponding SNR. The model trained with $\mathcal{T}_{0\text{dB}}$ is tested on $\mathcal{S}_{0\text{dB}}$, the model trained with $\mathcal{T}_{5\text{dB}}$ is tested on $\mathcal{S}_{5\text{dB}}$ and so forth. The test procedure is described in detail in Appendix J.

In Table 10.2 the accuracy results are seen from training on five noisy training sets and testing on five noisy test sets.

**Table 10.2:** Accuracy in percent of the baseline model with different training sets, test sets, and feature extraction methods. $\mathcal{T}_{0\text{dB}} \rightarrow \mathcal{S}_{0\text{dB}}$ means the model was trained on noisy training set with 0dB SNR and tested on the noisy test set with 0 dB SNR.

| | STFT | MFCC | IMFCC | GFCC | IGFCC |
|---|---|---|---|---|---|
| $\mathcal{T}_{0\text{dB}} \rightarrow \mathcal{S}_{0\text{dB}}$ | 92.05 ± 0.81 | 58.12 ± 1.39 | 95.76 ± 0.45 | 57.58 ± 1.02 | 96.35 ± 0.66 |
| $\mathcal{T}_{5\text{dB}} \rightarrow \mathcal{S}_{5\text{dB}}$ | 91.77 ± 1.92 | 67.70 ± 1.13 | 96.69 ± 0.31 | 67.13 ± 1.55 | 97.25 ± 0.38 |
| $\mathcal{T}_{10\text{dB}} \rightarrow \mathcal{S}_{10\text{dB}}$ | 93.85 ± 0.42 | 79.21 ± 1.00 | 97.30 ± 0.43 | 76.07 ± 1.09 | 97.75 ± 0.42 |
| $\mathcal{T}_{15\text{dB}} \rightarrow \mathcal{S}_{15\text{dB}}$ | 93.03 ± 4.57 | 85.25 ± 1.65 | 93.90 ± 0.56 | 84.07 ± 0.78 | 94.66 ± 0.35 |
| $\mathcal{T}_{20\text{dB}} \rightarrow \mathcal{S}_{20\text{dB}}$ | 94.78 ± 0.51 | 90.00 ± 1.00 | 94.19 ± 0.51 | 87.08 ± 0.87 | 93.65 ± 0.46 |

Table 10.2 shows that the IMFCC and IGFCC generally achieve the highest accuracies of approximately 93 % to 98 % with the STFT not far behind. The MFCC and GFCC perform poorly for low SNRs as expected. Their accuracies decreases from approximately 90 % at 20 dB SNR to approximately 58 % at 0 dB SNR. There is a general increase in performance as the SNR decreases except for the IMFCC and IGFCC where the performance decreases when going from 10 dB SNR to 15 dB SNR which is a bit peculiar. It is seen that in the two cases $\mathcal{T}_{15\text{dB}} \rightarrow \mathcal{S}_{15\text{dB}}$ and $\mathcal{T}_{15\text{dB}} \rightarrow \mathcal{S}_{15\text{dB}}$ the IMFCC and IGFCC features perform worse that in the three other SNR cases. This tendency only holds for the IMFCC and IGFCC features which might suggest that their dense focus at higher frequencies are vulnerable to some SNRs in this case. It is seen in Table 3.1 that for most combinations of normal file lenghts and target lenghts the adversarial SNRs are between 20 dB and 30 dB. One suggestion to explain the IMFCC and IGFCC features' peculiar behaviour could be that these feature extraction methods are affected when the amout of added noise is close to the adversarial SNR.

## 10.5 Known Single Noise Type

In section 10.4 five models were trained individually on the five cafeteria noisy training sets $\mathcal{T}_{0\text{dB}}$, $\mathcal{T}_{5\text{dB}}$, $\mathcal{T}_{10\text{dB}}$, $\mathcal{T}_{15\text{dB}}$, and $\mathcal{T}_{20\text{dB}}$. Here these five training sets are combined with $\mathcal{T}_A$ into one cafeteria noisy training set $\mathcal{T}_{\text{caf}}$. The purpose of this test is to evaluate the performance of the baseline model and the different features when it is trained on a wide set of SNRs.

By combining the training sets, six times as much data is used for training. This way all normal and adversarial training files will appear six times in the training set. This is accepted as their individual SNRs will be different. The baseline model is trained with $\mathcal{T}_{\text{caf}}$ and tested individually on the five SNR dependent cafeteria noisy test sets: $\mathcal{S}_{0\text{dB}}$,

$\mathcal{S}_{5\text{dB}}$, $\mathcal{S}_{10\text{dB}}$, $\mathcal{S}_{15\text{dB}}$ and $\mathcal{S}_{20\text{dB}}$.

Due to time constraints the batch size is increased to 512 from 64 in order to better utilize the capability to compute in parallel on the Graphics Processing Unit (GPU). This theoretically reduces the training time by a factor of 8. With an increase in batch size, the learning rate should also be increased [47, 48]. The learning rate is therefore set to 4.0e−4 on the basis of a few short tests. The full test procedure is described in Appendix K.

In Table 10.3 the results are shown from training on $\mathcal{T}_{\text{caf}}$ and testing individually on the SNR dependent cafeteria noisy test sets: $\mathcal{S}_{0\text{dB}}$, $\mathcal{S}_{5\text{dB}}$, $\mathcal{S}_{10\text{dB}}$, $\mathcal{S}_{15\text{dB}}$ and $\mathcal{S}_{20\text{dB}}$.

**Table 10.3:** Accuracy in percent of the baseline model with different training sets, test sets, and feature extraction methods. $\mathcal{T}_{\text{caf}} \rightarrow \mathcal{S}_{0\text{dB}}$ means the model was trained on the combined noisy training set of all five SNRs and tested on the noisy test set with 0 dB SNR.

| | STFT | MFCC | IMFCC | GFCC | IGFCC |
|---|---|---|---|---|---|
| $\mathcal{T}_{\text{caf}} \rightarrow \mathcal{S}_{0\text{dB}}$ | 92.88 ± 1.38 | 61.63 ± 1.88 | 95.90 ± 0.27 | 62.92 ± 1.22 | 95.42 ± 2.12 |
| $\mathcal{T}_{\text{caf}} \rightarrow \mathcal{S}_{5\text{dB}}$ | 94.85 ± 0.69 | 71.88 ± 1.11 | 96.94 ± 0.38 | 71.97 ± 1.72 | 96.66 ± 1.17 |
| $\mathcal{T}_{\text{caf}} \rightarrow \mathcal{S}_{10\text{dB}}$ | 96.11 ± 0.54 | 80.17 ± 1.30 | 98.03 ± 0.21 | 80.34 ± 1.52 | 97.92 ± 0.78 |
| $\mathcal{T}_{\text{caf}} \rightarrow \mathcal{S}_{15\text{dB}}$ | 97.14 ± 0.57 | 88.15 ± 0.97 | 99.04 ± 0.27 | 85.81 ± 0.86 | 98.60 ± 0.59 |
| $\mathcal{T}_{\text{caf}} \rightarrow \mathcal{S}_{20\text{dB}}$ | 96.82 ± 0.58 | 92.64 ± 0.91 | 99.30 ± 0.32 | 91.12 ± 1.13 | 98.96 ± 0.46 |

Table 10.3 shows that the IMFCC and IGFCC achieve the highest accuracies for all SNRs while the MFCC and GFCC both achieve the lowest accuracies. The IMFCC and IGFCC features accuracies are decreasing from approximately 99 % at 20 dB to approximately 95 % at 0 dB. In the same manner the MFCC and GFCC features' accuracies are decreases from approximately 92 % to 62 %, which is a significantly larger drop. The STFT features perform worse than the IMFCC and IGFCC features but still has accuracies above 92.88± 1.38 for all SNRs. The expectation that a decrease in SNR also results in a decrease in performance is confirmed from the results in Table 10.3.

## 10.6 Multiple Noise Types at Training - Known Noise Type at Testing

In this section the robustness of the baseline model and the different features towards noise, is further evaluated by using five other types of noise together with the cafeteria (CAF) noise for training. These five other noise types are the same as described in [44] which are: bus (BUS), street (STR), pedestrian (PED), babble (BBL) and speech shaped noise (SSN). Six different types of noise are then available for evaluating the baseline model. The model is tested individually on the five cafeteria noisy test sets: $\mathcal{S}_{0\text{dB}}$, $\mathcal{S}_{5\text{dB}}$,

$\mathcal{S}_{10\text{dB}}$, $\mathcal{S}_{15\text{dB}}$ and $\mathcal{S}_{20\text{dB}}$. This evaluates how the baseline model performs when it is trained on many types of noise and tested on a known noise type for each feature extraction method.

The training set $\mathcal{T}_{6\text{noise}}$ is a training set where the six noise types BUS, STR, PED, BBL, SSN and CAF are present. The training set $\mathcal{T}_{6\text{noise}}$ is generated in a similar way as in section 10.1 test 4. For each file in the white box dataset a noise type is randomly drawn from a uniform distribution. This is repeated five times, one for each of the five SNRs: 0dB, 5dB, 10dB, 15dB and 20dB. This way five SNR dependent training sets are generated where in each of them all six noise types are present. The training set $\mathcal{T}_{6\text{noise}}$ is then the combination of these five training sets and dataset A which has added no noise. The full test procedure is described in Appendix L.

In Table 10.4 the accuracies for all five features are shown for training on $\mathcal{T}_{6\text{noise}}$ and testing individually on the SNR dependent cafeteria noisy test sets: $\mathcal{S}_{0\text{dB}}$, $\mathcal{S}_{5\text{dB}}$, $\mathcal{S}_{10\text{dB}}$, $\mathcal{S}_{15\text{dB}}$ and $\mathcal{S}_{20\text{dB}}$.

**Table 10.4:** Accuracy in percent of the baseline model with different training sets, test sets, and feature extraction methods. $\mathcal{T}_{6\text{noise}} \rightarrow \mathcal{S}_{0\text{dB}}$ means the model was trained on the combined noisy training set of all six noise types and all five SNRs and tested on the noisy test set with 0 dB SNR.

| | STFT | MFCC | IMFCC | GFCC | IGFCC |
|---|---|---|---|---|---|
| $\mathcal{T}_{6\text{noise}} \rightarrow \mathcal{S}_{0\text{dB}}$ | 91.80 ± 1.36 | 64.55 ± 1.87 | 94.86 ± 0.79 | 64.49 ± 1.55 | 96.35 ± 0.57 |
| $\mathcal{T}_{6\text{noise}} \rightarrow \mathcal{S}_{5\text{dB}}$ | 94.44 ± 0.65 | 74.97 ± 1.65 | 97.44 ± 0.59 | 74.66 ± 1.56 | 97.61 ± 0.50 |
| $\mathcal{T}_{6\text{noise}} \rightarrow \mathcal{S}_{10\text{dB}}$ | 95.76 ± 0.40 | 83.34 ± 1.60 | 98.17 ± 0.30 | 82.67 ± 1.62 | 98.23 ± 0.25 |
| $\mathcal{T}_{6\text{noise}} \rightarrow \mathcal{S}_{15\text{dB}}$ | 96.83 ± 0.44 | 90.90 ± 1.57 | 99.13 ± 0.20 | 88.29 ± 0.59 | 99.16 ± 0.30 |
| $\mathcal{T}_{6\text{noise}} \rightarrow \mathcal{S}_{20\text{dB}}$ | 97.13 ± 0.31 | 93.60 ± 1.12 | 99.33 ± 0.19 | 92.19 ± 0.43 | 99.24 ± 0.27 |

Table 10.4 shows that the IMFCC and IGFCC achieve the highest accuracies for all SNRs while the MFCC and GFCC both achieve the lowest accuracies. The IMFCC and IGFCC features accuracies are decreasing from approximately 99 % at 20 dB to approximately 95 % at 0 dB. The MFCC and GFCC features' accuracies decrease from approximately 93 % at 20 dB SNR to approximately 64 % at 0 dB. The STFT features perform worse than the IMFCC and IGFCC features but still has accuracies above $91.80 \pm 1.36$ for all SNRs. The expectation that a decrease in SNR also results in a decrease in performance is confirmed from the results in Table 10.4.

## 10.7 Multiple Noise Types Training - Unknown Noise Type at Testing

This section aims to evaluate the performance of the model trained to generalize on many types of noise but is tested on a noise type that is previously unseen. The baseline model is trained for each feature extraction method on training set $\mathcal{T}_{5\text{noise}}$ and individually tested on the five SNR dependent cafeteria noisy test sets: $\mathcal{S}_{0\text{dB}}$, $\mathcal{S}_{5\text{dB}}$, $\mathcal{S}_{10\text{dB}}$, $\mathcal{S}_{15\text{dB}}$ and $\mathcal{S}_{20\text{dB}}$. The training set $\mathcal{T}_{5\text{noise}}$ is generated in a similar way as the training set $\mathcal{T}_{6\text{noise}}$ described in section 10.6. The difference is that for $\mathcal{T}_{5\text{noise}}$ the noise type to add is randomly drawn from only five types of noise being: BUS, STR, PED, BBL and SSN. $\mathcal{T}_{5\text{noise}}$ will then contain the same number of files as $\mathcal{T}_{6\text{noise}}$ and $\mathcal{T}_{\text{caf}}$. The full test procedure is described in Appendix M.

In Table 10.5 the accuracies for all five features are shown for training on $\mathcal{T}_{5\text{noise}}$ and testing individually on the SNR dependend cafeteria noisy test sets: $\mathcal{S}_{0\text{dB}}$, $\mathcal{S}_{5\text{dB}}$, $\mathcal{S}_{10\text{dB}}$, $\mathcal{S}_{15\text{dB}}$ and $\mathcal{S}_{20\text{dB}}$.

**Table 10.5:** Accuracy in percent of the baseline model with different training sets, test sets, and feature extraction methods. $\mathcal{T}_{5\text{noise}} \rightarrow \mathcal{S}_{0\text{dB}}$ means the model was trained on the combined noisy training set of all five noise types (CAF excluded) and all five SNRs and tested on the noisy test set with 0 dB SNR.

| | STFT | MFCC | IMFCC | GFCC | IGFCC |
|---|---|---|---|---|---|
| $\mathcal{T}_{5\text{noise}} \rightarrow \mathcal{S}_{0\text{dB}}$ | 89.83 ± 0.93 | 64.69 ± 2.03 | 94.86 ± 0.69 | 63.57 ± 1.40 | 95.90 ± 0.82 |
| $\mathcal{T}_{5\text{noise}} \rightarrow \mathcal{S}_{5\text{dB}}$ | 93.61 ± 1.05 | 74.69 ± 1.34 | 97.42 ± 0.42 | 72.53 ± 2.48 | 97.39 ± 0.37 |
| $\mathcal{T}_{5\text{noise}} \rightarrow \mathcal{S}_{10\text{dB}}$ | 95.22 ± 0.63 | 82.36 ± 1.11 | 97.67 ± 0.25 | 79.72 ± 2.60 | 98.12 ± 0.30 |
| $\mathcal{T}_{5\text{noise}} \rightarrow \mathcal{S}_{15\text{dB}}$ | 96.49 ± 0.63 | 89.86 ± 0.87 | 98.88 ± 0.16 | 88.12 ± 1.35 | 98.76 ± 0.22 |
| $\mathcal{T}_{5\text{noise}} \rightarrow \mathcal{S}_{20\text{dB}}$ | 96.77 ± 0.56 | 93.62 ± 0.87 | 99.10 ± 0.23 | 90.70 ± 1.42 | 98.85 ± 0.28 |

Table 10.5 shows that the IMFCC and IGFCC achieve the highest accuracies for all SNRs while the MFCC and GFCC both achieve the lowest accuracies. The IMFCC and IGFCC features accuracies are decreasing from approximately 99 % at 20 dB to approximately 95 % at 0 dB. The MFCC and GFCC features' accuracies decrease from approximately 92 % at 20 dB SNR to approximately 64 % at 0 dB. The STFT features reaches accuracies between approximately 90 %-97 % for all SNRs. The expectation that a decrease in SNR also results in a decrease in performance is confirmed from the results in Table 10.5.

## 10.8 Discussion of Noise Robustness

A series of tests were made in order to investigate the model's and the different features' robustness to noise. The training sets and test sets were chosen in a way that gives insight

into how the baseline model and the different features are affected in various cases with various types and amounts of noise present.

The results in Table 10.1 for training on $\mathcal{T}_A$ and testing on $\mathcal{S}_{0\text{dB}}$, $\mathcal{S}_{5\text{dB}}$, and so on, clearly shows the problem that can occur when the amount of noise is vastly different between training and testing. And while the case $\mathcal{T}_A \rightarrow \mathcal{S}_A$ achieve near perfect accuracies, the accuracies in the case $\mathcal{T}_A \rightarrow \mathcal{S}_{0\text{dB}}$ for the MFCC and GFCC features are reduced to near random guessing. It is interesting to note that the STFT outperforms the other feature extraction methods. Even though the STFT features are the best performing in test 1, it still only reaches an accuracy of $78.57 \pm 2.58$ for testing on $\mathcal{S}_{0\text{dB}}$. Test 1 is clearly the case where the baseline model and the different features have had their accuracies decreased the most, which is not unexpected. Test 1 is one of two test where an unknown noise type is present in the test set.

Test 5 is the other test where an unknown noise type is present in the test set, and here the results are more promising than in test 1. In Table 10.5 it is seen that for the IMFCC and IGFCC features it is possible to detect adversarial examples with approximately 95%-99% accuracy. These results are obtained by training the baseline model on various types of noise at various SNRs. It is unlikely that the high accuracies are solely a result of the model being trained on many different types and amounts of noise as the resulting dataset also had an increase in the amount of data by a factor of six compared to test 1. This is supported by the results in Table 10.2 and Table 10.3 where the expectation is that training on a single noise type at a single volume would outperform a model that is trained on the same noise type at multiple volumes. It is surprising that this is not the case but not inexplicable. The increase in data and the resulting changes to the batch size and learning rate also changes the way the model learns and how much it is possible to learn. This is likely the reason for the peculiar relation between the two.

For test 4 it was expected that the accuracies should in general descrease compared to the results from test 3. By comparing Table 10.4 and Table 10.3 it is seen that this is not the case. It holds for testing on $\mathcal{S}_{0\text{dB}}$ but the accuracies in the two tests become more similar as the test set SNR increases except for the MFCC and GFCC. These consistently perform better in test 4 than in test 3 which is unexpected. For the other features the accuracies' confidence intervals overlap which indicates that the models in the two tests perform equally. Two things could explain this unexpected behaviour. The fact that approximately $1778 \cdot 0.7 \simeq 1244$ cafeteria noisy files are still present in $\mathcal{T}_{6\text{noise}}$ could be the reason that the model in test 4 is able to reach equal accuracies as the model in test 3. Another thing could be that some of the other five noise types are similar to the cafeteria noise, e.g. PED and STR noise. However, this does not seem like a sufficient explanation for the behaviour of the MFCC and GFCC.

For test 5 it was expected that the accuracies should in general decrease compared to the results from test 4. By comparing Table 10.5 and Table 10.4 it is seen that this is the case but the difference is not as significant as expected. For example it is seen that the

IMFCC for testing on $\mathcal{S}_{20\text{dB}}$ has an accuracy of $99.33 \pm 0.19$ in test 4, but for the same test set in test 5 the IMFCC has an accuracy of $99.10 \pm 0.23$. For testing on $\mathcal{S}_{0\text{dB}}$ the IMFCC in test 4 has an accuracy of $94.86 \pm 0.79$ and in test 5 the IMFCC has an accuracy of $94.86 \pm 0.69$. These results suggests that by training with the types and amounts of noise present in $\mathcal{T}_{5\text{types}}$, the baseline model is able to generalize to unseen types of noise.

In general the IMFCC and IGFCC feature extraction methods outperform the others in all these tests as was expected when inspecting the spectrograms. The STFT is also not far behind those two in all these tests. The spectrogram showed that the addition of cafeteria noise had most of its frequency components focused in the lower frequencies. Thus the feature extraction methods IMFCC and IGFCC should also be able to better detect the higher frequency adversarial noise components because of their better resolution in the high frequencies.

When inspecting the plots of the false positive rate over the false negative rate in the appendices, Appendix I, J, K, L, and M, there appears no significant trade-off between the feature extraction methods in terms of one producing fewer false positives and another fewer false negatives. This means that the feature extraction method with the highest accuracies perform best no matter if FNs or FPs are weighted as costlier.

# Part III

# Discussion and Conclusion

# 11 | Discussion

Through the project several choices have been made both regarding what to investigate and how to handle data. It was chosen to focus on feature extraction methods, influence of speech and nonspeech, and influence of noise. In Chapter 8 a baseline neural network model was implemented with an architecture as described in [12]. Here it was chosen to investigate how it affected the performance when using each of the five feature extraction methods STFT, MFCC, IMFCC, GFCC, IGFCC. These five methods are just a few of many feature extraction methods which are applicable in the audio domain. It could have been chosen to investigate more feature extraction methods but this would also increase the time required for training models. The training of models has been one of the main concerns through the project as even small errors in the dataset, model architecture, hyper parameters etc. could require days of re-training models. The STFT feature requires significantly more time for training models as this feature contains more data. It takes approximately 16 hours to train the baseline model with the white box dataset using the STFT features. This is a result of the number of epochs and batch size being set as in Table 8.1, and the fact that 10 models are trained to calculate the 95 percent confidence interval. This is also the reason that the batch size and learning rate is increased for training with $\mathcal{T}_{\text{caf}}$, $\mathcal{T}_{\text{6noise}}$ and $\mathcal{T}_{\text{5noise}}$ as described in section 10.5.

How to handle data has also been a main concern through the project. It was chosen to extract the features separately to training the model as described in Chapter 7. It was also chosen that the input to the feature extraction should always be .wav files. This means that when splitting the data into speech and nonspeech, the input is a .wav file containing both speech and nonspeech and the output is two .wav files, one containing speech segments and one containing nonspeech segments. It also means that when adding noise the input is one .wav file from the white box dataset and the output is one .wav file containing the white box file with added noise. It was chosen to handle the data this way as it made it possible to examine the data to ensure it had been modified as desired. The possibility of verifying data is also the reason that the features are extracted separate from training the model. It would be possible to modify data and extract features as a part of the training of a model. This way errors could be avoided in the way that only one dataset should be handled outside training. The disadvantage of handling data this way is that it is more difficult to investigate and verify the datasets with added noise and the speech/nonspeech datasets.

In Chapter 9 the influence of the speech and nonspeech segments was investigated. The investigation was made based on the white box data set but it was also attempted to investigate using the the black box dataset. This was not possible as the length of the examples in the black box dataset did not fit with the parameters for the feature extraction methods and the architecture of the baseline model. This issue could have been solved by using zero padding on the black box dataset but then the issue described in Chapter 7 might occur. The problem with too small files are handled by simply discarding these

files. This is accepted as long as the number of discarded files is not to large, but it could have been handled in different ways. One concern when testing the influence of speech and nonspeech segments is that the amounts of speech and nonspeech is not equal. After splitting the white box dataset, the speech segments consist of 157.7 MB data and the nonspeech segments consist of 67.9 MB data. This suggest that another dataset might be more suited for evaluating the influence of speech and nonspeech segments as the white box dataset produces an unbalanced set in this case.

In Chapter 10 it was investigated how the presence of noise affects the performance of the baseline model and the respective feature extraction methods. It was chosen to only investigate this using the white box dataset and not the black box dataset. The reason of this is mainly the time constraint of the project. It was chosen to focus on investigating noise in the white box dataset rather than investigating noise in both the white box and black box dataset. There are two reasons that the white box dataset is chosen over the black box dataset: The white box dataset contains more data and varies more in length of files, lengths of targets, and in the amount of speech in each file.

One important thing to notice is that through the project the architecture of the model used for training was not changed. It is the baseline model with an architecture as descriped in [12] which is used in all tests. The different results and the improvements in accuracy has been achieved solely by changing the data. The only change besides in the data has been in the hyperparameters when training on $\mathcal{T}_{\text{caf}}$, $\mathcal{T}_{\text{6noise}}$ and $\mathcal{T}_{\text{5noise}}$ as described in section 10.5. This proves the importance of having representative data for training a model to be robust in a wide range of use-cases. This leaves room for future work in investigating how the model architecture affects detection of adversarial examples for speech recognition.

In Chapter 8, Chapter 9 and Chapter 10 it is found that the IMFCC and IGFCC feature extraction methods in general outperforms the others. For future work it could then be decided only to use the IMFCC and IGFCC feature extraction methods. The advantage of doing this is that the time required to train models will be significantly reduced especially by leaving out the STFT. The disadvantage would be that it could be missed if e.g. the MFCC performs best for other use-cases than the ones described in Chapter 8, Chapter 9 and Chapter 10.

There are several ways to further investigate the baseline model's and the different feature extraction methods' ability to detect adversarial examples for speech recognition. It would be interesting to repeat the test in section 10.7 and choosing a different type of noise as the unknown noise type in the test set. By doing this it would be possible to conclude more certainly if the baseline model and the respective feature extraction methods generalizes well when its training set contains several types and amounts of noise. It could also be interesting to look into if the baseline model and the feature extraction methods are robust to adaptive adversarial attacks. By adaptive adversarial examples is meant that the baseline model is seen as a part of the victim model when the adversarial examples

are generated. If it is possible to make adaptive adversarial attacks on the baseline model then the baseline model suffers from the same weaknesses as e.g. Deepspeech. However if the model and the feature extraction methods are robust against adaptive adversarial attacks then they would prove as a serious suggestion to detection of adversarial examples for speech recognition.

# 12 | Conclusion

This project has investigated the detection of adversarial examples for speech recognition with a focus on using NNs and five different feature extraction methods, STFT, MFCC, IMFCC, GFCC and IGFCC. In addition to this it was investigated how the detection of adversarial examples is influenced by speech and nonspeech in the examples. Furthermore the model's and the feature extraction robustness to noise was investigated. The project expands upon the work in [12], using the author's generated white box and black box datasets together with their model architecture.

The main concepts of ANNs have been described along with some of the possible challenges associated with ANNs. The neural network type CNN has been investigated and some of its adjustable parameters have been presented. The concept of adversarial examples has been described and some examples have been presented of how they pose a threat to state-of-the-art neural networks. Some methods for generating adversarial examples in the audio domain have been presented. The dataset used in the project have been investigated in order to have a clear understanding of how data can differ and which properties hold for the different types of data in the dataset. The five feature extraction methods STFT, MFCC, IMFCC, GFCC and IGFCC have been investigated with the purpose of being able to test different types of features as input for a neural network. Finally, different evaluation metrics have been presented. These are studied to be able to evaluate a neural network correctly and not be mislead by too simple evaluation metrics.

Part I gave insight into formulating a problem statement for the project.

**How do different feature extraction methods affect the performance of a CNN when detecting adversarial examples for speech recognition, how robust are they to noise, and is it the characteristic from speech or nonspeech that is most important when classifying?**

This problem statement was answered throughout the project and report through test, evaluations and analyzing.

In part II a baseline model was implemented and used throughout the project. The model's architecture is the same as found in [12]. To handle the various example lengths, the examples are split into blocks instead of using zero padding. The model was trained individually on the white box dataset A, black box dataset B, and the datasets combined AB, and then evaluated individually on Dataset A and B. The baseline model was evaluated per block and per file. Here it was found that a higher accuracy could be achieved when evaluating per file. It was also found that IMFCC and IGFCC performed best in all the aforementioned training and test scenarios.

Dataset A was split up into speech and nonspeech segments. A test was performed where

the model was trained on dataset A and evaluated individually on the speech segment and nonspeech segment. Here it was found that the model achieves the highest accuracy when evaluated on nonspeech, and the feature methods IMFCC and IGFCC performed best by a significant margin compared to the others. A second test was performed, training on speech and nonspeech individually and evaluating on speech and nonspeech individually. Here it was found that the characteristics for recognizing an adversarial example gained from speech is more important than from nonspeech. However, it was not a substitute for training on dataset A which contains both. Furthermore it was seen that IMFCC and IGFCC performed best in all of the aforementioned training and test scenarios.

Five tests were performed to test the model's robustness to noise. The tests had various number of noise types and various amounts of noise added to either or both of the training set and test set. When testing the model trained on dataset A without any noise added it was found that STFT was the method with highest accuracy when noise is present in the test set. In all other tests the IMFCC and IGFCC methods achieved the highest accuracies. In test 1 it was also found that when training on dataset A and testing on dataset A with noise all feature methods performed poorly. However by training on dataset A with noise and testing on dataset A with noise the accuracy improved remarkably. This is both true when the noise type used for testing is known and when it is unknown. Test 4 and 5 trained a model on many different types of noise and tested it on a known and unknown noise type respectively. There is no remarkably difference in accuracy between test 4 and 5, meaning that even without knowing the noise type the model is still able to classify adversarial examples correctly with a high accuracy. For test 5 the IMFCC and IGFCC achieved a lowest accuracy of $94.86 \pm 0.69$ and a highest of $99.10 \pm 0.23$ which are remarkably well performances for a system detecting an unknown noise type.

Part II has thereby examined how different feature extraction methods affect the performance of a CNN when detection adversarial for speech recognition. It has examined the importance of speech and nonspeech when classifying and how robust the CNN and different feature extracting methods are to different amount and types of noise.

# Bibliography

[1] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. 2014.

[2] Xiaoyong Yuan, Pan He, Qile Zhu, Rajendra Rana Bhat, and Xiaolin Li. Adversarial examples: Attacks and defenses for deep learning. *IEEE transactions on neural networks and learning systems*, 2019.

[3] Kevin Eykholt Ivan Evtimov Earlence Fernandes Bo Li Amir Rahmati Chaowei Xiao Atul Prakash Tadayoshi Kohno and Dawn Song. Robust physical-world attacks on deep learning visual classification. *CVPR*, abs/1707.08945, 2018.

[4] Mahmood Sharif, Sruti Bhagavatula, Lujo Bauer, and Michael K. Reiter. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. pages 1528–1540, 2016. doi: 10.1145/2976749.2978392.

[5] N. Carlini and D. Wagner. Audio adversarial examples: Targeted attacks on speech-to-text. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 1–7, May 2018. doi: 10.1109/SPW.2018.00009.

[6] google. Google assistant, what it can do, 2019. URL https://assistant.google.com/explore/c/13/?hl=en_us.

[7] Amazon. Alexa user guide: Learn what alexa can do, 2019. URL https://www.amazon.com/b/ref=s9_acss_bw_cg_aegshop_md1_w?node=17934671011.

[8] Milen Yanachkov. Google assistant now lets you make in-app purchases using only your voice, 2018. URL https://www.phonearena.com/news/google-assistant-in-app-purchases-digital-subscriptions-voice_id109584.

[9] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deepspeech: Scaling up end-to-end speech recognition. 12 2014.

[10] Krishan Rajaratnam and Jugal Kalita. Noise flooding for detecting audio adversarial examples against automatic speech recognition. *CoRR*, abs/1812.10061, 2018.

[11] Zhuolin Yang, Bo Li, Pin-Yu Chen, and Dawn Song. Characterizing audio adversarial examples using temporal dependency. *CoRR*, abs/1809.10875, 2018.

[12] S. Samizade, ZH. Tan, C. Shen, and X. Guan. Adversarial attack detection for deep speech recognition. 2019.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[14] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *CoRR*, abs/1811.03378, 2018.

[15] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359 – 366, 1989. ISSN 0893-6080.

[16] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951. ISSN 00034851.

[17] Boris Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr Computational Mathematics and Mathematical Physics*, 4:1–17, 12 1964. doi: 10.1016/0041-5553(64)90137-5.

[18] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 2014.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015. doi: 10.1109/iccv.2015.123.

[20] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[21] Ossama Abdel-Hamid, Abdel rahman Mohamed, Hui Jiang, Li Deng, Gerald Penn, and Dong Yu. Convolutional neural networks for speech recognition. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 22, October 2014. ISSN 2329-9304. doi: 10.1109/TASLP.2014.2339736.

[22] Samuel Thomas, Sriram Ganapathy, George Saon, and Hagen Soltau. Analyzing convolutional neural networks for speech activity detection in mismatched acoustic conditions. *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2014. ISSN 2379-190X. doi: 10.1109/ICASSP.2014.6854054.

[23] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks, 2015.

[24] Sumit Saha. A comprehensive guide to convolutional neural networks, 2018. URL https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

[25] Ujjwalkarn. An intuitive explanation of convolutional neural networks, 2016. URL https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/.

[26] Dang Ha The Hien. A guide to receptive field arithmetic for convolutional neural networks, 2017. URL https://medium.com/mlreview/a-guide-to-receptive-field-arithmetic-for-convolutional-neural-networks-e0f514068807.

[27] Denny Britz. Understanding convolutional neural networks for nlp, 2015. URL http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/.

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun 2016. doi: 10.1109/cvpr.2016.90.

[30] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *CoRR*, abs/1412.6572, 2014.

[31] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun 2015. doi: 10.1109/cvpr.2015.7298594.

[32] Nicholas Carlini, Pratyush Mishra, Tavish Vaidya, Yuankai Zhang, Micah Sherr, Clay Shields, David A. Wagner, and Wenchao Zhou. Hidden voice commands. In *USENIX Security Symposium*, 2016.

[33] Moustafa Alzantot, Bharathan Balaji, and Mani B. Srivastava. Did you hear that? adversarial examples against automatic speech recognition. *CoRR*, abs/1801.00554, 2018.

[34] Xuedong Huang, Alex Acero, and Hsiao-Wuen Hon. *Spoken Language Processing: A Guide to Theory, Algorithm, and System Development*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001. ISBN 0130226165.

[35] A. Adiga, M. Magimai, and C. S. Seelamantula. Gammatone wavelet cepstral coefficients for robust speech recognition. In *2013 IEEE International Conference of IEEE Region 10 (TENCON 2013)*, pages 1–4, Oct 2013. doi: 10.1109/TENCON.2013.6718948.

[36] P. P. Vaidyanathan. *Multirate Systems and Filter Banks*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993. ISBN 0-13-605718-7.

[37] B. Ayoub, K. Jamal, and Z. Arsalane. Gammatone frequency cepstral coefficients for speaker identification over voip networks. In *2016 International Conference on Information Technology for Organizations Development (IT4OD)*, pages 1–5, March 2016. doi: 10.1109/IT4OD.2016.7479293.

[38] Alan. Why accuracy alone is a bad measure for classification tasks, and what we can do about it, 2013. URL https://tryolabs.com/blog/2013/03/25/why-accuracy-alone-bad-measure-classification-tasks-and-what-we-can-do-about-it/.

[39] S. M. Ross. *Introduction to Probability and Statistics for Engineers and Scientists.* Elsevier Academic Press, 200 Wheeler Road, 6 th Floor, Burlington, MA 01803, USA, 2004. ISBN 0-12-598057-4.

[40] Tara N. Sainath and Carolina Parada. Convolutional neural networks for small-footprint keyword spotting. In *INTERSPEECH*, 2015.

[41] Md Sahidullah, Tomi Kinnunen, and Cemal Hanilçi. A comparison of features for synthetic speech detection. 2015.

[42] Malcolm Slaney. An efficient implementation of the patterson-holdsworth auditory filter bank. 2000.

[43] Z. Tan and B. Lindberg. Low-complexity variable frame rate analysis for speech recognition and voice activity detection. *IEEE Journal of Selected Topics in Signal Processing*, 4(5):798–807, Oct 2010. ISSN 1932-4553. doi: 10.1109/JSTSP.2010.2057192.

[44] M. Kolbœk, Z. 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)*, pages 305–311, Dec 2016. doi: 10.1109/SLT.2016.7846281.

[45] Jon Barker, Ricard Marxer, Emmanuel Vincent, and Shinji Watanabe. The third 'chime' speech separation and recognition challenge: Dataset, task and baselines. *2015 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*, pages 504–511, 2015.

[46] M. Kolbæk, Z. 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*, 25(1):153–167, Jan 2017. ISSN 2329-9290. doi: 10.1109/TASLP.2016.2628641.

[47] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014.

[48] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.

# Part IV

# Appendices

# A | Splitting into blocks and extracting features

In this project it is decided to split the input examples into blocks of equal size. This is done to solve the issue that NNs in general do not take input of different sizes. Once the input examples are split up into blocks features are extracted from each of these blocks. In order to explain the process some knowledge is needed of how the dataset is structured.

## A.1 Structure of the dataset

The complete dataset consists of a white box attack data set and a black box attack dataset.

The white box dataset has a flat structure where all files located in the same directory called "W_Box". The normal examples are named "sample-<sample number>.wav" where <sample number> could be 000003. The adversarial examples are named "adv-<attack type>-<sample number>-wav" where <attack type> for example could be "short2medium", "medium2short", "long2short" etc. In the white box directory 2100 wav files are located, 900 adverarial examples and 1200 normal examples. The normal examples consists of 300 examples which are used to generate the adversarial examples, and 900 examples which have no corresponding adversarial example. It is desired to only use the 900 adversarial examples and the 900 normal examples which do not have corresponding adversarial examples.

The black box data set has a more complex structure. The root directory is called "B_Box" which has the subdirectories "Adversarial-Examples" and "Normal-Examples". The directory "Normal-Examples" has the ten subdirectories "down", "go", "left", "no", "off", "on", "right", "stop", "up" and "yes" each containing 180 normal examples which do not have corresponding adversarial examples. The "Adversarial-Examples" directory has the subdirectories "Adversarial-Examples" and "Original-Examples". The structure of these two subdirectories correspond to the explanation of the black box attacks in section 3.2. An import thing to notice is that the individual examples are for example named "1c6e5447_nohash1.wav", which do not indicate if the sample is adversarial or not. It is desired to use all the examples of the black box directory except for the examples in the "Original-Examples" directory as these examples have corresponding adversarial examples.

At this point the dataset consists of wav files where the label of an example is found in either the file name or in the directory. In order to make the input for the NN similar for both white box and black box data it is decided to store the dataset as npz files. These npz files will contain the information; data, labels and file names. In Listing A.1 a code

snippet is seen of how the file names and labels are found from the wav files.

**Listing A.1:** Code snippet of the get_file_names function in feature_extraction.py script.

```
def get_file_names(root_dir):
    file_names = []
    labels = []
    for root, dirs, files in os.walk(root_dir):
        for name in files:
            name = os.path.join(root, name)
            if name.endswith(".wav"):
                file_names.append(name)
                if "adv-" in name or "Adversarial-Examples\Adversarial-Examples" in name:
                    name_label = 1
                    labels.append(name_label)
                else:
                    name_label = 0
                    labels.append(name_label)


    return file_names, labels
```

In Listing A.1 finds the file names by walking through the root directory and storing the names of all wav files. The labels of each sample are then found by checking if either "adv-" or "Adversarial-Examples\Adversarial-Examples" is a part of the examples name. If it is an adversarial example from the white box dataset "adv-" will be present in the file name and if it is an adversarial example from the black box dataset "Adversarial-Examples\Adversarial-Examples" will be present in the file name.

## A.2 Splitting input samples into blocks

Ones the file names and labels are found the next step is to divide the examples into block and then extract features of these blocks. In Listing A.2 a code snippet is seen of how this division into blocks is handled.

**Listing A.2:** Code snippet of the speech_processing function in feature_extraction.py script.

```
def speech_processing(function, path, output, block_size_ms, step_size_ms, n_mfcc):
    """
    Some of initial parts of the function is left of for simplicity

    """
    for i in range(length_dir):
            file = files_dir[i]
            file_label = labels[i]
            print("file number " +str(i+1) + " of " +str(length_dir) + " " + file)
            sr, data = scipy.io.wavfile.read(file) #load file
            data_len = len(data)
            too_short = 0
            block_sizes = int(block_size_ms*sr/1000) # block size in samples

            if block_size >= data_len:
                print("full size is reached for window")
```

```
17                too_short = 1

19            step_size = int(step_size_ms*sr/1000)

21            if step_size >= data_len:
22                print("full size is reached for step")
23                too_short = 1

25            #Find number of blocks in the sample
26            blocks = math.floor((data_len-block_size)/step_size)
27            if blocks == 0:
28                blocks = 1
29            data = data/2.0**15 # normalize
30            data = data.astype(float)
31            data_split = [None]*(blocks+1) # +1 to get the last part
32            mfcc_data_split = [None]*(blocks+1) # +1 to get the last part
33            #split data to equal sizes
34            for n in range(blocks):
35                data_split[n] = data[n*step_size:n*step_size+block_size] # split data into
                      blocks of block size
36                mfcc_data_split[n] = processing_func(function, data_split[n], eng, sr,
                      mfcc_window_length, n_fft, n_mfcc, filterbank) #extract features
37                if too_short == 0: # if sample is larger than block size
38                    labels_split.append(file_label)
39                    file_names_split.append(file+"block"+str(n))

41            data_split[blocks] = data[data_len-block_size:data_len] # The last part of the
                  file
42            data_split_mat = matlab.double(data_split[blocks].tolist()) # make data suited for
                   matlab
43            mfcc_data_split[blocks] = processing_func(function, data_split[blocks], eng, sr,
                  mfcc_window_length, n_fft, n_mfcc, filterbank)
44            if too_short == 0:
45                labels_split.append(file_label)
46                file_names_split.append(file+"block"+str(blocks))
47                blocks_total = blocks_total + blocks+1 #+1 to get the last part
48                mfcc_data_list.append(mfcc_data_split) # gather all files in one list
49            elif too_short == 1:
50                labels_split.append(file_label)
51                file_names_split.append('remove.wavblock0') # for handling too short files
52                mfcc_data_list.append([np.zeros_like(mfcc_data_list[0][0])])
53                blocks_total = blocks_total + 1
```

In Listing A.2 line 7 to 10 loads the example. Line 12 to 28 calculates the number of blocks in the example and handles if the example is too short i.e. the size of the example is smaller than the block size. Two cases are possible regarding the size of the example. One case is that the size of the example is equal to $n \cdot$block_size where $n$ is an integer, and the other case is where this does not hold. The second case can be handled by either discarding the last part of the example or by overrepresenting the last part of the example. It is chosen to overrepresent the last part of the example by finding the end of the example and go one block size back to make the last block. This way of dividing into blocks is handled in line 31 to 48. The variable too_short is used to handle if the example is too

short. If that is the case the example is simply discarded from the dataset. This might unbalance the dataset but as long as the number of discarded examples is not too high, this is accepted. When the example is divided into block features are extracted from each block. This is done in line 36 and 43 in Listing A.2.

## A.3 Extracting features

For extraction the features MFCC, IMFCC, GFCC and IGFCC some MATLAB scripts were provided. In order to run these MATLAB scripts the MATLAB Engine API for Python is used. In Listing A.3 a code snippet of the feature extractions is seen.

**Listing A.3:** Code snippet of the processing_func function in feature_extraction.py script.

```
1  def processing_func(function, data, eng, sr, mfcc_window_length, n_fft, n_mfcc, filterbank): #
       chose what kind of processing to do
2      data_split_mat = matlab.double(data.tolist()) # make data suited for matlab
3      if function == 'mfcc':
4          return np.asarray(eng.extract_mfcc_edited(data_split_mat, float(sr),float(
               mfcc_window_length), float(n_fft), float(n_mfcc))).T #(speech,Fs,Window_Length,
               NFFT,No_Filter)

6      elif function == 'imfcc':
7          return np.asarray(eng.extract_imfcc_edited(data_split_mat, float(sr),float(
               mfcc_window_length), float(n_fft), float(n_mfcc))).T #(speech,Fs,Window_Length,
               NFFT,No_Filter)

9      elif function == 'gfcc':
10         return np.asarray(eng.extract_fbcc_edited(data_split_mat, float(sr),float(
               mfcc_window_length), float(n_fft), float(n_mfcc), filterbank)).T

12     elif function == 'igfcc':
13         return np.asarray(eng.extract_fbcc_edited(data_split_mat, float(sr),float(
               mfcc_window_length), float(n_fft), float(n_mfcc), filterbank)).T

15     elif function == 'stft':
16         return np.abs(Get_STFT_from_wav(sr, data, mfcc_window_length, 16))
```

Line 2 in Listing A.2 makes the data suited as input for the MATLAB engine. The structure is very similar in the first four if statements of Listing A.2. Line 4, 7, 10 and 13 runs a MATLAB script such as extract_mfcc_edited.m. In line 10 and 13 the MATLAB script takes filterbank as input. This filterbank is also made in a MATLAB script. How the filterbanks are extracted is seen in Listing A.4

**Listing A.4:** Code snippet of how filterbanks are extracted in the feature_extraction.py script.

```
1      if function == 'gfcc':
2          filterbank = np.matrix(eng.fft2gammatonemx(float(n_fft), float(16000), float(20),
               float(1), float(50), float(8000), float(n_fft/2+1))) #
3          filterbank = filterbank.getH()
4          filterbank = matlab.double(filterbank.tolist())

6      elif function == 'igfcc':
```

```
7         filterbank = np.matrix(eng.fft2gammatonemx(float(n_fft), float(16000), float(20),
              float(1), float(50), float(8000), float(n_fft/2+1))) #
8         filterbank = filterbank.getH()
9         filterbank = filterbank[::-1] # reverse the filterbank
10        filterbank = matlab.double(filterbank.tolist())
```

In line 16 Listing A.3 the STFT features are extracted using the librosa.core.stft function for Python. The function Get_STFT_from_wav is seen in Listing A.5.

**Listing A.5:** Code snippet of how the stft features are extracted in the datasaver.py script.

```
1  def Get_STFT_from_wav(sr, wav, window_ms=25, shift_ms=10):
2      n_fft = int(sr*window_ms*0.001) # Number of samples to use for fft
3      n_shift = int(sr*shift_ms*0.001)    # Number of samples to jump for next fft frame
4      S = librosa.core.stft(wav, n_fft=n_fft, hop_length=n_shift)
5      return S
```

Once the features are extracted from the blocks they are stored as a numpy array of numpy arrays and saved in an npz file along with their corresponding file name and label. It is desired to know the order of the blocks of each sample. Therefore the final filenames of the feature extracted blocks will be for example "adv-long2long-000003.wavblock0", "adv-long2long-000003.wavblock1" etc. This is seen in Listing A.2 line 39 and 46.

# B | Add noise

The purpose of this appendix is to clarify how noise is added to the white box dataset. The noise is added based on the methods described in [44] and [46]. In Listing B.1 a code snippet is seen of the function add_noise() in the Python file add_noise.py

**Listing B.1:** Code snippet of how noise is added to the white box dataset.

```
1  def add_noise(rvad, noise, file_name): # takes rvad indices, the noise file, and the signal
       file name
2      sr, data = scipy.io.wavfile.read(file_name) # load signal
3      max_sample = np.max(data)
4      data = data/max_sample # normalize signal
5      data_speech = get_speech(rvad,data) # get speech part of signal
6      start_index, index = get_noise_index(len(noise),len(data)) # get segment of noise with
           same lenght as the signal
7      data_rms = get_rms(data)
8      speech_rms = get_rms(data_speech) # calculate rms of signal speech
9      noise_rms = get_rms(noise[index]) # calculate rms of noise
10     ratio = speech_rms/noise_rms
11     noise = noise*ratio # noise gets same rms as signal speech

13     SNRrange = np.arange(0,5)*5 # 0dB, 5dB, 10dB, 15dB and 20dB
14     for snr in SNRrange:
15         noise = noise/(10**(snr/20)) # scale noise to SNR
16         data_out = data + noise[index] # add noise
17         data_out = data_out/np.max(data_out) # normalize
18         data_out = data_out*max_sample # rescale

20         file = remove_dir_in_name(file_name)
21         out_name = file[:-4] +"_" + str(snr)+"dB.wav" # make file name

23         out_dir = "data/caf/noise_W_Box_train{}/".format(snr) # where to store file
24         os.makedirs(out_dir, exist_ok=True)
25         scipy.io.wavfile.write(out_dir+out_name, sr, data_out.astype(np.int16)) # store file
26     return np.asarray(data_out)
```

In Listing B.1 line 5 finds the parts of the signal where speech is present, line 6 returns a random part of the noise file, and line 7 to 18 adds noise to the signal. In line 7 to 11 the noise is manipulated to have the same RMS value as the speech part of the signal. In line 15 the noise is attenuated by the chosen SNR, and in line 16 the noise is added to the signal. The function get_speech() is similar to the function SnS_split() in Listing 9.1. A code snippet of the function get_noise_index() in Listing B.1 is seen in Listing B.2.

**Listing B.2:** Code snippet of how random noise parts is extracted from the noise sample.

```
1  def get_noise_index(noise_len, clean_len):
2      index = np.zeros((clean_len,))
3      start_index = np.random.randint(0, noise_len) # find random start index
4      if noise_len-start_index > clean_len: # coherent noise block is possible
5          index[0:clean_len] = np.arange(start_index,start_index+clean_len, dtype=np.int32)
6      else: # circular buffering is needed
```

```
7         index[0:noise_len-start_index] = np.arange(start_index, noise_len, dtype=np.int32) #
              from start index to end of noise sample
8         start_rest = noise_len-start_index
9         end_rest = clean_len-abs(start_index-noise_len)
10        index[start_rest:start_rest+end_rest] = np.arange(0,end_rest, dtype=np.int32) # fill
              rest of noise block with start of noise sample

12    index_out = [int(x) for x in index] # cast to integer
13    return start_index, index_out
```

In Listing B.2 line 3 randomly draws the start index of the noise block. The if statement in line 4 checks if the lenght from the start index to the end of the noise file is larger than the length of the signal. If this is the case the noise block can simply be taken from the start index and then one signal lengths forward. If this is not the case a circular buffer strategy is used, so the first part of the noise block is from start index to the end of the noise, and the rest will then be taken from the beginning of the noise file. This is done in line 7 to 10.

# C | Automatic Training and Testing with Configuration Files

When training many different networks with multiple initializations, types of feature extraction, and testing on multiple datasets, it is important to have a streamlined process. It is desired to have a versatile set of Python scripts that can handle changes in both network architecture, hyperparameters, feature extraction methods, or other changes of interest for researching their effect. It is desired to limit the amount of coding to implement new changes and avoid clutter with different scripts for each change.

There are different ways of producing versatile code but the focus is not to produce the best implementation of a versatile training and testing environment but rather limit the amount of time the authors of this report has to spend to implement new ideas. Given that the result of this report is not the environment for training and testing different configurations of data, feature extraction methods, and/or neural networks, there will not be made a well-informed choice for the best way to implement this versatile environment but simply the first method that allowed for enough versatility for the above use-case.

Three different scripts are made that should be able to run independently; each with their own objective. These are: `train.py` that trains all configurations of neural networks for this project with minimal coding changes between configurations, `test.py` that tests the trained networks on different test sets and saves all results of interest, and `plot.py` which should show the results in easily digestible formats. The testing and plotting is split to allow for style changes of graphical representations without rerunning the trained networks on the test sets. This keeps the script to produce plots lightweight and fast such that style changes can be implemented fast and easily.

Two methods are used to keep the scripts versatile: An argument parser `argparse` and a configuration file parser `ConfigParser`. The argument parser allow for command line arguments to be passed to the script. Thus making it possible for the programmer to change high-level variables without changing the script. The configuration parser is similar in that it allows for values from outside the script to be attributed to variables in the code. The difference is that the configuration parser reads from a configuration file and prevent that the programmer must write an exceedingly long command to run the script. Furthermore the configuration file will then contain the exact settings that were used to run the script. This allows for easy access in the future and is a crucial way to keep the three scripts independent if certain high-level variables require the same initialization across scripts. The configuration files will be used in all three scripts such to ensure continuity and the setup is shown on Figure C.1.

The configuration files will be used to change hyperparameters, the neural network architecture to use, the dataset, the feature extraction method, and how often to compute

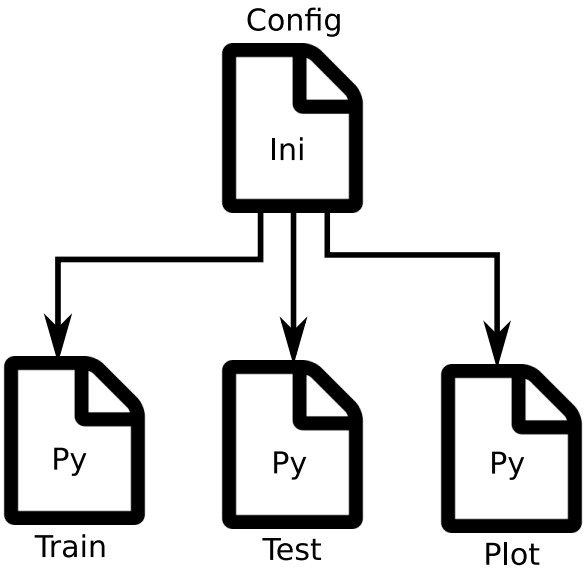



**Figure C.1:** Configuration files and script setup.

performance on the validation set during training. The argument parser will give the user the ability to choose which configuration file to use along with some script specific arguments.

## C.1 Argument Parser Implementation

The argument parser is implemented as shown in Listing C.1.

**Listing C.1:** Code snippet of the argument parser for the three scripts.

```
### Argument parser in train.py
import argparse
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Train neural network to detect adversarial
        examples.')
    parser.add_argument('-cfg',
        type=str,
        default='base_stft.ini',
        help='Configuration filename from "configs/" folder (default=base_stft.ini)')
    parser.add_argument('-usecuda',
        type=bool,
        default=1,
        choices=range(0, 2),
        help='Do you want to use CUDA? yes=1 (default), no=0')
    parser.add_argument('-gpuid',
        type=int,
        default=1,
        choices=range(0, 3),
        help='Which GPU do you want to use? [0-2] (default=1)')
    parser.add_argument('--multiple', '-m',
        type=int,
        default=1,
        choices=range(1, 11),
```

```
        help='How many networks with different seed for initial weights should be trained?')
    parser.add_argument('--startseed', '-ms',
        type=int,
        default=0,
        choices=range(0, 10),
        help='What seed should the training start at?')
    parser.add_argument('--trainset', '-T',
        type=str,
        default='A',
        choices=['A', 'AB', 'B', 'S', 'NS', '40', 'Tnoise', 'all5', 'all6',
                'cafT0', 'cafT5', 'cafT10', 'cafT15', 'cafT20', 'cafall',
                'B128', 'BS', 'BNS'],
        help='Which training set to use?')
    parser.add_argument('--valset', '-V',
        type=str,
        default='Same',
        choices=['A', 'B', 'S', 'NS', '40A', '40B', 'norm', 'Snoise',
                'S0', 'S5', 'S10', 'S15', 'S20',
                'cafT0', 'cafT5', 'cafT10', 'cafT15', 'cafT20'],
        help='What validation set to use? Same as training set if unspecified')

    args = parser.parse_args()
    for i in range(args.startseed, args.multiple):
        args.dataname = args.cfg.split('.')[0].split('_')[1]

        args.outfolder = os.path.join(*['results',  # 'results/
            args.cfg.split('.')[0].split('_')[0],   # 'base/'
            args.trainset,                          # 'A/' or 'AB/'
            '{}{}'.format(args.dataname, i)])       # 'mfcc1/' or 'stft5/' or 'igfcc9/'
        os.makedirs(args.outfolder, exist_ok=True)

        np.random.seed(i)
        print(i)
        train(config, args)

### Argument parser in test.py
import argparse
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Evaluate performance of network')
    parser.add_argument('-cfg',
        type=str,
        default='base_stft.ini',
        help='Configuration filename from "configs/" folder (default=base_stft.ini)')
    parser.add_argument('-usecuda',
        type=int,
        default=1,
        choices=range(0, 2),
        help='Do you want to use CUDA? (default=1)')
    parser.add_argument('-gpuid',
        type=int,
        default=1,
        choices=range(0, 3),
        help='Which GPU do you want to use? (default=2)')
    parser.add_argument('--trainset', '-T',
        type=str,
        default='A',
```

```
        choices=['A', 'AB', 'B', 'S', 'NS', '40A', 'Tnoise', 'all5', 'all6',
                'cafT0', 'cafT5', 'cafT10', 'cafT15', 'cafT20', 'cafall',
                'B128', 'BS', 'BNS'],
        help='Which training set to use?')
    parser.add_argument('--testset', '-S',
        type=str,
        default='all',
        choices=['all', 'A', 'B', 'S', 'NS', '40A', '40B', 'norm', 'Snoise',
                    'S0', 'S5', 'S10', 'S15', 'S20', 'noise'],
        help='Which test set to use?')

    args = parser.parse_args()
    args.dataname = args.cfg.split('.')[0].split('_')[1]
    args.outfolder = os.path.join(*['results', # 'Results/base/A/stft' or 'results/base/NS/
        mfcc' etc.
        args.cfg.split('.')[0].split('_')[0], args.trainset, args.dataname])

    Test(config, args)

### Argument parser in plot.py
import argparse
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Plot performance of network')
    parser.add_argument('plot',
        type=str,
        choices=['all', 'roc', 'val_acc', 'loss', 'wrong'],
        help='Which plot do you want to display?')
    parser.add_argument('-cfg',
        type=str,
        default='base_mfcc.ini',
        help='Configuration filename from "configs/" folder (default=base_stft.ini)')
    parser.add_argument('--testset', '-S',
        type=str,
        default='A',
        choices=['A', 'B', 'S', 'NS', '40A', '40B', 'norm', 'Snoise',
                    'S0', 'S5', 'S10', 'S15', 'S20'],
        help='Which test set should be used?')
    parser.add_argument('--trainset', '-T',
        type=str,
        default='A',
        choices=['A', 'AB', 'B', 'S', 'NS', '40A', 'Tnoise', 'all5', 'all6',
                'cafT0', 'cafT5', 'cafT10', 'cafT15', 'cafT20', 'cafall'],
        help='Which training set to use?')
    parser.add_argument('-pf',
        action='store_const',
        const=1,
        default=0,
        help='Evaluate on a per file basis instead of a per block basis')

    args = parser.parse_args()
    args.dataname = args.cfg.split('.')[0].split('_')[1]
    args.outfolder = os.path.join(*['results', # 'Results/base/A/stft' or 'results/base/NS/
        mfcc' etc.
        args.cfg.split('.')[0].split('_')[0], args.trainset, args.dataname])
    args.pf = '_pf' if args.pf else ''
```

```
133     if args.plot == 'roc':
134         Test_ROC(config, args)
135     elif args.plot == 'loss':
136         Train_loss(config, args)
137     elif args.plot == 'val_acc':
138         Validation_acc(config, args)
139     elif args.plot == 'wrong':
140         Wrong_names(config, args)
141     elif args.plot == 'all':
142         Train_loss(config, args)
143         Validation_acc(config, args)
144         Test_ROC(config, args)
```

The main function of the argument parser is to switch between configuration files. The argument parser will also make sure all training and test data associated with a given configuration file is saved in a folder corresponding to the name of the configuration file. Thus when the same configuration file is used in the test or plot script, the associated networks and results are automatically used as the folder structure is a direct consequence of the configuration file name.

## C.2 Configuration Parser Implementation

The configuration parser is implemented as shown in Listing C.2.

**Listing C.2:** Code snippet of the configuration parser in the train.py script.

```
1  from configparser import ConfigParser
2  import models   # Custom script containing neural network architectures
3  def train(config, args):
4      ### Code above removed for simplicity
5      ### Architecture
6      model_class = getattr(models, config['Architecture']['Model'])
7      net = model_class(x.shape[2], x.shape[3])
8      ### Code below removed for simplicity

10 if __name__ == '__main__':
11     ### Argument parser code here
12     config = ConfigParser(allow_no_value=True)
13     config.read(os.path.join('configs', args.cfg))
14     train(config, args)
```

The configuration parser reads the configuration file specified by the user and uses the values stored in the file e.g. to choose the neural network architecture defined in a models.py script. If it is desired to train a new network, test its performance and plot the results, it is as simple as defining the new architecture in the models.py script, creating a new configuration file that specifies that the new architecture should be used and specifying that the scripts should use the new configuration file with the argument parser. The process of getting new results is exceptionally time efficient for the user as the most time-consuming part is defining a new network architecture. The rest is copy-pasting a

configuration file and changing the name of the model to use, and then writing a single command for each script. The rest is handled automatically.

An example of a configuration file is shown in Listing C.3. The structure should remain the same but changing the values of the hyperparameters does not require any coding changes and are all neatly grouped together for ease of use.

**Listing C.3:** Example of the configuration file base_mfcc.ini.

```
[Data_A]
dataset = mfcc_A.npz
percent is train set = 0.7
percent is test set = 0.2
percent is validation set = 0.1

[Data_B]
dataset = mfcc_B.npz
percent is train set = 0.7
percent is test set = 0.2
percent is validation set = 0.1

[Architecture]
Model = Detection

[Training Methodology]
epochs = 50
batch size = 64
learning rate = 1e-4
beta1 = 0.9
beta2 = 0.999

[Validation]
val interval = 5

[Print]
running loss = True
loss interval = 10
```

# D | Performance Evaluation of Baseline Model Trained on White Box Attacks

**Name: Group 1070**
**Date 2/5 - 2019**

## D.1 Purpose

The purpose of this test is to evaluate the performance of the baseline model that was trained on white box attacks with different feature extraction methods. The model's performance of each feature extraction method will be evaluated on both white box attacks and black box attacks.

### Test setup

The baseline model shown on Figure 8.1 is trained on a dataset with white box attacks using the five different feature extraction methods: STFT, MFCC, GFCC, IMFCC, and IGFCC. This results in 5 different models; one for each method. Each model is trained 10 times with different random initial weights to compute the statistical significance of the performance w.r.t. both model architecture and feature extraction method. All models and all initializations are tested on both white box and black box attacks. The dataset of white box attacks will be referred to as dataset A and the dataset of black box attacks as dataset B. Both datasets are split into subsets with the following distribution: 70% is the training set, 20% is the test set, and 10% is the validation set. This test will evaluate the performance on test set A and test set B.

In summary:

1. Train 10 models on training set A for each feature extraction method.
2. For each feature extraction method, test all corresponding models on test set A.
3. For each feature extraction method, test all corresponding models on test set B.

This appendix will show the results of the different tests.

## D.2 Procedure

The procedure for evaluating the performance of the model is as follows:

1. Run the train.py script with arguments as follows.

```
python train.py -cfg base_stft.ini -m 10 -T A
```

2. Repeat the above with the base_mfcc.ini, base_gfcc.ini, base_imfcc.ini, and base_igfcc.ini configuration files.

3. Run the test.py script with arguments as follows.

```
python test.py -cfg base_stft.ini -T A
```

4. Repeat the above with the base_mfcc.ini, base_gfcc.ini, base_imfcc.ini, and base_igfcc.ini configuration files.

5. Run the plot.py script with arguments as follows.

```
python plot.py roc -cfg base_stft.ini -T A -S A
python plot.py roc -cfg base_stft.ini -T A -S B
python plot.py val_acc -cfg base_stft.ini -T A
```

Any of the configuration files could have been used in the final command line as all feature extraction methods are used in the plot script. The configuration file simply determines the model to use in the plot.py script.

## D.3 Results

The reported results are the outputs from the plot.py script that computes the evaluation metrics based on the output of the test.py script. The accuracy of the model is shown on Table D.1. The accuracy of the model on the validation set during training is shown on Figure D.1 with the 95th percentile confidence interval shown. The mean false negative rate over the mean false positive rate is shown for test set A and B on Figure D.2.

**Table D.1:** Accuracy in percent of the baseline model on test set A and test set B with difference feature extraction methods.

| Method | Accuracy on A | Accuracy on B |
|---|---|---|
| STFT | 93.39 ± 0.60 | 91.67 ± 1.59 |
| MFCC | 96.73 ± 0.10 | 53.82 ± 3.11 |
| IMFCC | 98.55 ± 0.10 | 93.18 ± 0.83 |
| GFCC | 94.39 ± 0.24 | 56.45 ± 2.44 |
| IGFCC | 98.54 ± 0.14 | 89.82 ± 1.02 |

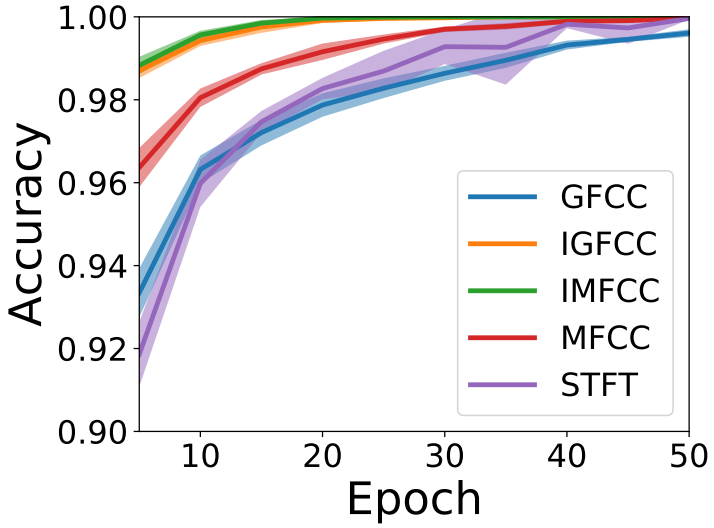
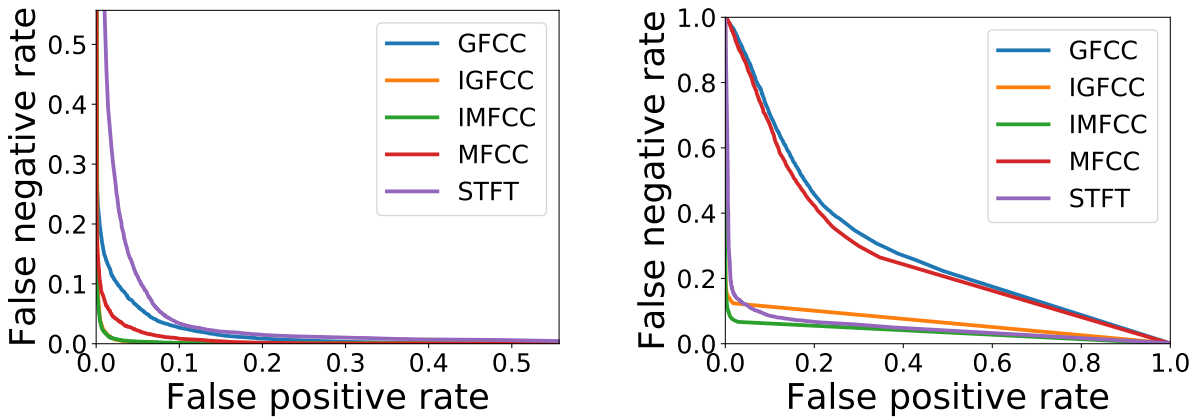



**Figure D.1:** Validation accuracy of the baseline model trained on training set A and evaluated on validation set A.




**(a)** Tested on A.

**(b)** Tested on B.

**Figure D.2:** Mean False positive and false negative rate curve of all models trained with A and tested on A and B respectively with STFT, MFCC, IMFCC, GFCC and IGFCC.

# E | Performance Evaluation of Baseline Model Trained on Black Box Attacks

**Name: Group 1070**
**Date 14/5 - 2019**

## E.1 Purpose

The purpose of this test is to evaluate the performance of the baseline model that was trained on black box attacks with different feature extraction methods. The model's performance of each feature extraction method will be evaluated on both white box attacks and black box attacks.

### Test setup

The baseline model shown on Figure 8.1 is trained on a dataset with black box attacks using the five different feature extraction methods: STFT, MFCC, GFCC, IMFCC, and IGFCC. This results in 5 different models; one for each method. Each model is trained 10 times with different random initial weights to compute the statistical significance of the performance w.r.t. both model architecture and feature extraction method. All models and all initializations are tested on both white box and black box attacks. The dataset of white box attacks will be referred to as dataset A and the dataset of black box attacks as dataset B. Both datasets are split into subsets with the following distribution: 70% is the training set, 20% is the test set, and 10% is the validation set. This test will evaluate the performance on test set A and test set B.

In summary:

1. Train 10 models on training set B for each feature extraction method.
2. For each feature extraction method, test all corresponding models on test set A.
3. For each feature extraction method, test all corresponding models on test set B.

This appendix will show the results of the different tests.

## E.2 Procedure

The procedure for evaluating the performance of the model is as follows:

1. Run the train.py script with arguments as follows.

```
python train.py -cfg base_stft.ini -m 10 -T B
```

2. Repeat the above with the base_mfcc.ini, base_gfcc.ini, base_imfcc.ini, and base_igfcc.ini configuration files.

3. Run the test.py script with arguments as follows.

```
python test.py -cfg base_stft.ini -T B
```

4. Repeat the above with the base_mfcc.ini, base_gfcc.ini, base_imfcc.ini, and base_igfcc.ini configuration files.

5. Run the plot.py script with arguments as follows.

```
python plot.py roc -cfg base_stft.ini -T B -S A
python plot.py roc -cfg base_stft.ini -T B -S B
python plot.py val_acc -cfg base_stft.ini -T B
```

Any of the configuration files could have been used in the final command line as all feature extraction methods are used in the plot script. The configuration file simply determines the model to use in the plot.py script.

## E.3 Results

The reported results are the outputs from the plot.py script that computes the evaluation metrics based on the output of the test.py script. The accuracy of the model is shown on Table E.1. The accuracy of the model on the validation set during training is shown on Figure E.1 with the 95th percentile confidence interval shown. The mean false negative rate over the mean false positive rate is shown for test set A and B on Figure E.2.

**Table E.1:** Accuracy in percent of the baseline model on test set A and test set B with difference feature extraction methods.

| Method | Accuracy on A | Accuracy on B |
|---|---|---|
| STFT | 64.74 ± 1.13 | 99.40 ± 0.17 |
| MFCC | 57.96 ± 0.76 | 99.06 ± 0.14 |
| IMFCC | 86.85 ± 1.44 | 99.75 ± 0.07 |
| GFCC | 58.58 ± 1.29 | 98.89 ± 0.24 |
| IGFCC | 85.26 ± 2.48 | 99.72 ± 0.09 |

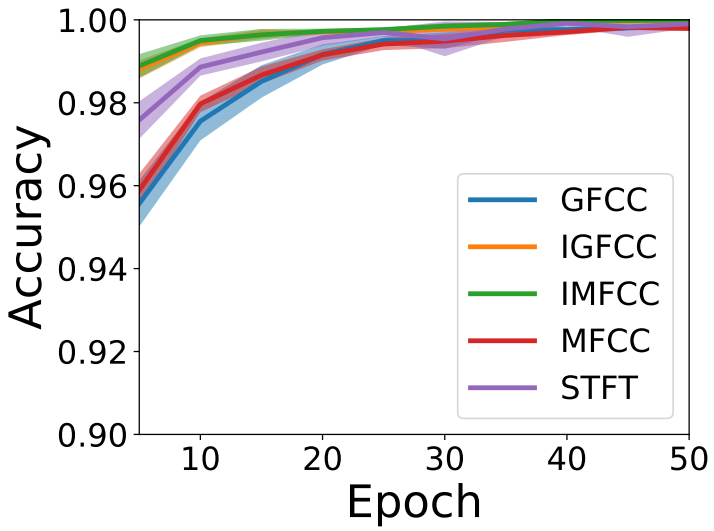
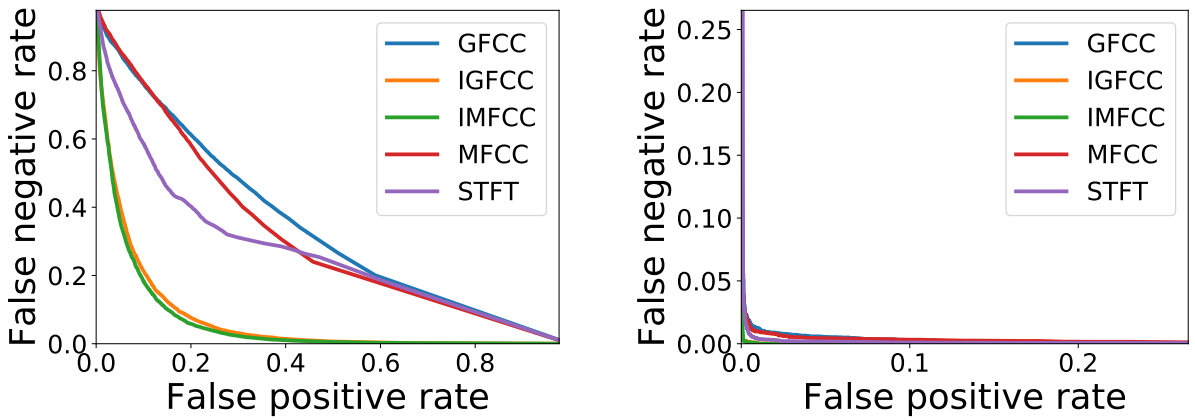



**Figure E.1:** Validation accuracy of the baseline model trained on training set B and evaluated on validation set B.




**(a)** Testing on A.

**(b)** Testing on B.

**Figure E.2:** Mean False positive and false negative rate curve of all models trained with B and tested on A and B respectively with STFT, MFCC, IMFCC, GFCC and IGFCC.

# F | Performance Evaluation of Baseline Model Trained on White Box and Black Box Attacks

**Name: Group 1070**
**Date 2/5 - 2019**

## F.1 Purpose

The purpose of this test is to evaluate the performance of the baseline model that was trained on both white box and black box attacks with different feature extraction methods. The model's performance of each feature extraction method will be evaluated on both white box attacks and black box attacks.

### Test setup

The baseline model shown on Figure 8.1 is trained on the combination of a dataset with white box attacks and a dataset with black box attacks using the five different feature extraction methods: STFT, MFCC, GFCC, IMFCC, and IGFCC. This results in 5 different models; one for each method. Each model is trained 10 times with different random initial weights to compute the statistical significance of the performance w.r.t. both model architecture and feature extraction method. All models and all initializations are tested on both white box and black box attacks. The dataset of white box attacks will be referred to as dataset A and the dataset of black box attacks as dataset B. The combination of datasets will be referred to as dataset AB. Both datasets are split into subsets with the following distribution: 70% is the training set, 20% is the test set, and 10% is the validation set. Dataset A and dataset B are split into these subsets before merging them into training set AB, etc. This test will evaluate the performance on test set A and test set B.

In summary:

1. Train 10 models on training set AB for each feature extraction method.
2. For each feature extraction method, test all corresponding models on test set A.
3. For each feature extraction method, test all corresponding models on test set B.

This appendix will show the results of the different tests.

## F.2 Procedure

The procedure for evaluating the performance of the model is as follows:

1. Run the train.py script with arguments as follows.

```
python train.py -cfg base_stft.ini -m 10 -T AB
```

2. Repeat the above with the base_mfcc.ini, base_gfcc.ini, base_imfcc.ini, and base_igfcc.ini configuration files.
3. Run the test.py script with arguments as follows.

```
python test.py -cfg base_stft.ini -T AB
```

4. Repeat the above with the base_mfcc.ini, base_gfcc.ini, base_imfcc.ini, and base_igfcc.ini configuration files.
5. Run the plot.py script with arguments as follows.

```
1 python plot.py roc -cfg base_stft.ini -T AB -S A
2 python plot.py roc -cfg base_stft.ini -T AB -S B
3 python plot.py val_acc -cfg base_stft.ini -T AB
```

Any of the feature specific configuration files could have been used with the plot.py script as all feature extraction methods are used in the plot script. The configuration file simply determines the model to use in the plot.py script.

## F.3 Results

The reported results are the outputs from the plot.py script that computes the evaluation metrics based on the output of the test.py script. The accuracy of the model is shown on Table F.1. The accuracy of the model on the validation set during training is shown on Figure F.1 with the 95th percentile confidence interval shown. The mean false negative rate over the mean false positive rate is shown for test set A and B on Figure F.2.

**Table F.1:** Accuracy in percent of the baseline model trained on training set AB and tested on test set A and test set B with different feature extraction methods.

| Method | Accuracy on A | Accuracy on B |
|---|---|---|
| STFT | 92.89 ± 0.81 | 99.42 ± 0.09 |
| MFCC | 95.63 ± 0.35 | 98.22 ± 0.27 |
| IMFCC | 98.15 ± 0.22 | 99.56 ± 0.12 |
| GFCC | 93.15 ± 0.48 | 97.68 ± 0.31 |
| IGFCC | 98.00 ± 0.34 | 99.46 ± 0.32 |

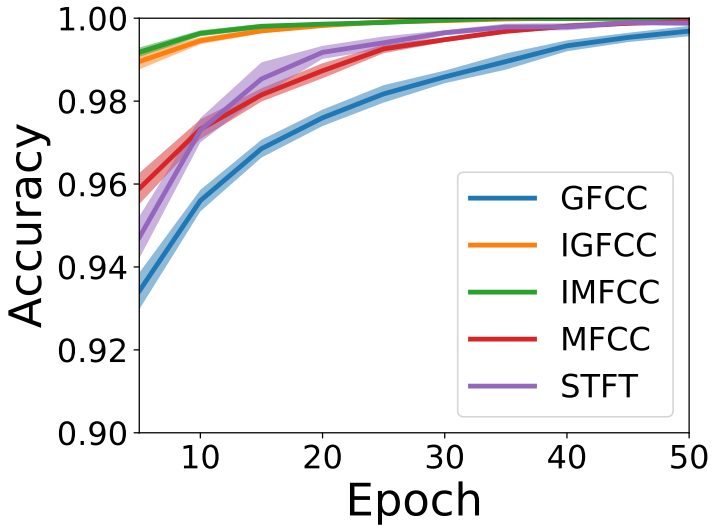
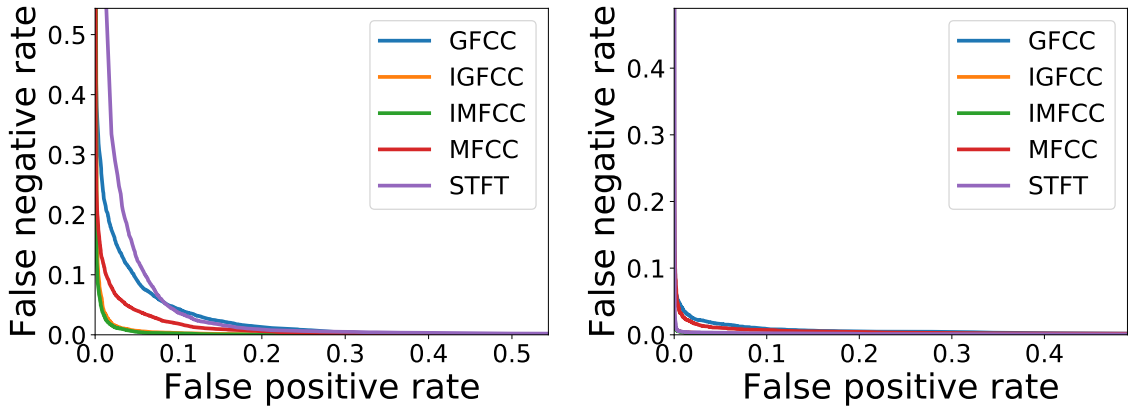



**Figure F.1:** Validation accuracy of the baseline model trained on training set AB and evaluated on validation set AB.

The results from training with AB and testing with A and the B are shown in Figure F.2.




**(a)** Evaluated with dataset A.

**(b)** Evaluated with dataset B.

**Figure F.2:** Mean false negative and false positive rate of the baseline implementation trained with dataset AB and evaluated with dataset A and dataset B with different feature extraction methods.

# G | Performance Evaluation of Baseline Model - Per File

**Name: Group 1070**
**Date 7/5 - 2019**

## G.1 Purpose

The purpose of this test is to evaluate the performance of the baseline model when per file based evaluation is used. The test will be performed separately on the models trained with white box data, black box data, and white and black box data combined. The three tests are evaluated on the five feature extraction methods STFT, MFCC, IMFCC, GFCC, and IGFCC.

### Test setup

The models trained in Appendix D, Appendix E and Appendix F are per file based tested on the white box data and the black box data using the five different feature extraction methods: STFT, MFCC, GFCC, IMFCC, and IGFCC. The reason of this test is partly seen in Figure G.1 showing all false positive and false negative blocks from the MFCC test from Appendix D.

```
Names of wrong files for MFCC

adv-long2long-016136.wavblock10
adv-long2long-016136.wavblock12
adv-long2medium-007792.wavblock10
adv-long2medium-007792.wavblock11
adv-long2medium-014619.wavblock7
adv-long2short-002622.wavblock6
adv-long2short-003816.wavblock5
adv-long2short-003816.wavblock8
adv-long2short-004283.wavblock2
adv-long2short-007792.wavblock3
adv-long2short-010740.wavblock3
adv-long2short-016571.wavblock9
adv-long2short-017108.wavblock4
adv-medium2short-001214.wavblock5
adv-medium2short-002738.wavblock4
adv-medium2short-002926.wavblock1
adv-medium2short-003319.wavblock2
adv-medium2short-003319.wavblock4
adv-short2short-008701.wavblock1
sample-074512.wavblock2
sample-086774.wavblock5
sample-116512.wavblock2
```

**Figure G.1:** .

In Figure G.1 it is seen the largest number of misclassified blocks from the same file is

two. Thus it should be possible to increase a models accuracy by collecting all blocks of one file and evaluate on the blocks average predictions. Three different datasets are used to train in Appendix D, Appendix E and Appendix F. 10 models are trained on each dataset to be able to calculate confidence intervals. For simplicity the models trained in Appendix D will be referred to as model_A, Appendix E as model_B, and Appendix F as model_AB. For each of these three models the per file based evaluation is tested on each of the five feature extraction methods for both white box data (A) and black box data (B). From this test a total of 30 results will be obtained.

In summary:

1. For each feature extraction method, test per file based evaluation of model_A on test set A and test set B
2. For each feature extraction method, test per file based evaluation of model_B on test set A and test set B
3. For each feature extraction method, test per file based evaluation of model_AB on test set A and test set B

This appendix will show the results of the different tests.

## G.2 Procedure

The procedure for evaluation the per file based evaluation is as follows:

1. Run the plot.py script with arguments as follows.

```
python plot.py roc -cfg base_stft.ini -T A -S A -pf
python plot.py roc -cfg base_stft.ini -T A -S B -pf
```

2. Repeat step one with the argument -T set to B and AB.

Any of the feature specific configuration files could have been used with the plot.py script as all feature extraction methods are used in the plot script. The configuration file simply determines the model to use in the plot.py script.

## G.3 Results

The reported results are the outputs from the plot.py script that computes the evaluation metrics based on the output of the test.py script. The per file based accuracy's of the models are shown on Table G.1.

**Table G.1:** Accuracy in percent of the baseline model evaluated on a per file basis with different training sets, test sets, and feature extraction methods. $\mathcal{T}_A \rightarrow \mathcal{S}_A$ means the model was trained with the white box training set $\mathcal{T}_A$ and tested on the white box testing set $\mathcal{S}_A$, etc.

| | STFT | MFCC | IMFCC | GFCC | IGFCC |
|---|---|---|---|---|---|
| $\mathcal{T}_A \rightarrow \mathcal{S}_A$ | 97.58 ± 0.36 | 99.80 ± 0.19 | 99.83 ± 0.14 | 99.44 ± 0.21 | 99.75 ± 0.15 |
| $\mathcal{T}_A \rightarrow \mathcal{S}_B$ | 92.78 ± 2.13 | 52.03 ± 2.37 | 94.15 ± 0.84 | 53.89 ± 2.46 | 90.65 ± 1.02 |
| $\mathcal{T}_B \rightarrow \mathcal{S}_A$ | 65.84 ± 2.29 | 58.17 ± 0.98 | 93.65 ± 1.59 | 58.71 ± 1.62 | 92.20 ± 2.94 |
| $\mathcal{T}_B \rightarrow \mathcal{S}_B$ | 99.58 ± 0.14 | 99.72 ± 0.20 | 99.99 ± 0.03 | 99.66 ± 0.23 | 99.97 ± 0.05 |
| $\mathcal{T}_{AB} \rightarrow \mathcal{S}_A$ | 97.36 ± 0.46 | 99.75 ± 0.20 | 99.63 ± 0.21 | 98.96 ± 0.37 | 99.72 ± 0.19 |
| $\mathcal{T}_{AB} \rightarrow \mathcal{S}_B$ | 99.65 ± 0.07 | 99.21 ± 0.33 | 99.89 ± 0.08 | 98.96 ± 0.24 | 99.75 ± 0.22 |

# H | Performance Evaluation of Baseline Model Trained on White Box Speech Nonspeech data

**Name: Group 1070**
**Date 14/5 - 2019**

## H.1 Purpose

The purpose of this test is to evaluate the performance of the baseline model trained on white box data, when using speech nonspeech data as test set and training set. The test is performed on each of the five feature extraction methods STFT, MFCC, GFCC, IMFCC, and IGFCC.

### Test setup

The test is split into two parts. The first part is where the model used for testing is the model trained in Appendix D which will be refered to as model_A. The second part is where the baseline model is trained individually on the speech segments and the nonspeech segments of the white box dataset. These two models will be refered to as model_S and model_NS The test data is generated from the white box dataset where each example is split into speech and nonspeech parts as described in Chapter 9, thereby obtaining the speech-nonspeech dataset. As the speech-nonspeech dataset is generated from the white box dataset only 20% of the speech-nonspeech dataset is used for testing. Model_A, model_S and model_NS are tested for the five feature extraction methods first using speech data as test set and secondly using nonspeech data as test set.

In summary:

1. For each feature extraction method, test model_A, model_S and model_NS using speech data as test set.
2. For each feature extraction method, test model_A, model_S and model_NS using nonspeech data as test set.

This appendix will show the results of the different tests.

## H.2 Procedure

The procedure for evaluating the performance of model_A when testing on speech-nonspeech data is as follows:

1. Run the test.py script with arguments as follows.

```
python test.py -cfg base_stft.ini -T A
```

2. Repeat the above with the base_mfcc.ini, base_gfcc.ini, base_imfcc.ini, and base_igfcc.ini configuration files.
3. Run the plot.py script with arguments as follows.

```
python plot.py roc -cfg base_stft.ini -T A -S S
python plot.py roc -cfg base_stft.ini -T A -S NS
```

Any of the configuration files could have been used in the final command line as all feature extraction methods are used in the plot script. The configuration file simply determines the model to use in the plot.py script.

The procedure for evaluating the performance of model_S and model_NS when testing on speech-nonspeech data is as follows:

1. Run the train.py script with arguments as follows.

```
python train.py -cfg base_stft.ini -m 10 -T S
python train.py -cfg base_stft.ini -m 10 -T NS
```

2. Repeat the above with the base_mfcc.ini, base_gfcc.ini, base_imfcc.ini, and base_igfcc.ini configuration files.
3. Run the test.py script with arguments as follows.

```
python test.py -cfg base_stft.ini -T S
python test.py -cfg base_stft.ini -T NS
```

4. Repeat the above with the base_mfcc.ini, base_gfcc.ini, base_imfcc.ini, and base_igfcc.ini configuration files.
5. Run the plot.py script with arguments as follows.

```
python plot.py roc -cfg base_stft.ini -T S -S S
python plot.py roc -cfg base_stft.ini -T S -S NS
python plot.py roc -cfg base_stft.ini -T NS -S S
python plot.py roc -cfg base_stft.ini -T NS -S NS
```

Any of the configuration files could have been used in the final command line as all feature extraction methods are used in the plot script. The configuration file simply determines the model to use in the plot.py script.

## H.3 Results

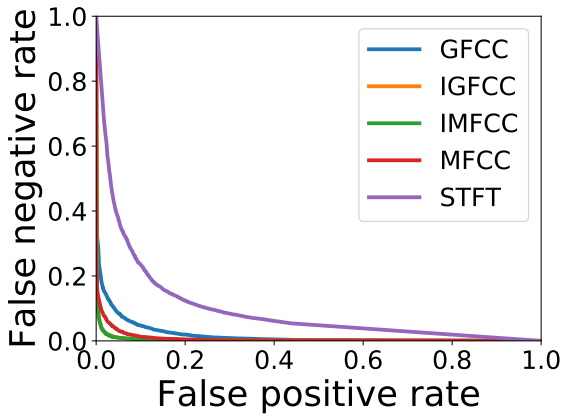
The reported results are the outputs from the plot.py script that computes the evaluation metrics based on the output of the test.py script. The accuracies of model_A are shown on Table H.1 and the accuracies of model_S and model _NS are shown on Table H.2. The mean false negative rate over the mean false positive rate for model_A is shown for test set speech on Figure H.1a and for test set nonspeech on Figure H.1b. The mean false negative rate over the mean false positive rate for model_S and model_NS is shown for test set speech on Figure H.2a and Figure H.3a, and for test set nonspeech on Figure H.2b and Figure H.3b.

**Table H.1:** Accuracy in percent of model_A with test sets being speech data and nonspeech data. $\mathcal{T}_A \rightarrow \mathcal{S}_S$ means the model was trained with the white box training set $\mathcal{T}_A$ and tested on the white box testing set only speech $\mathcal{S}_S$ only nonspeech $\mathcal{S}_{NS}$.

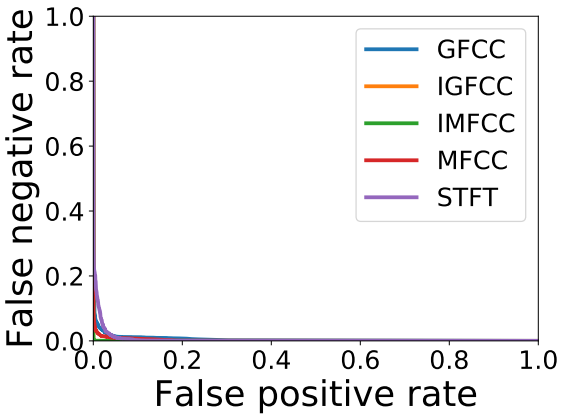
| | STFT | MFCC | IMFCC | GFCC | IGFCC |
|---|---|---|---|---|---|
| $\mathcal{T}_A \rightarrow \mathcal{S}_S$ | 83.88 ± 1.19 | 95.88 ± 0.21 | 97.77 ± 0.19 | 93.29 ± 0.45 | 97.67 ± 0.12 |
| $\mathcal{T}_A \rightarrow \mathcal{S}_{NS}$ | 97.11 ± 0.50 | 98.15 ± 0.23 | 99.55 ± 0.14 | 97.12 ± 0.44 | 99.67 ± 0.14 |

**Table H.2:** Accuracy in percent of the baseline model with different training sets, test sets, and feature extraction methods. $\mathcal{T}_{SN} \rightarrow \mathcal{S}_S$ means the model was trained with the white box training set segmented into nonspeech $\mathcal{T}_{SN}$ and tested on the white box testing set segmented into only speech $\mathcal{S}_S$.

| | STFT | MFCC | IMFCC | GFCC | IGFCC |
|---|---|---|---|---|---|
| $\mathcal{T}_S \rightarrow \mathcal{S}_S$ | 90.36 ± 0.55 | 96.09 ± 0.23 | 98.07 ± 0.19 | 93.68 ± 0.34 | 97.93 ± 0.10 |
| $\mathcal{T}_S \rightarrow \mathcal{S}_{NS}$ | 84.05 ± 6.09 | 91.18 ± 1.38 | 98.55 ± 0.40 | 89.07 ± 2.02 | 98.80 ± 0.23 |
| $\mathcal{T}_{NS} \rightarrow \mathcal{S}_{NS}$ | 96.24 ± 0.32 | 98.64 ± 0.23 | 99.71 ± 0.11 | 97.51 ± 0.23 | 99.76 ± 0.13 |
| $\mathcal{T}_{NS} \rightarrow \mathcal{S}_S$ | 61.63 ± 1.39 | 84.80 ± 1.45 | 90.05 ± 0.93 | 79.08 ± 1.10 | 89.98 ± 0.70 |

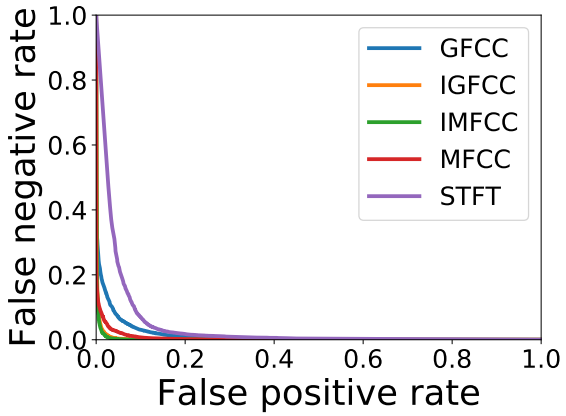



**(a)** Testing on speech data.

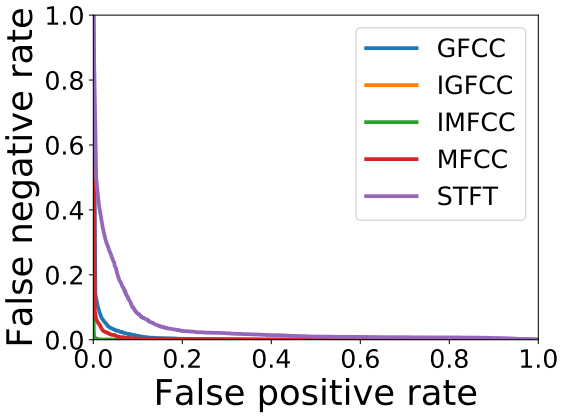



**(b)** Testing on nonspeech data.

**Figure H.1:** Mean False positive and false negative rate curve of model_A tested on speech and nonspeech data with STFT, MFCC, IMFCC, GFCC and IGFCC.

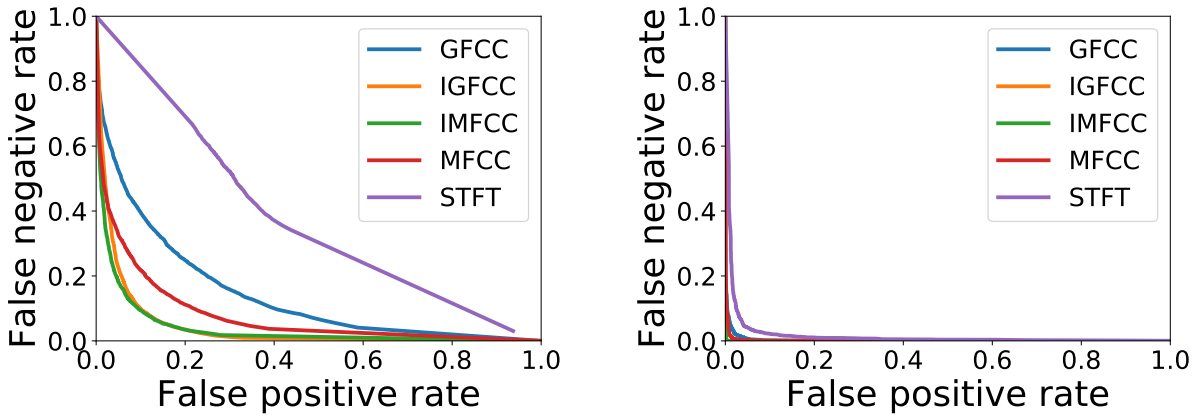



**(a)** Testing on speech data.




**(b)** Testing on nonspeech data.

**Figure H.2:** Mean False positive and false negative rate curve of model_S tested on speech and nonspeech data with STFT, MFCC, IMFCC, GFCC and IGFCC.




**(a)** Testing on speech data.

**(b)** Testing on nonspeech data.

**Figure H.3:** Mean False positive and false negative rate curve of model_NS tested on speech and nonspeech data with STFT, MFCC, IMFCC, GFCC and IGFCC.

# I | Performance Evaluation of Baseline Model with noise added to white box dataset

**Name: Group 1070**
**Date 15/5 - 2019**

## I.1 Purpose

The purpose of this test is to evaluate the performance of the baseline model trained on white box data, when cafeteria noise is added to the test set. The model will be tested on five test sets each with a specified SNR being: 0dB, 5dB, 10dB, 15dB and 20dB. Each of these test sets are evaluated using the STFT, MFCC, IMFCC, GFCC and IGFCC feature extraction methods.

### Test setup

The model used for testing is the model trained in Appendix D which will be refered to as model_A. Model_A is tested on the five cafeteria noisy test sets: $\mathcal{S}_{0\text{dB}}$, $\mathcal{S}_{5\text{dB}}$, $\mathcal{S}_{10\text{dB}}$, $\mathcal{S}_{15\text{dB}}$ and $\mathcal{S}_{20\text{dB}}$. For each test dataset 20% of the data is used. This is because the noise datasets are generated using the white box dataset as described in Appendix B, and it is not wanted to have replicas from the training set in the test set. Per file based evaluation is used in all tests

In summary:

1. For each feature extraction methods; test model_A on the five cafeteria noisy test sets: $\mathcal{S}_{0\text{dB}}$, $\mathcal{S}_{5\text{dB}}$, $\mathcal{S}_{10\text{dB}}$, $\mathcal{S}_{15\text{dB}}$ and $\mathcal{S}_{20\text{dB}}$.

This appendix will show the results of the different tests.

## I.2 Procedure

The procedure for evaluating the performance of model_A when testing on the five cafeteria noisy test sets is as follows:

1. Run the `test.py` script with arguments as follows.

```
python test.py -cfg base_stft.ini -T A -S noise
```

2. Repeat the above with the base_mfcc.ini, base_gfcc.ini, base_imfcc.ini, and base_igfcc.ini configuration files.

3. Run the plot.py script with arguments as follows.

```
python plot.py roc -cfg base_stft.ini -T A -S S0
python plot.py roc -cfg base_stft.ini -T A -S S5
python plot.py roc -cfg base_stft.ini -T A -S S10
python plot.py roc -cfg base_stft.ini -T A -S S15
python plot.py roc -cfg base_stft.ini -T A -S S20
```

Any of the configuration files could have been used in the final command line as all feature extraction methods are used in the plot script. The configuration file simply determines the model to use in the plot.py script.

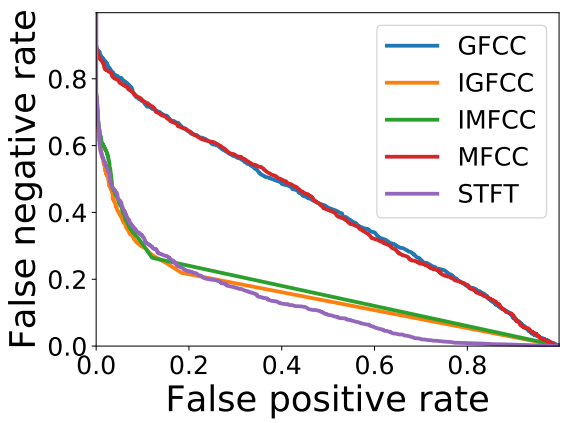
## I.3 Results

The reported results are the outputs from the plot.py script that computes the evaluation metrics based on the output of the test.py script. The accuracy of the model is shown on Table I.1. The mean false negative rate over the mean false positive rate is shown for test set $\mathcal{S}_{0\text{dB}}$ on Figure I.1a, $\mathcal{S}_{5\text{dB}}$ on Figure I.1b, $\mathcal{S}_{10\text{dB}}$ on Figure I.1c, $\mathcal{S}_{15\text{dB}}$ on Figure I.1d and $\mathcal{S}_{20\text{dB}}$ on Figure I.1e.

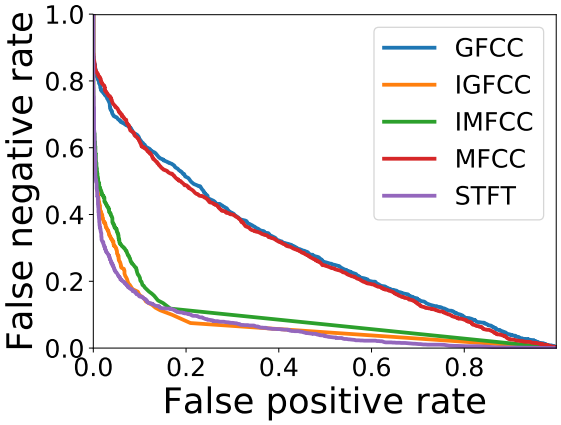
**Table I.1:** Accuracy in percent of the baseline model trained on $\mathcal{T}_A$ with different test sets, and feature extraction methods. $\mathcal{T}_A \rightarrow \mathcal{S}_{0\text{dB}}$ means the model was trained with the white box training set $\mathcal{T}_A$ and tested on the noisy test set $\mathcal{S}_{0\text{dB}}$.

| | STFT | MFCC | IMFCC | GFCC | IGFCC |
|---|---|---|---|---|---|
| $\mathcal{T}_A \rightarrow \mathcal{S}_{0\text{dB}}$ | 78.57 ± 2.58 | 53.03 ± 1.16 | 64.66 ± 0.93 | 50.84 ± 0.52 | 65.65 ± 0.63 |
| $\mathcal{T}_A \rightarrow \mathcal{S}_{5\text{dB}}$ | 86.91 ± 2.47 | 62.58 ± 2.18 | 70.45 ± 1.45 | 58.48 ± 1.96 | 71.94 ± 1.19 |
| $\mathcal{T}_A \rightarrow \mathcal{S}_{10\text{dB}}$ | 90.67 ± 1.25 | 72.08 ± 2.06 | 79.55 ± 1.53 | 66.07 ± 1.65 | 81.66 ± 0.85 |
| $\mathcal{T}_A \rightarrow \mathcal{S}_{15\text{dB}}$ | 93.65 ± 0.63 | 85.31 ± 0.83 | 89.27 ± 0.87 | 80.42 ± 2.93 | 89.35 ± 0.90 |
| $\mathcal{T}_A \rightarrow \mathcal{S}_{20\text{dB}}$ | 95.06 ± 0.48 | 89.30 ± 1.17 | 93.85 ± 0.81 | 85.90 ± 1.65 | 94.02 ± 0.74 |

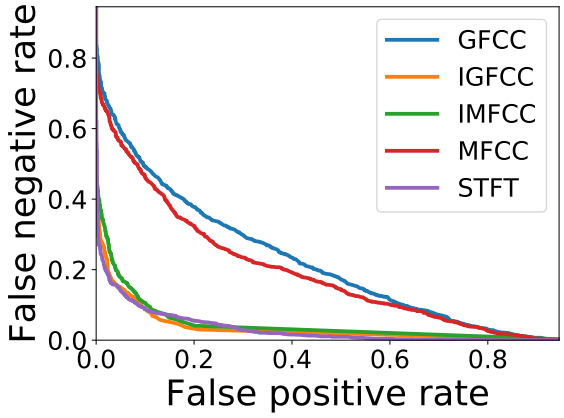
Appendix I. Performance Evaluation of Baseline Model with noise added to white box dataset



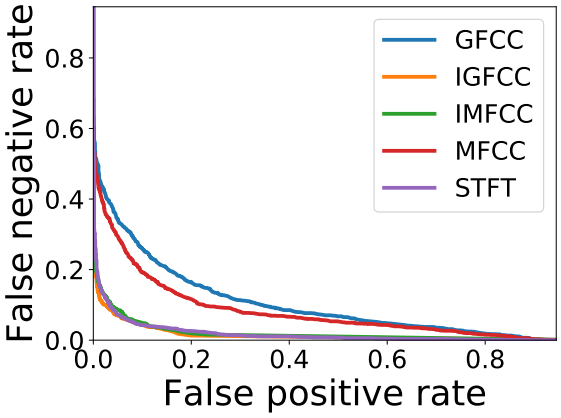

**(a)** Testing on $\mathcal{S}_{0\text{dB}}$.



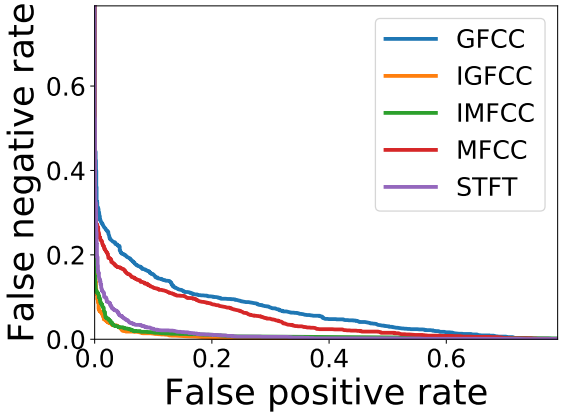

**(b)** Testing on $\mathcal{S}_{5\text{dB}}$.




**(c)** Testing on $\mathcal{S}_{10\text{dB}}$.




**(d)** Testing on $\mathcal{S}_{15\text{dB}}$.




**(e)** Testing on $\mathcal{S}_{20\text{dB}}$.

**Figure I.1:** Mean False positive and false negative rate curve of model_A tested on $\mathcal{S}_{0\text{dB}}$, $\mathcal{S}_{5\text{dB}}$, $\mathcal{S}_{10\text{dB}}$, $\mathcal{S}_{15\text{dB}}$ and $\mathcal{S}_{20\text{dB}}$.

# J | Performance Evaluation of Baseline Model training and testing on equal SNRs

**Name: Group 1070**
**Date 26/5 - 2019**

## J.1 Purpose

The purpose of this test is to evaluate the performance of the baseline model trained on white box data where a speficied amount of cafeteria noise has been added. The model is then tested on white box data where an equal amount of cafeteria noise have been added. The amounts of noise which are investigated are the SNRs: 0dB, 5dB, 10dB, 15dB and 20dB. Each of these test sets are evaluated using the STFT, MFCC, IMFCC, GFCC and IGFCC feature extraction methods.

### Test setup

The baseline model described in Chapter 8 is trained individually on the five cafeteria noisy training sets: $\mathcal{T}_{0\text{dB}}$, $\mathcal{T}_{5\text{dB}}$, $\mathcal{T}_{10\text{dB}}$, $\mathcal{T}_{15\text{dB}}$ and $\mathcal{T}_{20\text{dB}}$. The five SNR dependend models are tested on their corresponding cafeteria noisy test sets. The model trained with $\mathcal{T}_{0\text{dB}}$ is tested on $\mathcal{S}_{0\text{dB}}$, the model trained with $\mathcal{T}_{5\text{dB}}$ is tested on $\mathcal{S}_{5\text{dB}}$ and so forth. 70% of the cafeteria noisy training set is used for training set, 10% of the cafeteria noisy training set is used for validation set and 20% of the cafeteria noisytest set is used for test set.

In summary:

1. For each feature extraction method; individually train a model using the five cafeteria noisy training sets: $\mathcal{T}_{0\text{dB}}$, $\mathcal{T}_{5\text{dB}}$, $\mathcal{T}_{10\text{dB}}$, $\mathcal{T}_{15\text{dB}}$ and $\mathcal{T}_{20\text{dB}}$.
2. For each feature extraction method; test the five trained models on their SNR corresponding test set.

This appendix will show the results of the different tests.

## J.2 Procedure

The procedure for evaluating the performance of training and testing on equal SNRs is as follows:

1. Run the train.py script with arguments as follows.

```
python train.py -cfg base_stft.ini -m 10 -T cafT0
python train.py -cfg base_stft.ini -m 10 -T cafT5
python train.py -cfg base_stft.ini -m 10 -T cafT10
python train.py -cfg base_stft.ini -m 10 -T cafT15
python train.py -cfg base_stft.ini -m 10 -T cafT20
```

2. Repeat the above with the base_mfcc.ini, base_gfcc.ini, base_imfcc.ini, and base_igfcc.ini configuration files.

3. Run the test.py script with arguments as follows.

```
python test.py -cfg base_stft.ini -T cafT0 -S noise
python test.py -cfg base_stft.ini -T cafT5 -S noise
python test.py -cfg base_stft.ini -T cafT10 -S noise
python test.py -cfg base_stft.ini -T cafT15 -S noise
python test.py -cfg base_stft.ini -T cafT20 -S noise
```

4. Repeat the above with the base_mfcc.ini, base_gfcc.ini, base_imfcc.ini, and base_igfcc.ini configuration files.

5. Run the plot.py script with arguments as follows.

```
1 python plot.py roc -cfg base_stft.ini -T cafT0 -S S0
2 python plot.py roc -cfg base_stft.ini -T cafT5 -S S5
3 python plot.py roc -cfg base_stft.ini -T cafT10 -S S10
4 python plot.py roc -cfg base_stft.ini -T cafT15 -S S15
5 python plot.py roc -cfg base_stft.ini -T cafT20 -S S20
```

Any of the configuration files could have been used in the final command line as all feature extraction methods are used in the plot script. The configuration file simply determines the model to use in the plot.py script.

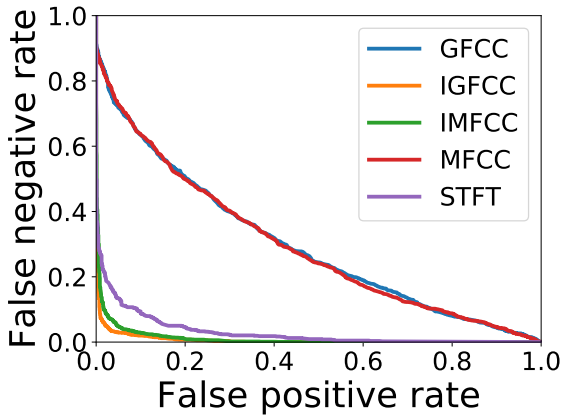
## J.3 Results

The reported results are the outputs from the plot.py script that computes the evaluation metrics based on the output of the test.py script. The accuracies of the models are shown on Table J.1. The mean false negative rate over the mean false positive rate is shown for test set $\mathcal{S}_{0\text{dB}}$ on Figure J.1a, $\mathcal{S}_{5\text{dB}}$ on Figure J.1b, $\mathcal{S}_{10\text{dB}}$ on Figure J.1c, $\mathcal{S}_{15\text{dB}}$ on Figure J.1d and $\mathcal{S}_{20\text{dB}}$ on Figure J.1e.

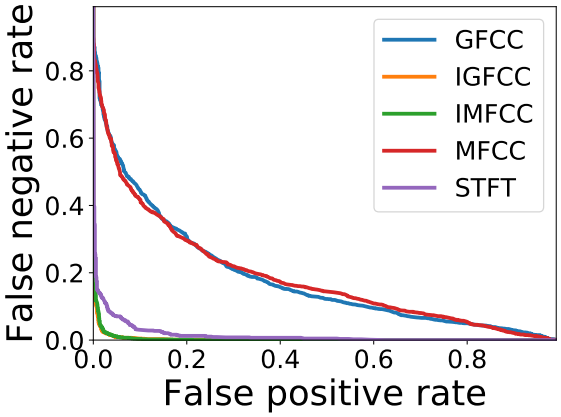
## Appendix J. Performance Evaluation of Baseline Model training and testing on equal SNRs

**Table J.1:** Accuracy in percent of the baseline model with different training sets, test sets, and feature extraction methods. $\mathcal{T}_{0\text{dB}} \rightarrow \mathcal{S}_{0\text{dB}}$ means the model was trained on noisy training set with 0dB SNR and tested on the noisy test set with 0 dB SNR.

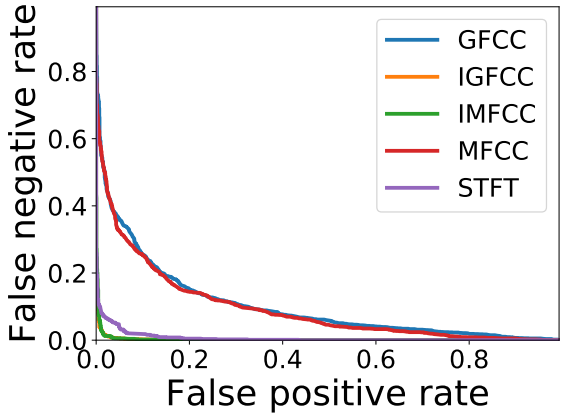
| | STFT | MFCC | IMFCC | GFCC | IGFCC |
|---|---|---|---|---|---|
| $\mathcal{T}_{0\text{dB}} \rightarrow \mathcal{S}_{0\text{dB}}$ | 92.05 ± 0.81 | 58.12 ± 1.39 | 95.76 ± 0.45 | 57.58 ± 1.02 | 96.35 ± 0.66 |
| $\mathcal{T}_{5\text{dB}} \rightarrow \mathcal{S}_{5\text{dB}}$ | 91.77 ± 1.92 | 67.70 ± 1.13 | 96.69 ± 0.31 | 67.13 ± 1.55 | 97.25 ± 0.38 |
| $\mathcal{T}_{10\text{dB}} \rightarrow \mathcal{S}_{10\text{dB}}$ | 93.85 ± 0.42 | 79.21 ± 1.00 | 97.30 ± 0.43 | 76.07 ± 1.09 | 97.75 ± 0.42 |
| $\mathcal{T}_{15\text{dB}} \rightarrow \mathcal{S}_{15\text{dB}}$ | 93.03 ± 4.57 | 85.25 ± 1.65 | 93.90 ± 0.56 | 84.07 ± 0.78 | 94.66 ± 0.35 |
| $\mathcal{T}_{20\text{dB}} \rightarrow \mathcal{S}_{20\text{dB}}$ | 94.78 ± 0.51 | 90.00 ± 1.00 | 94.19 ± 0.51 | 87.08 ± 0.87 | 93.65 ± 0.46 |



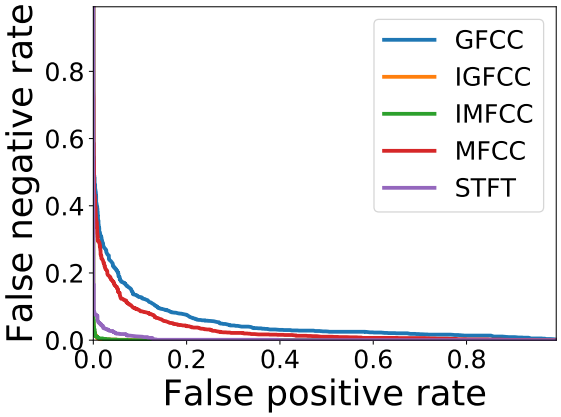

(a) $\mathcal{T}_{0\text{dB}} \rightarrow \mathcal{S}_{0\text{dB}}$.



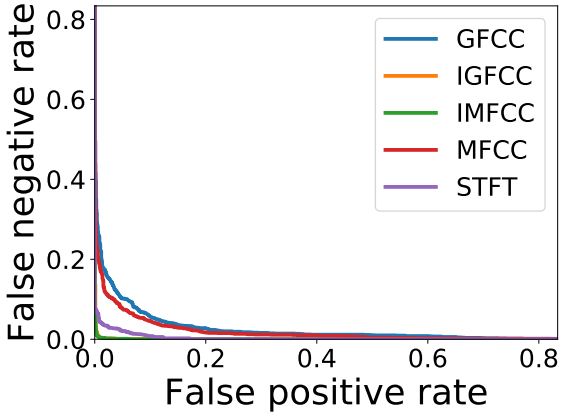

(b) $\mathcal{T}_{5\text{dB}} \rightarrow \mathcal{S}_{5\text{dB}}$.




(c) $\mathcal{T}_{10\text{dB}} \rightarrow \mathcal{S}_{10\text{dB}}$.




(d) $\mathcal{T}_{15\text{dB}} \rightarrow \mathcal{S}_{15\text{dB}}$.




(e) $\mathcal{T}_{20\text{dB}} \rightarrow \mathcal{S}_{20\text{dB}}$.

**Figure J.1:** Mean False positive and false negative rate curve of training and testing af model on data with same SNRs.

# K | Performance Evaluation of Baseline Model trained narrowly noise wise

**Name: Group 1070**
**Date 26/5 - 2019**

## K.1 Purpose

The purpose of this test is to evaluate the performance of the baseline model trained on white box data where an unspecified amount of cafeteria noise has been added. The model is then tested on white box data where a specified amount of cafeteria noise have been added. The specified amounts of noise in the test sets are the SNRs: 0dB, 5dB, 10dB, 15dB and 20dB. Each of these test sets are evaluated using the STFT, MFCC, IMFCC, GFCC and IGFCC feature extraction methods.

### Test setup

The baseline model described in Chapter 8 is trained individually on a combined cafeteria noisy training set $\mathcal{T}_{\text{caf}}$. This training set is a combination of the five cafeteria noisy training sets: $\mathcal{T}_{0\text{dB}}$, $\mathcal{T}_{5\text{dB}}$, $\mathcal{T}_{10\text{dB}}$, $\mathcal{T}_{15\text{dB}}$ and $\mathcal{T}_{20\text{dB}}$, and $\mathcal{T}_A$. The trained model called model_CAF is tested individually on the five cafeteria noisy test sets: $\mathcal{S}_{0\text{dB}}$, $\mathcal{S}_{5\text{dB}}$, $\mathcal{S}_{10\text{dB}}$, $\mathcal{S}_{15\text{dB}}$ and $\mathcal{S}_{20\text{dB}}$. 70% of the cafeteria noisy training sets are used for training set, 10% of the cafeteria noisy training sets are used for validation set and 20% of the cafeteria noisy test set is used for test set.

In summary:

1. For each feature extraction method; Train the baseline model using $\mathcal{T}_{\text{caf}}$ as training set.
2. For each feature extraction method; test model_CAF on the five cafeteria noisy test sets: $\mathcal{S}_{0\text{dB}}$, $\mathcal{S}_{5\text{dB}}$, $\mathcal{S}_{10\text{dB}}$, $\mathcal{S}_{15\text{dB}}$ and $\mathcal{S}_{20\text{dB}}$ .

This appendix will show the results of the different tests.

## K.2 Procedure

The procedure for evaluating the performance of training on white box data with unspecified amounts of cafeteria noise and testing on white box data with specified amounts of cafeteria noise is as follows.:

1. Run the train.py script with arguments as follows.

```
python train.py -cfg base_stft.ini -m 10 -T cafall
```

2. Repeat the above with the base_mfcc.ini, base_gfcc.ini, base_imfcc.ini, and base_igfcc.ini configuration files.

3. Run the test.py script with arguments as follows.

```
python test.py -cfg base_stft.ini -T cafall -S noise
```

4. Repeat the above with the base_mfcc.ini, base_gfcc.ini, base_imfcc.ini, and base_igfcc.ini configuration files.

5. Run the plot.py script with arguments as follows.

```
1 python plot.py roc -cfg base_stft.ini -T cafall -S S0
2 python plot.py roc -cfg base_stft.ini -T cafall -S S5
3 python plot.py roc -cfg base_stft.ini -T cafall -S S10
4 python plot.py roc -cfg base_stft.ini -T cafall -S S15
5 python plot.py roc -cfg base_stft.ini -T cafall -S S20
```

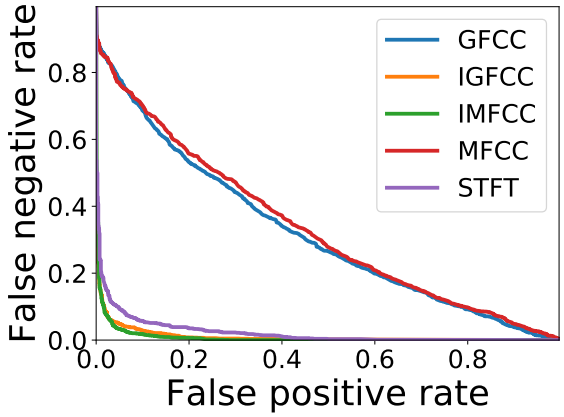
Any of the configuration files could have been used in the final command line as all feature extraction methods are used in the plot script. The configuration file simply determines the model to use in the plot.py script.

## K.3 Results

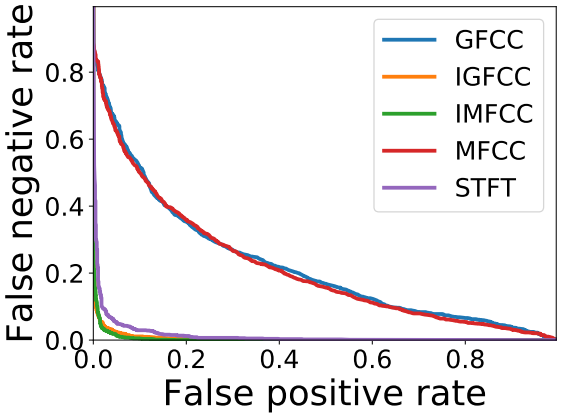
The reported results are the outputs from the plot.py script that computes the evaluation metrics based on the output of the test.py script. The accuracies of the models are shown on Table K.1. The mean false negative rate over the mean false positive rate is shown for test set $\mathcal{S}_{0\text{dB}}$ on Figure K.1a, $\mathcal{S}_{5\text{dB}}$ on Figure K.1b, $\mathcal{S}_{10\text{dB}}$ on Figure K.1c, $\mathcal{S}_{15\text{dB}}$ on Figure K.1d and $\mathcal{S}_{20\text{dB}}$ on Figure K.1e.

**Table K.1:** Accuracy in percent of the baseline model with different training sets, test sets, and feature extraction methods. $\mathcal{T}_{\text{caf}} \rightarrow \mathcal{S}_{\text{0dB}}$ means the model was trained on the combined noisy training set of all five SNRs and tested on the noisy test set with 0 dB SNR.

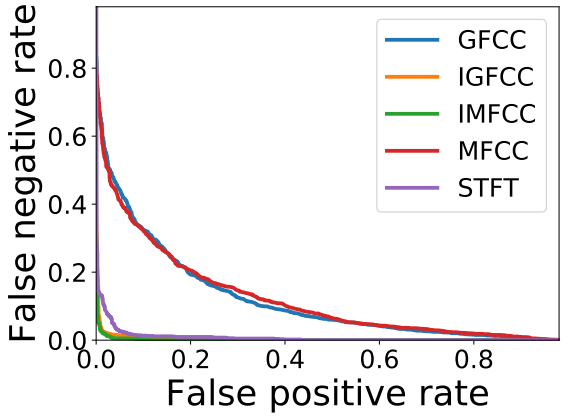
| | STFT | MFCC | IMFCC | GFCC | IGFCC |
|---|---|---|---|---|---|
| $\mathcal{T}_{\text{caf}} \rightarrow \mathcal{S}_{\text{0dB}}$ | 92.88 ± 1.38 | 61.63 ± 1.88 | 95.90 ± 0.27 | 62.92 ± 1.22 | 95.42 ± 2.12 |
| $\mathcal{T}_{\text{caf}} \rightarrow \mathcal{S}_{\text{5dB}}$ | 94.85 ± 0.69 | 71.88 ± 1.11 | 96.94 ± 0.38 | 71.97 ± 1.72 | 96.66 ± 1.17 |
| $\mathcal{T}_{\text{caf}} \rightarrow \mathcal{S}_{\text{10dB}}$ | 96.11 ± 0.54 | 80.17 ± 1.30 | 98.03 ± 0.21 | 80.34 ± 1.52 | 97.92 ± 0.78 |
| $\mathcal{T}_{\text{caf}} \rightarrow \mathcal{S}_{\text{15dB}}$ | 97.14 ± 0.57 | 88.15 ± 0.97 | 99.04 ± 0.27 | 85.81 ± 0.86 | 98.60 ± 0.59 |
| $\mathcal{T}_{\text{caf}} \rightarrow \mathcal{S}_{\text{20dB}}$ | 96.82 ± 0.58 | 92.64 ± 0.91 | 99.30 ± 0.32 | 91.12 ± 1.13 | 98.96 ± 0.46 |



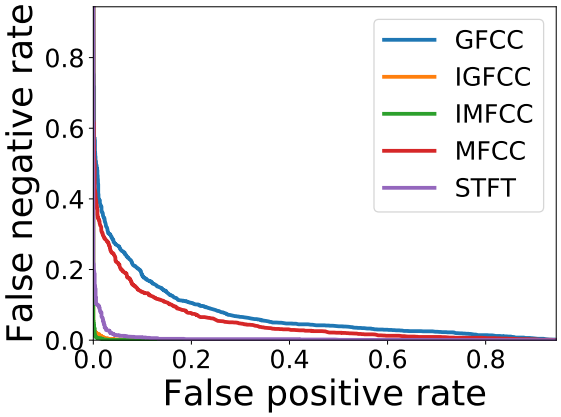

**(a)** Testing on $\mathcal{S}_{0\text{dB}}$.



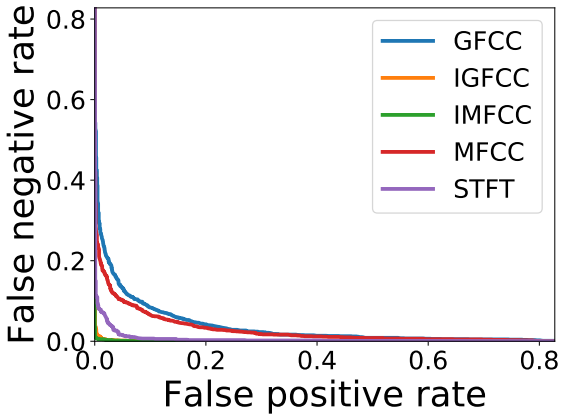

**(b)** Testing on $\mathcal{S}_{5\text{dB}}$.




**(c)** Testing on $\mathcal{S}_{10\text{dB}}$.




**(d)** Testing on $\mathcal{S}_{15\text{dB}}$.




**(e)** Testing on $\mathcal{S}_{20\text{dB}}$.

**Figure K.1:** Mean False positive and false negative rate curve of model_CAF tested on $\mathcal{S}_{0\text{dB}}$, $\mathcal{S}_{5\text{dB}}$, $\mathcal{S}_{10\text{dB}}$, $\mathcal{S}_{15\text{dB}}$ and $\mathcal{S}_{20\text{dB}}$.

# L | Evaluation of Baseline Model Tested on known noise type

**Name: Group 1070**
**Date 27/5 - 2019**

## L.1 Purpose

The purpose of this test is to evaluate the performance of the baseline model trained on white box data where an unspecified type and amount of noise has been added. The model is then tested on white box data where specified amounts of cafeteria noise have been added. The specified amounts of noise in the test sets are the SNRs: 0dB, 5dB, 10dB, 15dB and 20dB. Each of these test sets are evaluated using the STFT, MFCC, IMFCC, GFCC and IGFCC feature extraction methods.

### Test setup

The baseline model described in Chapter 8 is trained individually on a combined noisy training set $\mathcal{T}_{\text{6types}}$. This training set contains the white box data with no added noise and white box data with the SNRs: 0dB, 5dB, 10dB, 15dB and 20dB. For each of the SNRs six types of noise are present: CAF, BUS, STR, PED, BBL and SSN. The model trained with $\mathcal{T}_{\text{6types}}$ will be refered to as model_6types. Model_6types is tested individually on the five cafeteria noisy test sets: $\mathcal{S}_{\text{0dB}}$, $\mathcal{S}_{\text{5dB}}$, $\mathcal{S}_{\text{10dB}}$, $\mathcal{S}_{\text{15dB}}$ and $\mathcal{S}_{\text{20dB}}$. 70% of the cafeteria noisy training sets are used for training set, 10% of the cafeteria noisy training sets are used for validation set and 20% of the cafeteria noisy test set is used for test set.

In summary:

1. For each feature extraction method; Train the baseline model using $\mathcal{T}_{\text{6types}}$ as training set.
2. For each feature extraction method; test model_6types on the five cafeteria noisy test sets: $\mathcal{S}_{\text{0dB}}$, $\mathcal{S}_{\text{5dB}}$, $\mathcal{S}_{\text{10dB}}$, $\mathcal{S}_{\text{15dB}}$ and $\mathcal{S}_{\text{20dB}}$ .

This appendix will show the results of the different tests.

## L.2 Procedure

The procedure for evaluating the performance of training on white box data with unspecified amounts and types of noise and testing on white box data with specified amounts of cafeteria noise is as follows.:

1. Run the train.py script with arguments as follows.

```
python train.py -cfg base512_stft.ini -m 10 -T all6
```

2. Repeat the above with the base512_mfcc.ini, base512_gfcc.ini, base512_imfcc.ini, and base512_igfcc.ini configuration files.

3. Run the test.py script with arguments as follows.

```
python test.py -cfg base512_stft.ini -T all6 -S noise
```

4. Repeat the above with the base512_mfcc.ini, base512_gfcc.ini, base512_imfcc.ini, and base512_igfcc.ini configuration files.

5. Run the plot.py script with arguments as follows.

```
python plot.py roc -cfg base512_stft.ini -T all6 -S S0
python plot.py roc -cfg base512_stft.ini -T all6 -S S5
python plot.py roc -cfg base512_stft.ini -T all6 -S S10
python plot.py roc -cfg base512_stft.ini -T all6 -S S15
python plot.py roc -cfg base512_stft.ini -T all6 -S S20
```

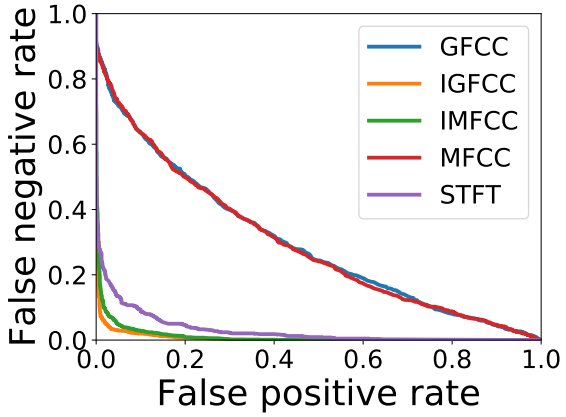
Any of the configuration files could have been used in the final command line as all feature extraction methods are used in the plot script. The configuration file simply determines the model to use in the plot.py script.

## L.3 Results

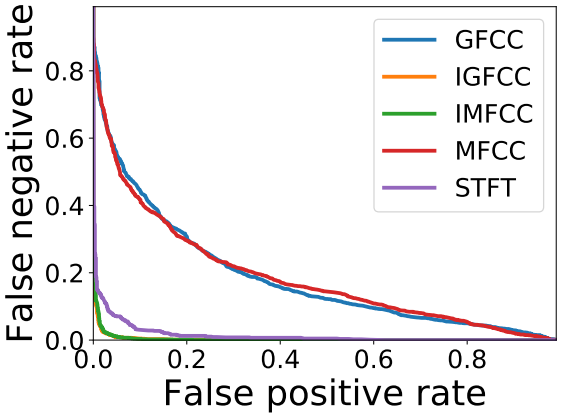
The reported results are the outputs from the plot.py script that computes the evaluation metrics based on the output of the test.py script. The accuracies of the models are shown on Table L.1. The mean false negative rate over the mean false positive rate is shown for test set $\mathcal{S}_{0\text{dB}}$ on Figure L.1a, $\mathcal{S}_{5\text{dB}}$ on Figure L.1b, $\mathcal{S}_{10\text{dB}}$ on Figure L.1c, $\mathcal{S}_{15\text{dB}}$ on Figure L.1d and $\mathcal{S}_{20\text{dB}}$ on Figure L.1e.

**Table L.1:** Accuracy in percent of the baseline model with different training sets, test sets, and feature extraction methods. $\mathcal{T}_{\text{6noise}} \rightarrow \mathcal{S}_{\text{0dB}}$ means the model was trained on the combined noisy training set of all six noise types and all five SNRs and tested on the noisy test set with 0 dB SNR.

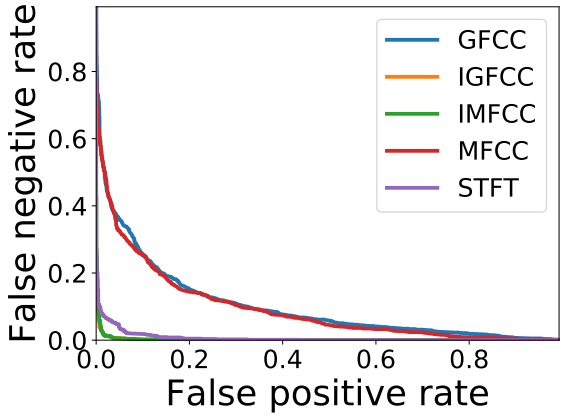
| | STFT | MFCC | IMFCC | GFCC | IGFCC |
|---|---|---|---|---|---|
| $\mathcal{T}_{\text{6noise}} \rightarrow \mathcal{S}_{\text{0dB}}$ | 91.80 ± 1.36 | 64.55 ± 1.87 | 94.86 ± 0.79 | 64.49 ± 1.55 | 96.35 ± 0.57 |
| $\mathcal{T}_{\text{6noise}} \rightarrow \mathcal{S}_{\text{5dB}}$ | 94.44 ± 0.65 | 74.97 ± 1.65 | 97.44 ± 0.59 | 74.66 ± 1.56 | 97.61 ± 0.50 |
| $\mathcal{T}_{\text{6noise}} \rightarrow \mathcal{S}_{\text{10dB}}$ | 95.76 ± 0.40 | 83.34 ± 1.60 | 98.17 ± 0.30 | 82.67 ± 1.62 | 98.23 ± 0.25 |
| $\mathcal{T}_{\text{6noise}} \rightarrow \mathcal{S}_{\text{15dB}}$ | 96.83 ± 0.44 | 90.90 ± 1.57 | 99.13 ± 0.20 | 88.29 ± 0.59 | 99.16 ± 0.30 |
| $\mathcal{T}_{\text{6noise}} \rightarrow \mathcal{S}_{\text{20dB}}$ | 97.13 ± 0.31 | 93.60 ± 1.12 | 99.33 ± 0.19 | 92.19 ± 0.43 | 99.24 ± 0.27 |



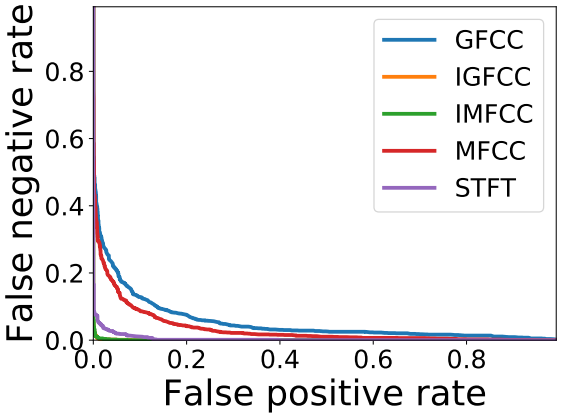

**(a)** Testing on $\mathcal{S}_{0\text{dB}}$.



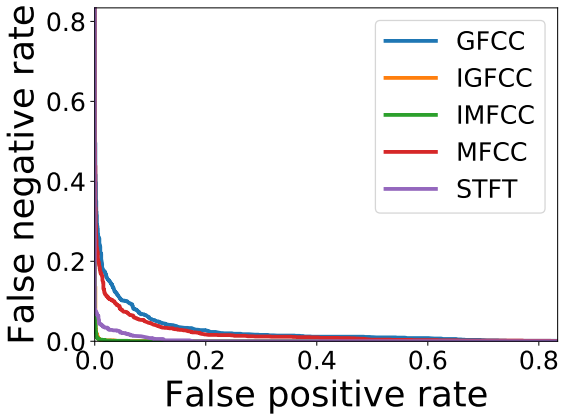

**(b)** Testing on $\mathcal{S}_{5\text{dB}}$.




**(c)** Testing on $\mathcal{S}_{10\text{dB}}$.




**(d)** Testing on $\mathcal{S}_{15\text{dB}}$.




**(e)** Testing on $\mathcal{S}_{20\text{dB}}$.

**Figure L.1:** Mean False positive and false negative rate curve of model_6types tested on $\mathcal{S}_{0\text{dB}}$, $\mathcal{S}_{5\text{dB}}$, $\mathcal{S}_{10\text{dB}}$, $\mathcal{S}_{15\text{dB}}$ and $\mathcal{S}_{20\text{dB}}$.

# M | Performance Evaluation: 5 Noise Types

**Name: Group 1070**
**Date 27/5 - 2019**

## M.1 Purpose

The purpose of this test is to evaluate the performance of the baseline model trained on 5 different types of noise with unspecified amounts. The model is then tested on white box data where different amounts of cafeteria noise have been added. The amounts of noise which are investigated are the SNRs: 0dB, 5dB, 10dB, 15dB and 20dB. Each of these test sets are evaluated using the STFT, MFCC, IMFCC, GFCC and IGFCC feature extraction methods.

### Test setup

The baseline model described in Chapter 8 is trained on a noisy training set $\mathcal{T}_{\text{5noise}}$. This dataset is a combination of 6 datasets with the following amounts of noise added: No noise, 0 dB SNR, 5 dB SNR, 10 dB SNR, 15 dB SNR, and 20 dB SNR. Each of these sets have a uniformly and independently selected noise type added to each file in the set from the following noise types: BUS, STR, SSN, PED, and BBL. Thus each set contains many different types of noise besides the set with additional noise and $\mathcal{T}_{\text{5noise}}$ is the combination of all these sets.

The model trained on $\mathcal{T}_{\text{5noise}}$ is tested on the cafeteria noisy test sets. 70% of the training set is used for training set, 10% of the training set is used for validation set and 20% of the test set is used for test set.

In summary:

1. For each feature extraction method: Train a model using $\mathcal{T}_{\text{5noise}}$ as a training set.
2. For each feature extraction method: Test the model on the 5 different test sets with different SNRs; $\mathcal{S}_{\text{0dB}}$, $\mathcal{S}_{\text{5dB}}$, $\mathcal{S}_{\text{10dB}}$, $\mathcal{S}_{\text{15dB}}$ and $\mathcal{S}_{\text{20dB}}$.

## M.2 Procedure

The procedure for evaluating the performance of training on $\mathcal{T}_{\text{5noise}}$ and testing on white box data with different amounts of cafeteria noise is as follows:

1. Run the train.py script with arguments as follows.

```
python train.py -cfg base512_stft.ini -m 10 -T all5
```

2. Repeat the above with the base512_mfcc.ini, base512_gfcc.ini, base512_imfcc.ini, and base512_igfcc.ini configuration files.

3. Run the test.py script with arguments as follows.

```
python test.py -cfg base512_stft.ini -T all5 -S noise
```

4. Repeat the above with the base512_mfcc.ini, base512_gfcc.ini, base512_imfcc.ini, and base512_igfcc.ini configuration files.

5. Run the plot.py script with arguments as follows.

```
python plot.py roc -cfg base512_stft.ini -T all5 -S S0
python plot.py roc -cfg base512_stft.ini -T all5 -S S5
python plot.py roc -cfg base512_stft.ini -T all5 -S S10
python plot.py roc -cfg base512_stft.ini -T all5 -S S15
python plot.py roc -cfg base512_stft.ini -T all5 -S S20
```

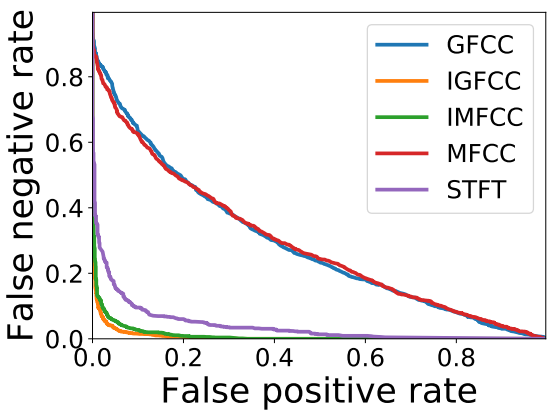
Any of the configuration files could have been used in the final command line as all feature extraction methods are used in the plot script. The configuration file simply determines the model to use in the plot.py script.

## M.3 Results

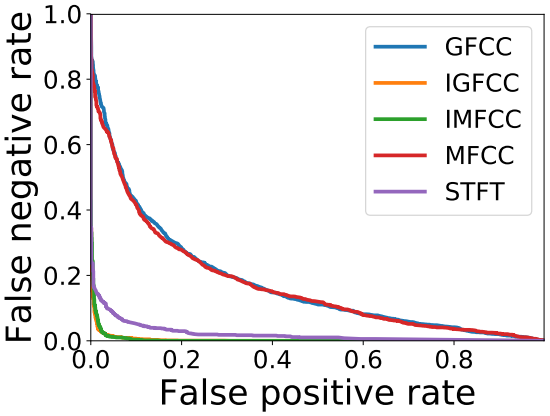
The reported results are the outputs from the plot.py script that computes the evaluation metrics based on the output of the test.py script. The accuracies of the models are shown on Table M.1. The mean false negative rate over the mean false positive rate is shown for the different test sets on Figure M.1.

**Table M.1:** Accuracy in percent of the baseline model with different training sets, test sets, and feature extraction methods. $\mathcal{T}_{\text{5noise}} \rightarrow \mathcal{S}_{\text{0dB}}$ means the model was trained on the combined noisy training set of all five noise types (CAF excluded) and all five SNRs and tested on the noisy test set with 0 dB SNR.

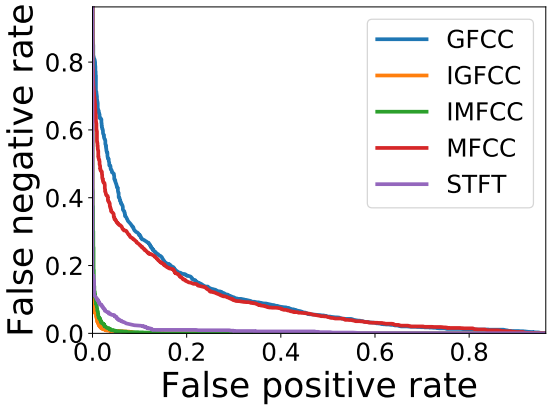
| | STFT | MFCC | IMFCC | GFCC | IGFCC |
|---|---|---|---|---|---|
| $\mathcal{T}_{\text{5noise}} \rightarrow \mathcal{S}_{\text{0dB}}$ | 89.83 ± 0.97 | 64.69 ± 2.03 | 94.86 ± 0.69 | 63.57 ± 1.40 | 95.90 ± 0.82 |
| $\mathcal{T}_{\text{5noise}} \rightarrow \mathcal{S}_{\text{5dB}}$ | 93.31 ± 1.38 | 74.69 ± 1.34 | 97.42 ± 0.42 | 72.53 ± 2.48 | 97.39 ± 0.37 |
| $\mathcal{T}_{\text{5noise}} \rightarrow \mathcal{S}_{\text{10dB}}$ | 95.34 ± 0.31 | 82.36 ± 1.11 | 97.67 ± 0.25 | 79.72 ± 2.60 | 98.12 ± 0.30 |
| $\mathcal{T}_{\text{5noise}} \rightarrow \mathcal{S}_{\text{15dB}}$ | 96.63 ± 0.92 | 89.86 ± 0.87 | 98.88 ± 0.16 | 88.12 ± 1.35 | 98.76 ± 0.22 |
| $\mathcal{T}_{\text{5noise}} \rightarrow \mathcal{S}_{\text{20dB}}$ | 96.91 ± 0.70 | 93.62 ± 0.87 | 99.10 ± 0.23 | 90.70 ± 1.42 | 98.85 ± 0.28 |



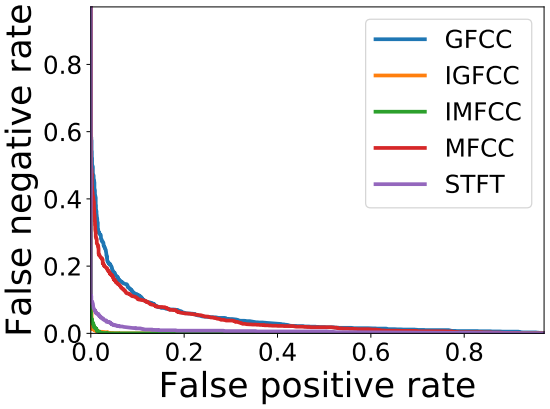

**(a)** 0 dB.



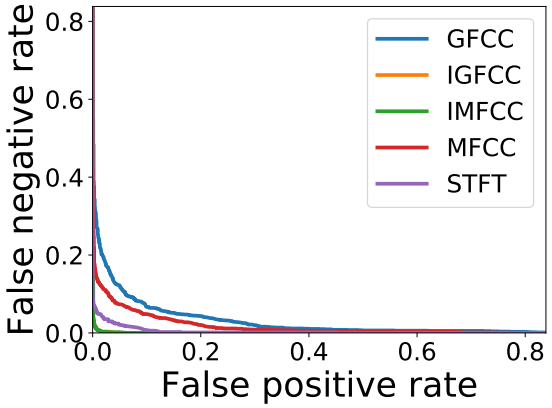

**(b)** 5 dB.




**(c)** 10 dB.




**(d)** 15 dB.




**(e)** 20 dB.

**Figure M.1**