HSB
Hochschule Bremen
City University of Applied Sciences

Neural Network on an FPGA for Speech Command Recognition on an Autonomous Vehicle

Fynn Hagen
Jan Hartig
Roland Helmich
Mattheo Mahnke
Dylan-Noah Schade
Florian Sommerfeld
Hagen Stöver
Philipp Wittje
Prof. Dr. Jan Bredereke

City University of Applied Sciences Bremen
Faculty 4: Electrical Engineering and Computer Science

March 8, 2022

Abstract

This research focuses on solutions for executing Neural Networks (NNs) on field programmable gate arrays (FPGAs) of comparably limited power as efficiently as possible, for use of such networks on autonomous vehicles.

Specifically, we propose a concept of a speech command recognition using a system-on-chip (SoC), where we implement the first data processing stages on the FPGA of the SoC. A proof of concept for acquiring audio data via the FPGA and converting it to Mel Frequency Cepstral Coefficients (MFCCs) has been made. Using a Quantized Neural Network (QNN) we have reached an accuracy of 90% on an audio test set for driving commands. However the NN yields worse results during inference. The general concept of the data processsing pipeline in this research, enables researchers to proceed the implementation on FPGA hardware. As demonstrated by the developed NN speech recognition is possible on comparative performance-limited FPGA boards.

Contents

Chapter 1

Introduction

written by Jan Bredereke

In this project, we work on solutions for executing neural networks on FPGAs of comparably limited power as efficiently as possible, for use of such networks on autonomous vehicles. This task comprises the areas of field programmable gate arrays (FPGAs), system-on-chip (SoC), and speech command recognition with a neural network.

1.1 Motivation of the Project

written by Jan Bredereke

Neural networks are often used in data centers with powerful and power consuming special hardware. A current research question is how to make full use of neural networks also at the "edge" of the Cloud. That is, close to the sensors and the actors, or even autonomously from data and power supply connections. The CPU of a microcontroller has too scarce data processing resources for this. A field programmable gate array (FPGA) can offer more data processing resources, both in absolute numbers and in relation to its power consumption. An FPGA is very well suited for a highly parallel structure such as that of a neural network. In practice, however, many optimization tasks need to be solved before full use of the potential of the FPGA can be made.

The motivation for this project comes from space craft engineering in particular. On-board computers provide particularly scarce data processing resources. Access to the ground segment usually is available only intermittently. Due to space radiation, current off-the-shelf processors would fail soon. Therefore, one uses special processors. Their chips feature structural widths of at least 65 nm. These special processors are sufficiently robust. But they provide correspondingly less data processing resources than those of 10 nm currently in use elsewhere. An extremely small number of computers of this kind are made. Therefore, they usually are not made with application specific integrated circuits (ASICs), but with programmable standard hardware (FPGAs). Radiation-hard versions of some FPGAs are available, which have correspondingly larger structural widths. There is increasing demand for more on-board computing resources. Examples are on-board image processing, e.g. for autonomous rovers on other celestial bodies, or for constellations of nano-satellites with narrow bandwidth to the ground segment each.

1.2 Original Task Description

written by Jan Bredereke

There is a simultaneously taught course "Projekt: SoC-NN – FPGAs für neuronale Netze: Edge Computing auf autonomen Vehikeln" (project: SoC-NN – FPGAs for neural networks: edge computing

on autonomous vehicles). Like several preceding courses [Alt+21; Mül21; Mül+21; Hut+20], it employs an autonomous model car as a practical vehicle, serving as a representative for an autonomous space craft. See Figure 1.1. The vehicle is equipped with a camera and a SoC Zynq-7020. The SoC features an Arm CPU and, in particular, an FPGA Artix-7. The data processing resources of the Artix-7 are quite close to those of an FPGA suitable for space. The SoC is integrated into a PYNQ-Z1 board. In Figure 1.1, the board can be spotted easily due to its pink colour.

In the current mini-project, however, we do not work on visual object recognition, but on speech command recognition. Nevertheless, we keep the above vehicle as the application. We enter the area of acoustic signals, new to us. This area provides less turnkey solutions for neural networks than there are for visual object recognition. The parallel project SoC-NN aims to make the vehicle recognize simple human arm gestures and use them to control the vehicle's driving. Now, we aim to make the vehicle recognize simple speech commands and use them analogously.

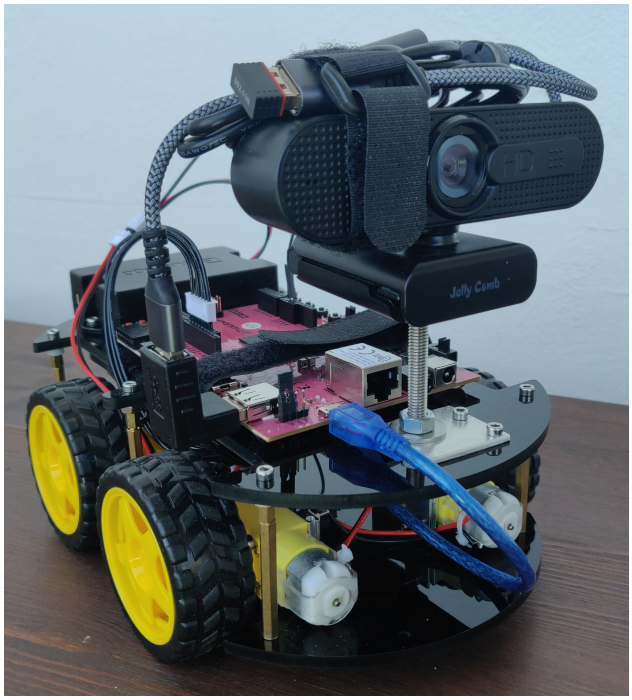

Figure 1.1: The vehicle of the simultaneously taught course, with visual object recognition on an FPGA.
photo: Felix Müller

Full speech recognition is a complex task. It comprises the analog/digital conversion, the temporal windowing, the breakdown into the frequency spectrum by a fast fourier transform (FFT), a filtering of the resulting frequency spectrum plus an entailing meta-frequency analysis (Cepstrum), and a pattern recognition using an acoustic model, a pronunciation dictionary and a language model, in order to obtain a text in written form, finally. This requires comprehensive knowledge of the structure of language, and also considerable data processing power. Therefore, we restrict ourselves to the first part of this processing chain. We content ourselves with the recognition of a few, simple speech commands for our vehicle, like, e.g., the commands "left", "right", "forward", and "stop". For this, we need to realize mainly the major processing steps of the temporal windowing, the fast fourier transform (FFT), and the classification of patterns in the output of the FFT by a neural network.

Before the start of the project, we devised an original plan to perform the following intermediate steps: At first, we use the PYNQ-Z1 boards of the mini-project, but not yet embedded into a vehicle. We implement the FFT on the FPGA, preferably based on a turnkey IP core for the FFT. We realize the neural network on the CPU of the SoC, provisionally. This postpones the familiarization with the rather complex implementation of a neural network on an FPGA to later. As soon as we got a grasp of the processing steps and have put them into practice, we optimize the processing speed. For this, we move the neural network to the FPGA, too, for example. Furthermore, we then integrate the speech command recognition into the real vehicle. The vehicle may need to do listening pauses during driving, in order to be able to understand speech commands. Maybe we need to add a better microphone than the one integrated into the board. We might even need several microphones to achieve a directional characteristic, in order to reach a sufficient range, and in order to handle noise. We do not expect to achieve all of these steps completely in the current term already.

The high-level goal of the project is to improve the performance of the signal processing massively, while contending with the restrictions of the existing hardware. As a practical result, we expect a correspondingly increased quality of the vehicle's autonomous driving. The expected over-all result of the project is a deeper understanding of the many potential approaches to optimization, both in the area of digital circuits and in the area of neural networks.

1.3 Actual Scope of Work

written by Philipp Wittje

As this mini-project took place as a separate part of the course at the university, the time frame was very limited. In order to avoid leaving an unfinished project, we have already agreed at the beginning which sub-goals are achievable and can be completed and which further goals may be developed by subsequent projects. In accordance with the time and personnel resources, we have agreed to develop the project separately from the vehicle, i. e. explicitly not to have the requirement that the system ultimately runs on the vehicle and has the full range of functions. Rather, the work includes the extensive preparation for this, i. e. the evaluation of the audio signals from the boards onboard microphone, as well as its further processing by FFTs on the Programmable Logic (PL). Linked to this is the integration of the corresponding intellectual property core (IP-Core) and libraries on the FPGA. In addition, it includes the implementation of a neural network, as well as a data generator, which offers the possibility to generate an arbitrary amount of different train and test data from a finite amount of audio data to train and test the neural network accordingly. Since the feasibility of this project was an important aspect, we paid attention to their further usability when selecting the hardware and software used, as well as to the corresponding noise due to driving noise already in the test data generator, even if we did not adapt the result to the vehicle in motion.

Chapter 2

Fundamentals

2.1 Basic Principles of Neural Networks and Deep Learning

written by Dylan-Noah Schade

This section is an introduction to Neural Networks and deep learning. Most of this section's content as well as part of its structure are based on the *Deep Learning Specialization* [NKM], an online course at the learning platform *Coursera* [Inc21] co-founded by *Dr. Andrew Ng*, who is also the main author of the deep learning course.

While giving an overview over the topic and introducing the most basic principles and keywords, the underlying mathematical relationships of NNs are also summarised and explained examplarily. Since especially these are taken from transcripts of the *Deep Learning Specialization* [NKM], excessive references to individual formulas and contexts have been omitted.

2.1.1 Introduction

Definition of Terms

The principle of *Deep Learning* can be roughly described as the empirical, step-by-step approximation of a very complex mathematical function. This complex function with often thousands or even millions of parameters is built up in a structure of different *layers*, which is generally referred to as *a Neural Network (NN)*. With the help of such a network, an attempt is made to model a real-world problem. In principle, it is assumed that there is an arbitrarily complex mathematical function for the real-world problem that describes it. The step-by-step process of approaching this function through systematic trial and error and minimizing the deviation between the function's result and the expected result is called *training* or learning. This way, the NN "learns" through training to produce an output, often called *prediction*, based on defined inputs. The formats of in- and outputs, the exact network structure and the possible uses are far-reaching and always depend on the application context. The term "Deep Learning" describes networks with more than one *hidden layer*, i.e. multilayer NNs. [Hei17]

Historical Background

The foundation of machine learning was laid as early as the 1950s [Hei17]. In the following decades, some techniques were already developed and papers were published, but there was a lack of technology and especially computing power to add value from NNs as there is today. A detailed overview of early publications and milestones of Deep Learning is provided by [Sch14].

It took until the mid-2000s for the first major successes to be achieved due to the now sufficient computing power. Since then, interest in the many possibilities offered by Deep Learning has been growing - not only in the research sector, but increasingly also in industry.

Application examples

The following is an unsorted and incomplete list to give a rough overview of the wide range of application examples:

- image recognition, such as classifying images according to a simple true-false scheme or even recognising over 9000 different objects, as the "YOLO9000" network can do [RF16].
- Speech recognition, as most smartphones offer these days.
- translations
- Anomaly detection, e.g. detection of fractures in X-rays of bones or analysis of outliers in time histories.

One of the most popular introductory examples is classifying images as those with cats and those without. The NN receives an image as input and the output is a floating point value between 0.0 and 1.0, which indicates the probability that the image is a picture of a cat. This example will be used throughout the following explanations.

2.1.2 Structure of Neural Networks

The formulas introduced in this section make use of the symbol sets and notations defined in the *Standard notations for Deep Learning* (see appendix A.2) [NG].

A Single Neuron

A Neural Network consists of a set of so-called neurons in several layers. The simplest network would consist of a single neuron and thus only one layer. A neuron forms the basic unit of a NN. Each neuron has the same structure. It expects an input x and produces an output a. Each neuron consists of a simple linear function $z(x)$ and a non-linear activation function g, which produces the output a.

The function z always has the form $z(x) = wx + b$, where x is the input, w is the so-called weight and b is the so-called bias. The variables w and b are the two parameters of each neuron that are trained. The training process later results in the condition that the smaller w in the neuron, the less this particular input is weighted.

The activation function can be chosen as one of many hyperparameters, but it is always applied as $g(z(x))$ to z, producing the output a of the neuron. Thus, for a single neuron this results in 2.1.

$$a = g(z(x)) = g(wx + b) \tag{2.1}$$

Vectorization

Previously, all variables were presented as single decimal numbers. Each neuron computes an output a to an x. However, since in a network each neuron of a layer is connected to each neuron of the previous layer, the input x is not a scalar but a (column) vector. Similarly, each neuron must then have a weight for each input value, making w also a (row) vector. Capital letters are subsequently used for the vector quantities, so x becomes the neuron's input vector X and w becomes the weight vector W. However, each neuron still yields a scalar output a, since the scalar product $W \cdot X$ is formed from W and X. Thus, with the two vectorized quantities, 2.1 results in 2.2.

$$a = g(z(X)) = g(W \cdot X + b) \tag{2.2}$$

The bias b here is the same scalar value for all input values in a neuron. To make this clear, the bias remains in its lowercase notation in the following. In contrast, W now represents a weight for each input value of X.

Forming a Network of Multiple Neurons

Vectorization per Layer

So far, each neuron has been considered separately. For the further explanations, however, not only single neurons but all neurons of a layer are considered together. For this a further vectorization of the variables from 2.2 is necessary. This also simplifies a corresponding vectorized implementation of the formulas.

To clarify that each layer l is a set of neurons $n^{[l]}$, an extended vectorized notation is introduced. In each layer, multiple values are processed, in particular each neuron of a layer processes all outputs of the previous layer. Each layer still gets an input vector X, which is the same for each neuron in the layer. The computation within each neuron remains unchanged to 2.2, the bias b also remains in its lower case notation.

If all neurons in a layer are combined, the bias b, the function z, and thus the output a are vectorized. The weights of each neuron W are also expanded by another dimension for the entire layer, so W now becomes a matrix. Thus, for each layer with all its individual neurons, 2.2 results in 2.3, which describes all neurons in an entire layer l.

$$A = g(Z(X)) = g(W \cdot X + b) \tag{2.3}$$

The activation function g remains the same for all neurons in the layer. The exact dimensions are discussed in more detail in section 2.1.2. It is also shown that when 2.3 is extended to all training data, A and Z become matrices of the same dimension, but the bias b remains a one-dimensional vector, which is why it is still noted as lowercase here.

Deep Layers

In a multi-layer network, as already explained, each neuron is connected to each neuron in the next layer. The output values $A^{[l]}$ of all neurons of a layer l serve as input to each neuron of the following layer $l+1$, respectively all inputs of layer l are the outputs $A^{[l-1]}$ of the preceding layer $l-1$. Thus, for all layers l, each of which is preceded by a layer $l-1$, 2.3 gives the more general form 2.4, which describes all neurons in a deep layer (i.e. any layer between the in- and output layer).

$$A^{[l]} = g^{[l]}(Z^{[l]}) = g^{[l]}(W^{[l]} \cdot A^{[l-1]} + b^{[l]}) \tag{2.4}$$

These layers, which are surrounded by other layers, are also called *deep layers*. $g^{[l]}$ is still the non-linear activation function of the whole layer l, all other values are vector quantities with values for each neuron of the layer, similar to 2.3.

Input Layer

Before the very first layer lies the input of the entire network, also called the zeroth layer, in form of the input vector X. The first layer is a special case of 2.4 with $l = 1$ and $A^{[0]} = X$ as shown in 2.5.

$$A^{[1]} = g^{[1]}(Z^{[1]}) = g^{[1]}(W^{[1]} \cdot X + b^{[1]}) \tag{2.5}$$

Output Layer

After a fixed set of layers L (the depth of the network) follows the output layer as the last layer of the network. The output vector of the output layer $A^{[L]}$ forms the output of the entire network $\hat{Y}$. Thus, in the special case of the output layer, 2.4 becomes 2.6.

$$\hat{Y} = A^{[L]} = g^{[L]}(Z^{[L]}) = g^{[L]}(W^{[L]} \cdot A^{[L-1]} + b^{[L]}) \tag{2.6}$$

Considering a binary classification problem, the output layer consists of only one neuron, which returns a scalar value, in that case $\hat{y}$. This is usually a floating point number between 0.0 and 1.0 and gives a probability or *prediction*.

So, in the example mentioned before, it gives the probability that the input vector (an image) belongs to the set of cat images. If one needs a binary decision as output, one sets a threshold, here e.g. at 0.5, above which the computed probability is considered large enough that one commits to the prediction as True (cat image).

Clarifying the Dimensions

Since each individual neuron should always return a scalar value a, the output vector $A^{[l]}$ of each layer l is equal in length to the number of neurons $n^{[l]}$ in the layer. Since $A^{[l]}$ is the result of $g^{[l]}(Z^{[l]})$, $Z^{[l]}$ must correspondingly have the same dimensions as $A^{[l]}$, since $g^{[l]}$ is simply a (non-linear) function on all values of $Z^{[l]}$.

Now, for $Z^{[l]}$ to also have the length $n^{[l]}$, $W^{[l]}$ must have the dimensions $(n^{[l]}, n^{[l-1]})$. This is due to the fact that $Z^{[L]} = W^{[L]} \cdot A^{[L-1]}$. The scalar product of $W^{[L]}$ and $A^{[L-1]}$, since $A^{[L-1]}$ has length $n^{[l-1]}$, can only yield a vector of length $n^{[l]}$ if the matrix $W^{[L]}$ has the dimensions $(n^{[l]}, n^{[l-1]})$.

The bias $b^{[l]}$ does not further change the dimensions of $Z^{[l]}$ and also has the length $n^{[l]}$. Thus, it contains only one value per neuron of the layer.

If we now additionally note that all layers of the entire network are trained with m data samples during the training process, then for all layers of the network for all training data at the same time, the dimensions are as follows:

$$
\begin{aligned}
Z^{[l]} : (n^{[l]}, m) &\quad \Rightarrow Z^{[l]} \in \mathbb{R}^{n^{[l]} \times\ m} \\
W^{[l]} : (n^{[l]}, n^{[l-1]}) &\quad \Rightarrow W^{[l]} \in \mathbb{R}^{n^{[l]} \times\ n^{[l-1]}} \\
A^{[l]} : (n^{[l]}, m) &\quad \Rightarrow A^{[l]} \in \mathbb{R}^{n^{[l]} \times\ m} \\
b^{[l]} : (n^{[l]}, 1) &\quad \Rightarrow Z^{[l]} \in \mathbb{R}^{n^{[l]}}
\end{aligned}
$$

Input Layer

The input layer necessarily has the same dimension as the input vector. It is therefore a special case of the number of neurons $n^{[l]}$ for any layer l, where in case of the input layer $l = 0$ and $n^{[0]} = n_x$. Again, under consideration of the fact that the network is trained with m samples, the dimensions of X reflect the training dataset of the network:

$$X \in \mathbb{R}^{n_x \times\ m}$$

Output Layer

Analogous to the input layer, the dimension of the output in the output layer is $n^{[L]} = n_y$, which results in the following definition of its dimensions:

$$\hat{Y} \in \mathbb{R}^{n_y \times\ m}$$

So, n_y is the number of neurons in the last layer. This is always $n_y = 1$ for binary classification problems.

For multiclass problems, it would correspond to the number of classes to be recognized. If one would modify the example and - instead of whether the picture shows a cat at all - want to determine the breed of the cat shown, n_y would correspond to the number of cat breeds to be determined. The network would then provide a probability per class, or in the example per cat breed, that the input belongs to the corresponding class.

It is common for multi classification problems to use a *softmax*-layer as the last layer of the network. This layer simply applies the softmax-function to the output values, so that they sum up to 1 over all categories. The necessary calculation is shown in 2.7, where $Z^{[L]}$ is the output of the last layer L before the activation function. Thus, the softmax-function is replacing the activation function of the last layer

and producing $a^{[L]}$ instead.

$$a^{[L]} = \frac{e^{Z^{[L]}}}{\sum e^{Z^{[L]}}} \tag{2.7}$$

Summary

In this section the structure of the neurons in each of the layers, including input and output layers, was explained and generalized for all layers and over m data samples. From the definitions of the individual neurons, the construct of a NN can now be formed abstractly by linking the individual layers and their neurons. The whole network always receives an input vector X as input. When this input vector is created, all its values are then used in different weights to compute a single output or prediction $\hat{Y}$ across multiple layers, indicating a probability. This value forms the output of the network.

Such a construct is exemplified in figure 2.1, which shows a simple NN with three layers, thus $L = 3$. The input vector X of length $n_x = 4$ has been divided into its individual values $x_1^{(i)} \dots x_4^{(i)}$, where (i) represents one sample each of the training data, e.g. a single image. Since the output layer ($l = L = 3$) consists of only one neuron, the network provides one scalar output per data set: the prediction $\hat{y}^{(i)}$. Each neuron, represented by a circle in the figure, contains the previously explained computation $a^{[l]} = g(z^{[l]}(A^{[l-1]}))$, which is generally obtained for each layer represented by substituting $A^{[l-1]}$ for X in 2.2.

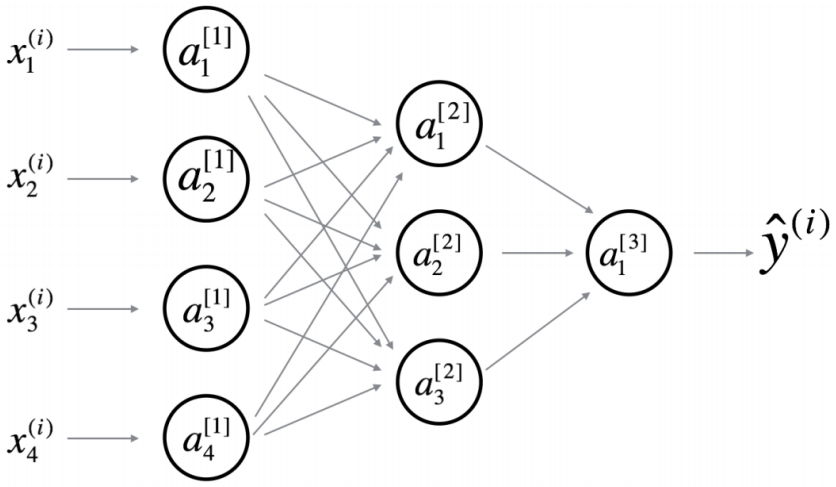

Figure 2.1: Neural Network with three layers ($L = 3$), input vector X and output value $\hat{y}$ Source: [NG]

2.1.3 Training Process

The static structure of a classical NN was explained in detail in the previous section. One could now apply an input vector to the network and would get an output. However, this output would not be related to the input. Therefore it is necessary to train the network in advance. In this training process the weights, which are usually initialized randomly at the beginning, and the respective biases of each neuron are changed step by step in such a way that the output corresponds more and more to the expected result and the predictions become increasingly more accurate. The two methods *Forward Propagation* as well as *Backward Propagation* play a crucial role here.

Forward Propagation

Letting the network initialized with random weights process an input vector, one obtains corresponding output values for each neuron in each layer. For each layer, a vector $Z^{[l]}$ as well as $A^{[l]}$ is computed.

These are cached for subsequent backward propagation along with the bias vector $b^{[l]}$ and the weight matrix $W^{[l]}$ of each layer, both of which still contain arbitrary values at the beginning.

Forward propagation is thus nothing more than computing an output to an input vector. Since this is the logical path "forward" through the network, this process is called forward propagation. It is also applied later to use an already trained network for predictions, which is then called *inference*.

Labeling of Training Data

To train a network, pre-labeled (sometimes also called *ground truth*) training data is needed. For each data set or each sample from the training data there is an expected result. In the simplest case, as in the cat image recognition example, this can be a binary value. For example, if the network should return a value close to 1.0 for cat images, the label in the training dataset would be 1 for each cat image and 0 for every other image. The expected outputs are recorded in what is called the label matrix Y, which has the following dimensions:

$$Y \in \mathbb{R}^{n_y \times\ m}$$

For a single training sample i, e.g. an image, Y contains the label $y^{(i)}$. Similarly, the prediction matrix $\hat{Y}$ contains the single prediction $\hat{y}^{(i)}$. The comparison of the output $\hat{y}^{(i)}$ of each sample i with the expected value $y^{(i)}$ is used in backward propagation in a *cost function* to optimize the network.

Backward Propagation

If one imagines the whole network as a complex function, the training tries to find the minimum of a so-called cost function J. For this purpose, the principle of *gradient descent* is used. One "moves" on the function in the direction of the largest or steepest descent to reach the minimum. The cost function J forms the difference between the expected value Y and the prediction $\hat{Y}$ of the network. The cost calculated in this way is now used "backward" through each layer to update the weights with the derivatives of the previous layer such that the cost function is minimized.

The gradient descent process is shown in figure 2.2 as an example for two-dimensional space. Accordingly, the principle is carried out by partial derivatives of each parameter for each neuron starting at the last layer and ending at the first layer. The dimension of the function to be optimized corresponds to the number of trainable parameters in the network. This already exceeds four to five digit values for small networks, for very complex networks, e.g. for image or speech recognition, it can be several million parameters. This can no longer be visualized and is difficult to imagine.

Cost Function

The training process is used to optimize or find the minimum of the cost function J by using the gradients to move step by step further towards an optimal minimum. The cost function is sometimes called a "loss" or "error" function. A simple example of a cost function is shown in 2.8.

$$J = \left| Y - \hat{Y} \right| \tag{2.8}$$

Another, more complex cost function, based on the *cross-entropy loss function* averaged over al m training samples, is shown in 2.9.

$$J = -\frac{1}{m} \sum_{n=1}^{m} [y_n \log \hat{y}_n + (1 - y_n) \log(1 - \hat{y}_n)] \tag{2.9}$$

The deduction of the derivatives of individual parameters for all layers of a NN is omitted here. However, a good visualization of backward propagation and the influences of individual neurons and their weights on the cost function can be seen in [3BI17].

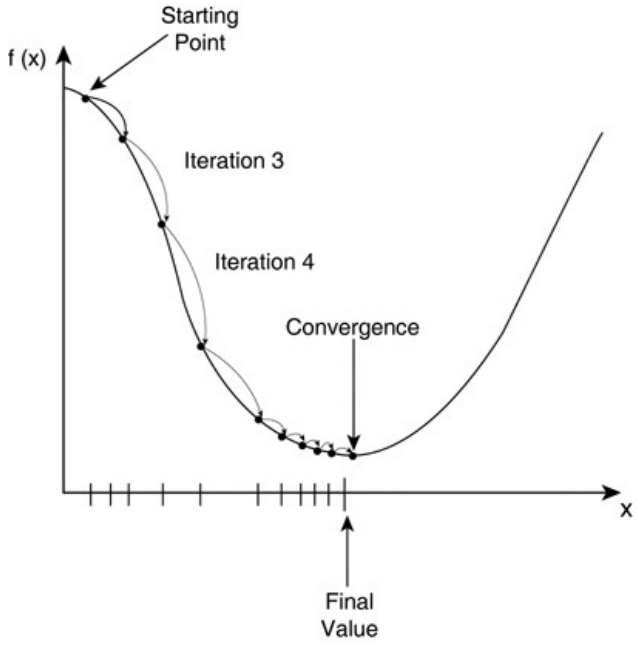

Figure 2.2: Gradient descent in several steps using a two-dimensional function Source: [McD17]

Parameter Updating
Each trainable parameter Φ, i.e. the parameters $W^{[l]}$ and $b^{[l]}$ of each layer, is generally updated in the following way:

$$\Phi = \Phi - \alpha \cdot \frac{\partial J}{\partial \Phi} \tag{2.10}$$

Here α is the so-called *learning rate* and $\frac{\partial J}{\partial \Phi}$ is the partial derivative of the cost function in respect to Φ.

Performing Training

In the actual training process all of the collected training data is often processed multiple times in the network to drive the learning. It is also referred to as an *epoch* when the network has processed the entire data set once. For example, one could have $10,000$ training images trained in 20 epochs. It is self-explanatory that this requires as much computing power as possible.

In order to reduce the computation time, several samples are often combined in so-called *minibatches* or simply *batches*. The calculated costs are then averaged for all samples of the minibatch.

Depending on the application, a few thousand samples as a training set can already be sufficient to achieve results, but it can also be significantly more. One of the biggest problems in Deep Learning is the compilation of the labeled training data. This can often only be done by hand and thus takes a corresponding amount of time. However, there are some standard datasets that are freely available for experiments, exercises etc., such as the MNIST dataset, which contains $70,000$ images of handwritten numbers from 0 to 9 [LCB].

2.1.4 Evaluation

Once a network has been trained on sufficient data, the accuracy and performance of the network must then be evaluated. For this purpose additional data is used, which was not part of the training data set. This so-called *test set* has therefore never been "seen" by the network before. The performance of the network can be determined on the basis of the predictions on the test data, which are also provided with labels. If noticeably more than half of the test data is classified correctly, it is proven that the network does not only produce random output, but has learned correlations in the input data.

Various metrics can be used to evaluate a network. First, there are basically four types of results how to classify each test sample:

- correct positives: the number of correctly detected positive samples.
- correct negatives: the number of correctly recognized negative samples.
- false positives: the number of samples falsely detected as positive.
- false negatives: the number of samples falsely recognized as negative

From this, we can derive the following ratios, among others:

- Accuracy: Ratio of correctly positive and correctly negative samples to the total number of samples in the test data set.
- Precision: Ratio of correctly positive detected samples to the total number of all samples detected as positive (correct and false positives).
- Recall: Ratio of correctly positive detected samples to the total number of positive samples in the test data set.
- F1 score: $F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$

Hyperparameters

In order to optimize the network for the corresponding application, it is necessary to adjust the so-called hyperparameters. Hyperparameters are all variable values of the network architecture and the training process that are not optimized by the training itself. There are different hyperparameters depending on the architecture of the network and the algorithms used.

Some examples of common hyperparameters are

- the activation functions $g^{[l]}$ in each layer.
- the number of neurons per layer $n^{[l]}$
- the number of layers L in total
- the number of training epochs
- the size of minibatches
- the calculation of costs, i.e. the cost function J
- the learning rate α

Improving the hyperparameters so that the performance of the network is optimized is called hyperparameter tuning. This can be done manually in the simplest way, by training and comparing the network several times with different configurations. However, since this requires a lot of time and structure, it is useful to find an automated solution that tries different hyperparameters and chooses the next set of parameters based on the results.

2.1.5 Format of Input Data

In subsections *Labeling of Training Data* (2.1.3) as well as *Performing Training* (2.1.3) it has already been discussed that one of the most important requirements for the successful training of a NN is an appropriately labeled and extensive data set. However, another important property of this dataset is that all samples are in the same format or dimensions. So, in the example of an image dataset, all images must have the same dimension and should use the same encoding for their pixel values, e.g. RGB color values between 0.0 and 1.0 for each pixel. This is due to the static architecture of a NN, in which the first layer expects a fixed number of parameters or a fixed length of the input vector X.

For any future use of the network, it must likewise be ensured that the format of the data matches the format of the training data in order to obtain plausible outputs. This may need to be ensured by appropriate conversion measures.

While an end user of a trained network only has to care about the correct format of the input data to get the corresponding output, the underlying structure of the NN with its various layers can remain hidden to him. In this way, NNs can also be used as an Application Programe Interface (API) that promises a corresponding output if the input is correct.

2.1.6 Preprocessing of Training Data

A step in Artificial Intelligence (AI) projects that should not be underestimated is the preprocessing of input data. It often makes sense not to train a NN with raw data, but to process it in advance in the same way. There are various methods, depending in particular on the use case and the data format, to preprocess training data. Since every input into a network, independent of the actual data format, always ends in an input vector X of floating point numbers, it is often useful to normalize it. Likewise, it must be decided how to encode the input data into such a vector.

Example of an Input Vector for Images
For images, one often chooses a square format, e.g. 256×256 pixels. Thus, when dealing with RGB color images, the dimensions of the images are $256 \times 256 \times 3$. Since this three-dimensional matrix cannot serve as input for the network, it is flattened into the one-dimensional vector X by writing all columns one after the other for the three color values. This input vector X of length $n_x = 256 \times 256 \times 3 = 196,608$ then serves as input for the network. Accordingly, each neuron of the first layer of the network has $n_x = 196,608$ weights w.

2.1.7 Summary and Outlook

An introduction to the topic of deep learning and Neural Networks was given and the mathematical foundations of a simple network were derived and explained. An attempt was also made to place the theory into an overall picture. By appropriately explaining the preprocessing, the training and evaluation process as well as the outputs and their interpretation by means of some examples, a basic end-to-end use of a NN was circumscribed.

Thereby it was limited to the most necessary basics. Below is a list of some more advanced topics that were not covered in this introduction:

- more complex network architectures, such as Convolutional Neural Networks (CNNs) for image analysis, Recurrent Neural Networks (RNNs) as well as Long Short Term Memorys (LSTMs) for time-critical data or data histories, Generative Adversarial Networks (GANs) etc.
- mathematical optimization and regularization methods, such as. “Batch Normalization”, “Weight Initiliazation”, “L2-Regularization”, “Dropout”, “Adam Optimization” etc.
- different approaches and exceptions to the explained proceedings, such as unlabed training or various activation functions, like $\tanh$ or ReLU

2.2 Speech Interfaces

written by Florian Sommerfeld

Speech is regarded as the primary means of human communication [Jou08]. It is natural for humans and thus can be used for Human Machine Interaction without the need of a special training. Furthermore a speech interface provides several advantages over traditional interfaces, such as the improvement of multitasking by leaving hands and eyes free for use. Current Automatic Speech Recognition systems have shown that they are applicable to various HMI tasks. It is already being used successfully in numerous fields such as Smart Home [PWS18]. ASR seeks to create a system capable of converting audio signals into a linear sequence of words.

2.2.1 Keyword Detection

Speech interface systems most commonly make use of the Wake-Up-Word concept. Amazons Alexa for example starts streaming user speech to the Alexa Voice Service when the wake word engine has detected the wake word "Alexa" [wake-word-verification]. The stream is closed as soon as an intent has been identified or the user stopped speaking. The WUW speech recognition paradigm was first proposed as a method to explicitly request the attention of a computer using a spoken word or phrase by Këpuska V in 2009 [Kep11]. The method consists of multiple components: Frontend, Voice Activity Detector, Backend and a Support Vector Machine Classifier. As illustrated:

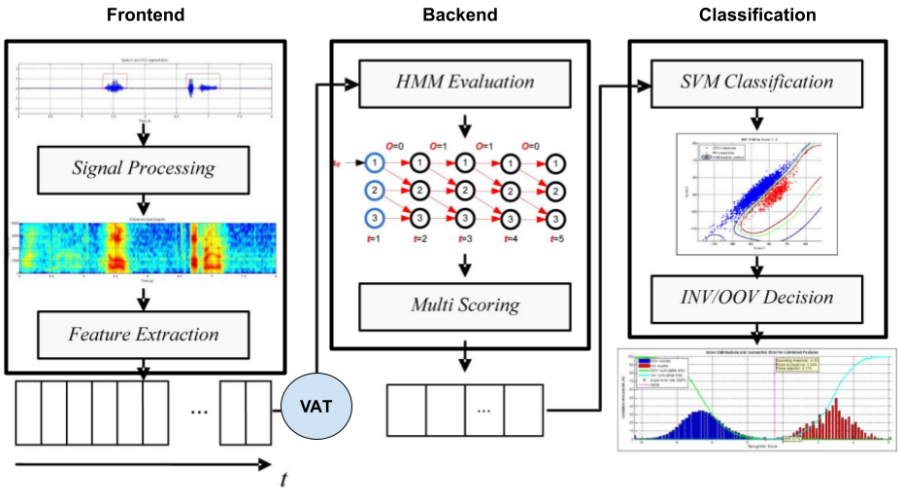

Figure 2.3: WUW system design Adapted from: [Kep11]

The frontends task is to extract features of the input signal using MFCCs and Linear Predictive Coding Coefficients (LPCs). The extracted features will be anaylzed for any voice activity and if the input frame is speech-like it will be send to the backend for a recognition procedure based on Hidden Markov Models. The models are trained to recognize a specific word as the WUW. When the WUW was detected, the speech input will be forwarded to the classification component which classifies a word as In-Vocabulary or Out-of-Vocabulary using Support Vector Machines.

In comparison to the work of Këpuska V. we are using a neural network as our backend, so hereinafter we focus on the fundamantals of feature extraction, more specifically the MFCCs analysis since studies have shown that it has the highest performance rate and lowest complexity [McL09].

Feature Extraction

Feature extraction is the process of determining a value or vector that may be utilized as an object or individual identifier in the classification process [Aul19]. MFCC analysis is the standard method for

feature extraction in ASR [Mot02]. The anylsis is based on possible frequency differences detected by the human ear and commonly shown on a Mel (derived from Melody) scale. The following ilustration shows the feature extraction process with MFCC:

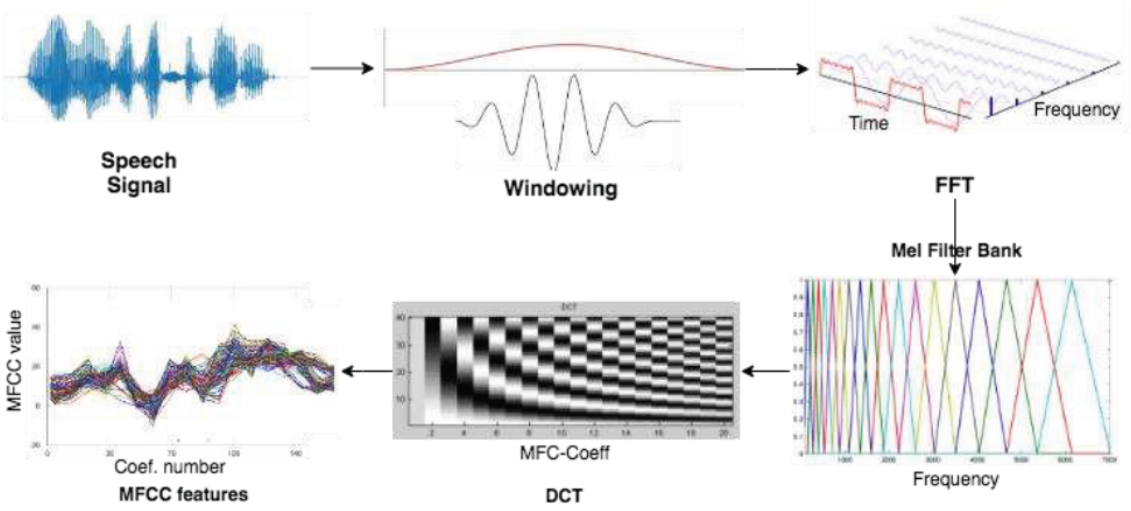

Figure 2.4: MFCC analysis block diagram Source: [Aul19]

At first a pre-emphasis is applied on the analog signal which emphasizes high frequencies to remove noise. The pre-emphasis filter increases high frequencies and reduces low frequencies of the signal:

$$y(t) = x(t) - \alpha\, x(t-1) \qquad \text{where filter coefficient } \alpha \in [0.9, 1.0] \tag{2.11}$$

The filter output will then be split into short time frames to deal with common problems such as aliasing, spectral leakage, and discontinuity. After splitting the frames, a Hamming window will be applied to the signal before it is passed to the FFT. The N variable of the window function equals the samples of a frame:

$$w(n) = 0.54 - 0.46\cos(\tfrac{2\pi n}{N-1}) \qquad \text{where } 0 \leq n \leq (N-1) \tag{2.12}$$

The windowed signal is transformed from the discrete time domain to the frequency by taking the discrete Fourier transform (DFT) of that sequence using an FFT algorithm for efficiency [GD15]. In our case x_n equals the nth value of the windowed signal in the following mathematical definition of a DFT [JC14]:

$$X_k = \textstyle\sum_{n=0}^{N-1} x_n\, \mathrm{e}^{-\frac{j2\pi nk}{N}} \qquad k = 0, \ldots, N-1 \tag{2.13}$$

The result of the FFT is a spectrum which then will be mapped to the Mel scale by applying a triangular function [JC14] to compute Mel filter banks:

$$Mel(f) = 2595\log_{10}(1 + \tfrac{f}{700}) \tag{2.14}$$

After computing the Mel filter banks, a base 10 logarithm is applied:

$$X_i = \log_{10}(\textstyle\sum_{k=0}^{N-1} |X(k)|H_i(k)) \tag{2.15}$$

"X_i is the value of frequency spectrum to i, N is the number of coefficients of FFT, and $H_i(f)$ is the filter value to i on the frequency spot f." [Aul19]

Finally the Mel cepstrum (power spectrum of the logarithmic filter bank) is converted using a Discrete Cosine Transform to decorrelate the log filterbank. The following DCT formula is used to compute the MFCCs [Aul19]:

$$C_n = \sum_{i=1}^{M}(\log(X_i))[n(M-\tfrac{1}{2})\tfrac{\pi}{M}] \tag{2.16}$$

2.3 Audio Formats

written by Jan Hartig

The used SoC has a Micro Electro-Mechanical System (MEMS) microphone onboard [PZ1]. This kind of microphone is used in most modern devices because of its small size, low cost and low power usage [pdm]. In case of the PYNQ-Z1 board, the microphone provides Pulse Density Modulation (PDM) encoded data. For further processing, PDM encoded voice data has to be converted to Pulse Code Modulation (PCM) encoding.

2.3.1 PDM

PDM encoded (audio) data is a stream of single bits at a comparably high sample rate (often around one to three Mhz). The value of a single bit is determined by a process called delta-sigma modulation as seen in 2.5. This circuit consists of an integrator and a feedback loop of the last PDM value to be added to the analog input. The maximum frequency is a stream of only '1's and the lowest only '0's. Because of the high sample-rate (f_s, rate at which a signal is sampled to produce a discrete-time representation) the bandwidth (maximal frequency that can be sampled (following rules of Nyquist and Shannon [nyquist-shannon49])) is very high for most use cases ($f_s/2$). To deal with PDM data, it has to be decimated (removing unnecessary frequency levels with digital filtering). An example (PDM encoded) sine wave can be seen in 2.6. [digital-audio12] [pdm]

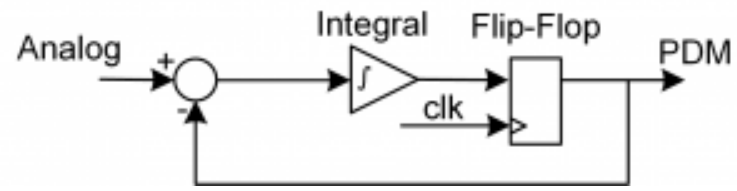

Figure 2.5: delta-sigma modulator Source: digilent.com [PZ1]

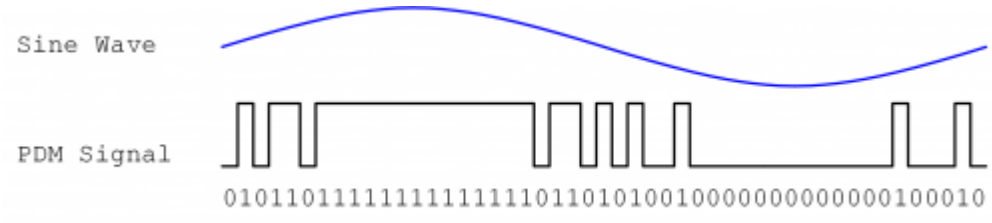

Figure 2.6: PDM signal modulation Source: digilent.com [PZ1]

2.3.2 PCM

PCM data is a series of samples with a fixed wordlength. The bandwidth characteristic is the same as in the PDM modulation. Oppose to PDM data, a lower sample-rate is used (usually 48 kHz). The quality

of the signal is determined by the wordlength (N). A higher wordlength z means more quantization levels ($n = 2^z$). The more quantization levels there are, the smaller is the difference in (voltage) frequency that can be distinguished. The result of this is a lower quantization noise (error of real signal and output) which leads to a better representation of the actual signal. To evaluate the quality of a signal the signal to noise ratio (SNR) is used ($SNR = (6.02N + 1.76)dB$). [digital-audio12]

2.4 Fast Fourier Transformation

written by Jan Hartig

Algorithm to (efficiently) calculate the DFT. A DFT is a (limited) timeframe of a signal decomposed into discrete frequencies components. There are different implementations (see figure 2.7). Most commonly the Cooley-Tukey algorithm is used [CoolTuk65]. Parameters for the FFT have to be set according to the incomming signal and the desired output singal quality. The incomming signal (left side of 2.7) has to be sampled with a sample-rate (f_s) and a chunklength (CL, power of 2) for the DFT decomposition has to be adapted to the application case. The result of the transformation is a spectrum of all frequencies the given chunk consists (right side of 2.7). [FFT]

Signal paramters

- bandwidth $f_n = f_s/2$
- timeframe (length of chunk) D $= CL/f_s$
- frequency resolution (minimal frequency step) $d_f = f_s/CL$

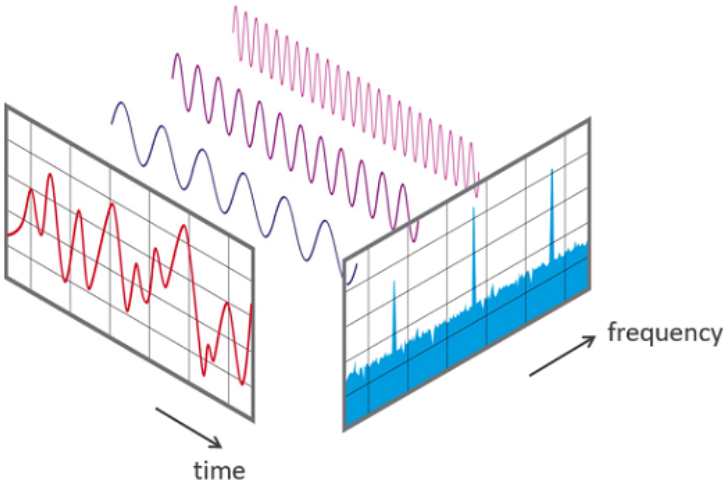

Figure 2.7: fast fourier transform picturized Source: nti-audio.com [FFT]

For the FPGA used in this project, XILINX is providing a IP-Core with different alogrithms to use for (chip) size and speed requirements (view the documentation [xFFt21] for more detailed information). In most cases the Radix-N algorithm is used to calculate the FFT. Compared to the Cooley-Tukey algorithm the number of multiplications needed is reduced significantly (Radix-4, 25% less) [xFFt21].

2.5 FPGA

written by Philipp Wittje

An FPGA is a digital component that can be used to realize different circuits by means of different programming. This is also the main advantage of FPGAs over ASICs, which are chips that are manufactured for a special case. An FPGA can be deleted and reprogrammed again and again, which makes them very flexible in their application. Especially for smaller quantities, they are also cheaper in production than ASICs, although usually with a smaller number of gates, so it tends to be lower clock frequencies than conventional ASICs.

In detail, an FPGA consist of a large number of logic elements, often these are flipflops with logic circuits stored in front of them and look-up table (LUT) with which the logic functions are realized. A LUT can realize any combinatorial function (NAND, XOR, AND, multiplexer, etc.) from the input signals. The number of input signals per LUT depends on the FPGA. For functions that require more inputs than a single LUT has (high fan-in), several LUTs are connected directly to each other. The flipflops are used to cache signal values so that they can be processed in the following clockcycle. As universally applicable digital integrated circuits (ICs), FPGAs support a variety of signal standards in order to be able to communicate with the different digital components on the market. The I/O behaviour is set in a file along with many other parameters. [mik21]

2.5.1 IP-Cores

For frequently used circuits, it is advisable to combine certain functions already pre-made into reusable units, analogous to libraries in programming languages. These prefabricated design blocks are called IP-Cores. The main advantage of IP-Cores is the saving of development time by using prefabricated blocks and reusing self-developed blocks. This advantage is particularly noticeable for large and complex circuits. [Pog03]

2.5.2 AXI-Stream

The Advanced eXtensible Interface Bus (AXI)-Stream is an interface for connecting peripheral components. Usage data is also transmitted via this bus, while configuration data is transmitted via other interfaces, such as AXI or AXI-Lite. Unlike AXI or AXI-Lite, the AXI-Stream does not use address lines to address the target. Differences are still made between the direct transfer of data from the main memory to the IP cores and the transfer of data from the local memory. [kam22]

Chapter 3

Design of System Architecture

written by Hagen Stöver

coauthored by Jan Hartig

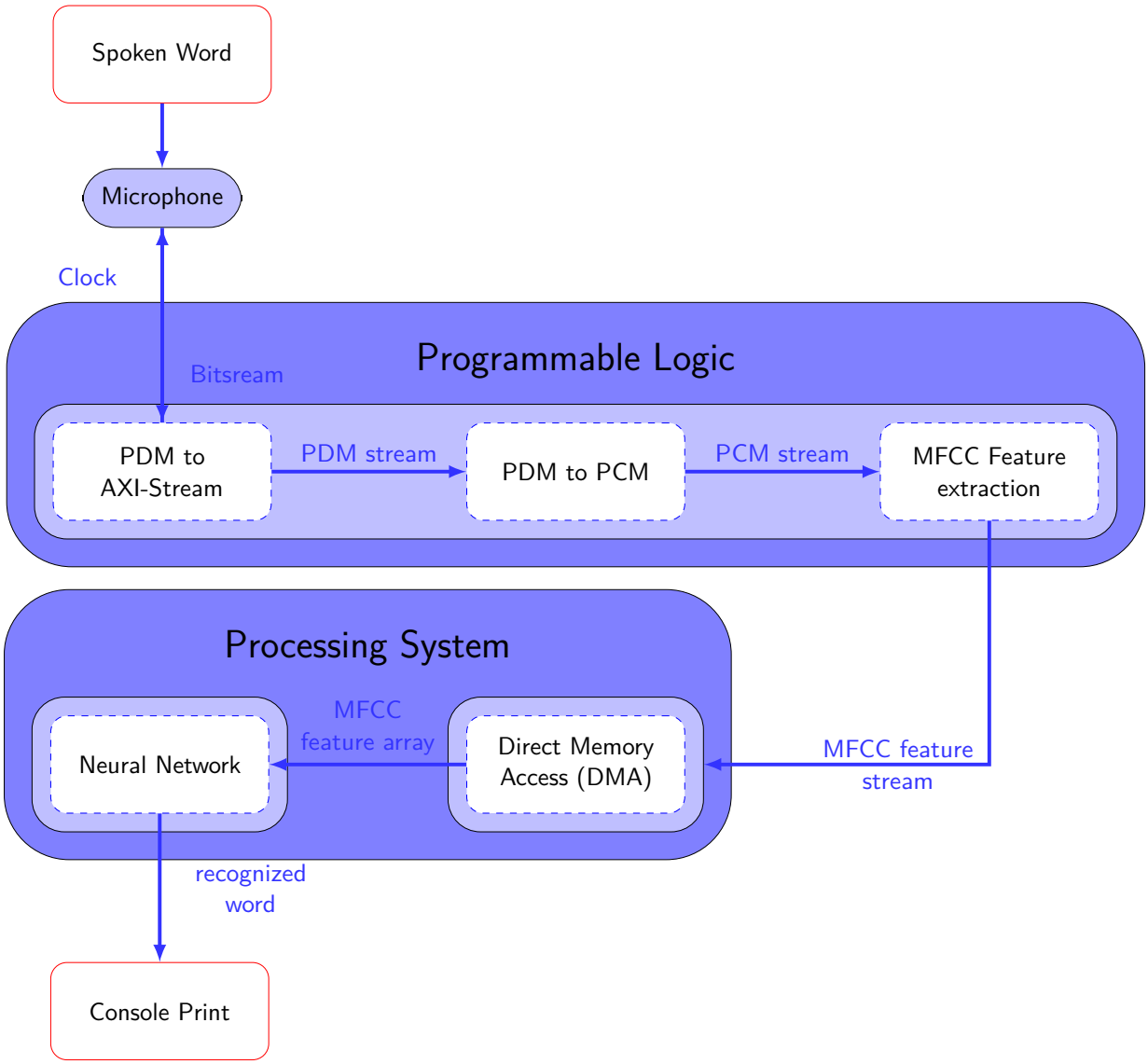

Figure 3.1: System architecture as block diagram

The Architecture of this project consists of two main blocks, the PL (FPGA part of ZYNQ SoC) part with IP-Cores to fetch microphone input and convert it to MFCC-Features. The Processing System (PS) (Central Processing Unit (CPU) with different interfaces and controller, see reference [17] for details) is

running the NN with a Python program. To transport data from PS to PL a DMA is used. [17, p.3]

At the beginning, the microphone on the PYNQ-Board will record audio and send it to the PL-block. There the audio will be processed through various steps. After this, the result will be send to the PS and the NN where the audio-data will be analyzed. Finally, the NN will output a recognized command [1] in the console.

3.1 Interfaces Between System Components

The many components and parts of the system work with different representations of data. At the beginning, the data is transmitted as words through sound waves. Those waves are being picked up by the microphone installed on the PYNQ board. Inside the microphone the data is converted into a bitstream and send to the PL-block. Before this block can work with the data, it first has to be transformed, because the bitstream send from the microphone is encoded with PDM, however, the PCM encoding is needed to work with the data.

After the Audio Processing the component on the FPGA will send a stream of numbers to a Python program running on a SoC. There, the stream of bits will be segmented into an array with a fixed data type.

This array will then be read by the NN, that finally prints a recognized command into a console.

[1] left, right etc.

Chapter 4

Used Materials

4.1 Hardware

written by Roland Helmich

The target platform for the project is a PYNQ-Z1 development board, featuring a ZYNQ XC7Z020-1CLG400C SoC. Notable features of the SoC are that it integrates two ARM Cortex-A9 processor cores with an Artix-7 FPGA. The board contains 512 Megabytes of random access memory (RAM) which can be used by the SoC. A Knowles SPK0833LM4H-B PDM microphone is integrated into the development board. It is used to record the audio samples. [17]
A standard class 10 8 GiB microSD card is utilized as the boot medium for the PYNQ-Z1 board. To connect the PYNQ-Z1 to power and network, the included power brick and network cable are utilized.

4.2 Operating System and Development Environment

written by Roland Helmich

Xilinx provides a pre-built operating system image for the PYNQ-Z1 board. Version 2.7 of the image is used for this project. The PYNQ image provides Python 3.8.2. A development environment for Python, Jupyter Notebook, is also pre-installed and configured. [21]
To develop FPGA code, the Vivado ML Design Suite 2021.2 is used.

4.3 Software

written by Fynn Hagen

For the development of the neural network the Python library Brevitas, a PyTorch research library which provides support for quantization-aware training of neural networks, was used. Brevitas offers an interface to the FINN-framework [Pap21]. The framework is capable of compiling a quantized NN into a format that can be deployed on a FPGA. The FINN-framework itself was not used in this project scope. But in order to provide a NN for further future development which then in the next step is able to be deployed on a PYNQ-FPGA, all relevant interfaces to the FINN-Framework were needed to be respected.
The neural network was trained using multiple audio files containing the commands to control the vehicle. To make the net more robust against variations in the audio signal, the original audio samples used for training were post processed using the Python library Numpy and PyDub. Both of these offer an easy support for .mp3 files which are used as data format for the audio samples. Additionally a few other Python packages are used as well. An overview over all used packages can be found in appendix A.

Chapter 5

Data Processing on FPGA

In this chapter, all preprocessing steps (every step before the DMA controller of 3.1) will be elaborated. The sections are orientated to the flow of audio data through the system.

5.1 Recording PDM Samples

written by Roland Helmich

In order to analyze audio samples, they first need to be recorded. The development board used in this project features an onboard microphone that will be used for this. The microphone (Knowles SPK0833LM4H-B) is a PDM type microphone. It has two connections, a clock line and a data line [mic]. Both are connected directly to the Actix-7 FPGA of the SoC. The microphone can operate in HIGH and LOW mode, the difference being when valid data is made available on the data line [mic]. On a PYNQ-Z1 board, the LOW mode is always selected [PZ1, p. 17]. This means valid data is present at a rising edge.

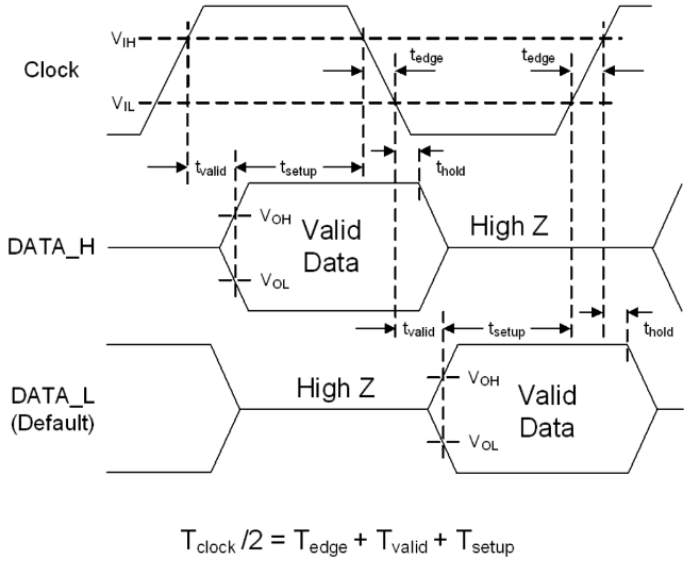

Figure 5.1: SPK0833LM4H-B Timing Diagram Source: [mic]

The clock signal for the microphone has to be generated by the FPGA. Supported clock rates for the microphone lay between 1 and 3.25 MHz [mic]. Because of the way PDM microphones work, the clock rate is also the sample rate. The last sampled value can be read directly from the data line, since the PDM format supports only binary values anyways.

The entire processing pipeline on the FPGA is connected together via AXI-Stream interfaces. Since AXI-Stream interfaces have a minimum width of at least a byte [stream-spec10], values read from the microphone need to be collected into small packages before they can be send to be processed further.

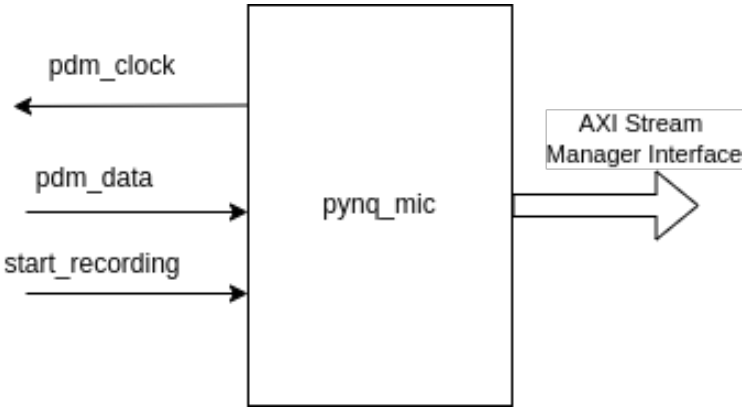

Figure 5.2: Interface of pynq_mic IP-Cores

To manage clock generation, data acquisition and output to an AXI-Stream, a prototype for a custom IP-Cores named *pynq_mic* was implemented. The core includes a clock generator that can generate a configurable PDM clock based on the AXI bus clock. It also acts as a AXI-Stream manager [1]. Data is sampled from the microphone whenever the clock generator produces a rising edge. If enough samples to fill a single AXI-Stream transaction have been collected, they are output on the AXI-Stream manager interface. After a certain amount of packages have been sent on the AXI interface, *tlast* will be asserted by the core, signaling downstream subordinates that a complete frame has been send. This behavior is needed because many AXI-Stream IPs do expect to receive *tlast* at some point. After how many transaction *tlast* is asserted can be configured as well. To reiterate, a single AXI-Stream transaction consists of multiple individual sample values (how many depends on the configured bus with), and a single frame is made up of multiple AXI transactions. The *pynq_mic* core will start to sample data after the *start_recording* signal was high. It will stop recording after a frame has been finished if *start_recording* is not high.

The design of the core is deliberately kept relatively simple, especially compared to the audio core included in the PYNQ baseoverlay, to allow quick changes to the behaviour and interface of the core.

5.2 Converting PDM to PCM

written by Roland Helmich

coauthored by Jan Hartig

The Xilinx *xfft* IP-Core that will be used later in the audio processing pipeline supports input values between 8 and 34 bits in length [xFFt21]. In order to effectively use the single bit samples acquired from the microphone, it is thus necessary to convert them from the PDM format into the PCM format. To achieve this conversion, the PDM signal from the microphone should first be low pass filtered. After filtering, the signal then needs to be decimated until a reasonable sampling frequency (for example 48 KHz) is reached.

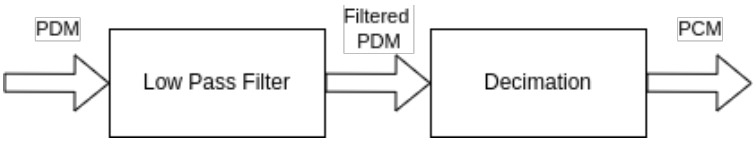

Figure 5.3: Conversion from PDM to PCM

[1] Note: older version of the AXI specification use *master* and *slave* terminology instead of *manager* and *subordinate*

5.2.1 Low Pass Filtering

Low pass filtering of the input signal is needed because PDM samples are acquired at a very high sampling rate. In our case, at least 1 MHz and potentially up to 3.25 MHz. At such high sampling rates, unwanted high frequency noises will be present in the signal. If the sampling rate is lowered, these high frequency noises can lead to aliasing [nyquist-shannon49]. Also, only frequencies in the audible spectrum (generally between 20 Hz and 20 KHz [Ros07]) are relevant for analysis of spoken words. Thus, by low pass filtering before further processing the signal, those unwanted high frequency noises can be safely removed. A filter that accepts frequencies up to 48 KHz should be a good choice for this particular processing step, as it - considering the sampling theorem by Nyquist [nyquist-shannon49] - would cover the entire audible range of humans with a small bit of headroom.

5.2.2 Decimation

After low pass filtering, the signal still has its original sampling rate in the megahertz range. In order to reduce the sampling rate, the signal is now decimated by a constant factor - the decimation factor. To calculate a decimation factor M, the following simple equation can be used:

$$f_{target} = \frac{f_{original}}{M}$$

The decimation by an integer factor n effectively means that only every n-th sample will be kept. All other samples will be discarded. For an input signal with a sampling rate of 3.072 MHz, a decimation factor of exactly 64 would result in a 48 KHz output signal.

5.2.3 Conversion on the FPGA

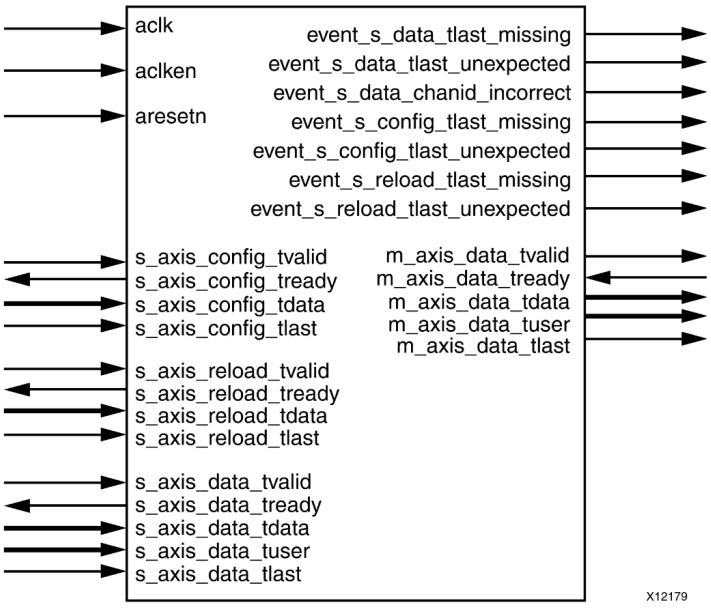

Figure 5.4: finite impulse response (FIR) Compiler Core Ports Source: [Xil21]

For digital signal processing, Xilinx provides the *FIR Compiler* IP-Cores. With this IP-Cores, different types of FIR filters can be implemented on the FPGA [Xil21]. The filter type *polyphase decimator* of this core executes both of the previously described steps. It behaves similarly to MATLABs *FIRDecimator* [Xil21] [FIR].

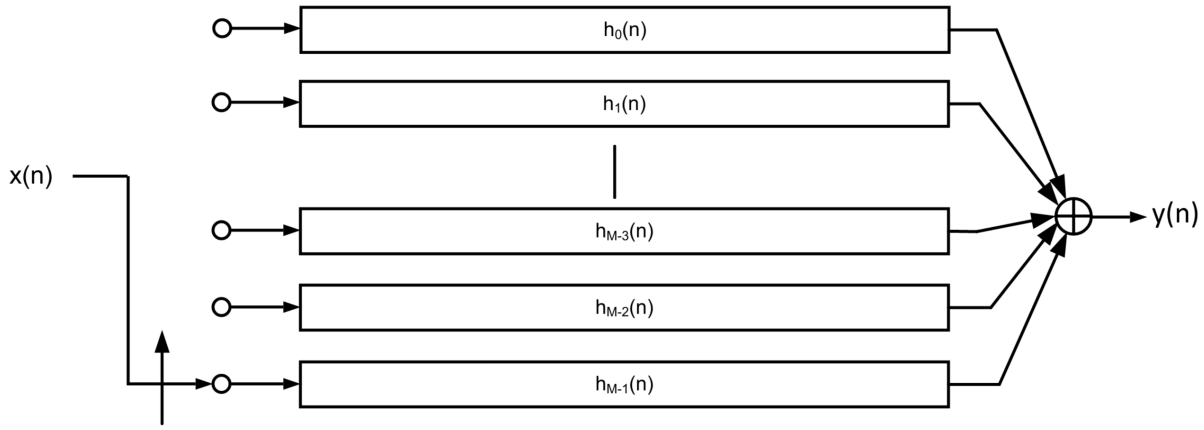

Figure 5.5: M-to-1 Polyphase Decimator Source: [Xil21]

Sample data is accepted by the core via a AXI-Stream subordinate interface and the filtered data is output via a AXI-Stream manager interface [Xil21]. It is thus easy to integrate the core into the pipeline. Additionally the core also has a configuration channel and a reload channel (both AXI subordinates). These two channels are used to load and select filter coefficients for the FIR filter. Filter coefficients can be designed with the MATLAB filter designer [FIRD] and exported in a Xilinx compatible format.

5.3 MFCC Feature Extraction

written by Florian Sommerfeld

coauthored by Jan Hartig

As previously mentioned, see 2.2.1, MFCC is based on the human sound perception. Common steps to extract the features using the MFCC algorithm are (based on 2.4):

- pre-emphasis
- framing
- windowing
- FFT
- Mel filter banks
- DCT

For our use case we would serve the PCM speech signal as the input for the MFCC algorithm. At the beginning of processing a time discrete signal is already preset. The sample-rate and wordlength (number of bits used to represent a sample) for the FFT should be adopted from the previous decimation 5.2.2.

To implement a MFCC core on the FPGA two main researches have been taken into consideration: [mfcc-core] and [Dao+17]. For the pre-emphasis filter the typical filter coefficient is $\alpha \in [0.95, 0.97]$. Using $\alpha = 1 - \frac{1}{32} = 0.96875$ which is in the range, this can be implemented with a shift register [mfcc-core]. The filtered signal would then have to be segmented into small frames. Therefore [Dao+17] uses a length within the range of 20 to $40ms$ and divides the signal into 256 frames. For the hamming windowing $w(n) = 0.54 - 0.46\cos(\frac{2\pi n}{N-1})$ would have to be implemented with $0 \leq n \leq (N-1)$. Next the windowed frame would go through the FFT (with parameters from the decimation). An approach would be to load data samples into the memory, then apply the FFT and finally samples can be read. [mfcc-core] uses a Decimation-in-time (DIT) Radix-2 FFT algorithm block which rearranges the DFT

equation into two parts for achieving a faster computation [GEO19]. The power spectrum is then being applied to a Mel filter bank with 32 triangular filters. Afterwards the logarithm is computed based on the algorithm presented in [Tur10]. Finally a Type II DCT is used which uses less arithmetic operations than conventional algorithms without sacrificing numerical accuracy [SJ08] to compute the MFCC features.

5.4 Provide converted Data for further processing

written by Hagen Stöver

The data provided by the mel-frequency cepstrum (MFC) will be put onto an AXI-Stream. The consumer of the data will be a DMA coontroller. Through a DMA controller the system is able to directly save the data from the MFC in the RAM of the SoC without the CPU needing to be involved. This will relieve the CPU, since it does not need to be used to transmit the data and in total the transportation of data through a DMA-Controller should be faster than through a CPU.

To reduce complexity and potential timing problems, the MFC will always override the output data, whether the data was read or not. The data itself is a stream of MFCCs.

Chapter 6

Data Interface Between FPGA and Neural Network

written by Hagen Stöver

The FPGA will send a bitstream of data directly to the Python program. To do so, a DMA controller will be used. The address of the RAM will be provided by Vivado. The Python program will access the data at that address and save it into a Numpy buffer-array. The `Xlnk` class from the `pynq` package will do all the work in the background. The only setup needed is the specification of the data type, the size of the buffer and the loading of an overlay, where a DMA-Controller is configured.

The example below, shows how the DMA-Interface can be loaded and used to access data from a DMA-Controller. The buffer will contain 5 integer numbers.

Listing 6.1: Accessing data from a DMA-Controller [DMA-PYNQ]

```
1 overlay = Overlay('example.bit')
2 dma = overlay.axi_dma
3
4 output_buffer = xlnk.cma_array(shape=(5,), dtype=np.uint32)
5 dma.recvchannel.transfer(output_buffer)
6 dma.recvchannel.wait()
7 print(output_buffer)
```

As soon as the specified buffer is filled, the Python program then gives the array to the NN. The length of a number, sent by the FPGA component, should be a multiple of 8 bits. If the default length of a coefficient from the MFC is for example 12 bits long, than the number should be padded with 4 bits. In Python, the data type of the numbers also have to match the provided coefficient from the MFC. If a coefficient is 4 Byte long, than the data type could be, for example, `uint32`.

Since the audio sample is split into smaller segments, the program also has to read the result multiple times.

Chapter 7

Machine Learning

7.1 Generation of Test Data

7.1.1 Source of Raw Audio Data

written by Dylan-Noah Schade

The raw audios used for generating the training and test data were recorded by the authors of this document using mobile devices or headsets. To mimic the rather low quality of the onboard microphone the use of high-end recording equipment was omitted. The background noise of the radio-controlled car (rc car) was also recorded using a smartphone driving along with the car.

To have a mixture of different voices and pronouncements everyone recorded each trigger word several times, each time saying the word a bit different. Afterwards the audios were manually processed in a Digital Audio Workstation (DAW). In particular each audio was trimmed, so that it contained only the word. Also, the audios were all normalized to have more or less the same volume. Afterwards they were all exported in the same format and with the same sample rate. In the end there were about 400 audios containing one of the trigger words each and another 75 containing other spoken words.

7.1.2 Data Augmentation

written by Fynn Hagen

Sample Processing

In order to enhance the variety of training samples to train a NN different processors are applied on each sample. Each of these processors can be added independently onto each training sample. The processors either add another background to the sample or vary the volume or speed of the sample. The processors that can be used are the following:

Adjust Playback Speed Processor

This processor speeds up or slows down the given sample. In order to adjust the playback speed of the sample a simple pitch control is used. In multiple tests a range of 25% slowing down and 25% speeding up were considered to be a good value range for this. Due to the physics of audio pitch control, an increase in the playback speed results in a pitch shift to the upper range, leading to the sample sounding pitched up. The opposite effect can be seen, when slowing down the sample. This results in a lowered pitch for the sample.

Adjust Volume Processor

This processor adjusts the volume of the given sample. A range of changing the volume between 60% to 110% of the original volume has been identified to be a good starting point. Especially the value for the increase of the volume should be handled with care. Values greater then 115% tend to lead to a distortion of the audio signal.

Distortion Processor

The distortion processor is technically the same as the volume processor. The only difference is that the volume is increased up to 135% of the original sample volume. This leads to a distortion effect on the sample.

Background Noise Processor

This processor is used to imitate driving noises of the car that should be controlled. Therefore it adds background noise to a given sample. The background sample to add is applied with a 50% reduced volume. The sample to add is a sound recording of a driving rc car, which might not be accurate comparing to the real used vehicle. The used sample can be configured in the config file and can be extended to other backgrounds, i.e. white noise.

Background Voice Processor

This processor is used to mimic the case of people talking to each other besides the input of the original voice command. For this a random selection from all negatives (samples that shall not be recognized as any of the categories) is taken and added with a reduced volume at a random point of the original sample.

7.2 Neural Network

written by Mattheo Mahnke

A neural network was used to classify the recorded samples. The process to create this network is described in the following sections.

7.2.1 Neural Network Architecture

The used neural network has a three layer architecture as shown in Figure 7.1.

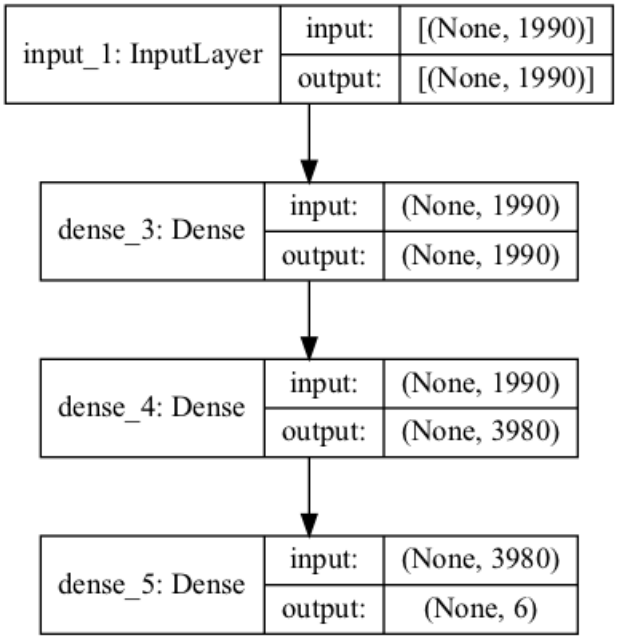

Figure 7.1: The used neural network topology

It uses three dense (fully-connected) layers, where the first two use ReLUs as activation functions. The last layer does not have an activation directly attached to it, since the cross entropy loss of PyTorch expects logits. The layers do not use a bias.
The input of the network are the MFCC which were computed from the audio input. The outputs are the logits for the classes the network shall classify (in its base version "forward", "back", "left", "right", "stop", and "none"). To get the probabilities for the classes from the logits one has to apply the softmax function.

To later be able to use the network on an FPGA this was converted to a QNN. QNNs do not work on floats or doubles but use small integers to speed up the computation in hardware, since no floating point units are used. The weights of the layers were limited to 3-bit integers, the ReLUs are 4 bits wide to prevent overflow.

This architecture was chosen because of its simplicity. The first iteration of the architecture was based on a RNN, but this was changed because of Brevitas not having RNN support in the the current version (see Appendix A for the used versions). This simple network does also take up less space than a more complex architecture while implementing it on the FPGA, which allows it to also be used for other applications. Since this structure showed good results during training (see 7.2.2), there was no need for changing it since then.

7.2.2 Training

The model was trained with the synthesised samples and their corresponding labels. The model was fit to the labels by minimizing the cross entropy loss of the network output. To minimize this loss the Adam optimizer with a learning rate of 0.02 and the defaults of PyTorch was used. The model was trained for 100 epochs, while halving the loss every 40 epochs, resulting in the following loss values:

- Epoch 0-39: 0.02
- Epoch 40-79: 0.01
- Epoch 80-100: 0.005

From the 401 input samples, which were split into a train and test dataset (see Section 7.2.3) 20000 samples were synthesized for training and test. Both for training and test a batch size of 512 was used

to limit the loss difference between batches by diversifying them.
For each epoch the batches were re-shuffled, to further limit big changes in the loss caused by the input. To speed up training and make the output more stable a batch normalisation layer was used after the last dense layer of the model. This normalises the output of the model based on the batch properties during training and during inference using a mean and variance which was fit on the training data during training. This layer is not quantized and because of that has to run on the microcontroller of the SoC. But since this runs on the output of the network, which is composed of 6 integers, this is not computationally expensive.

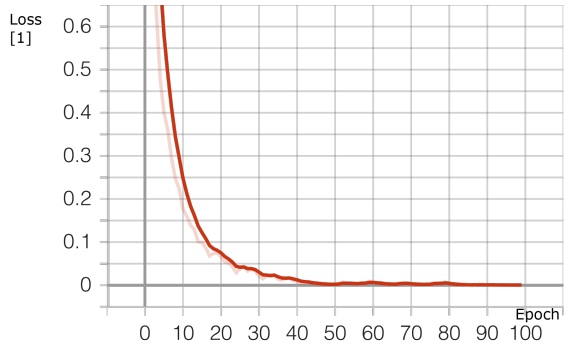

Figure 7.2: The loss of the model on the train set during training

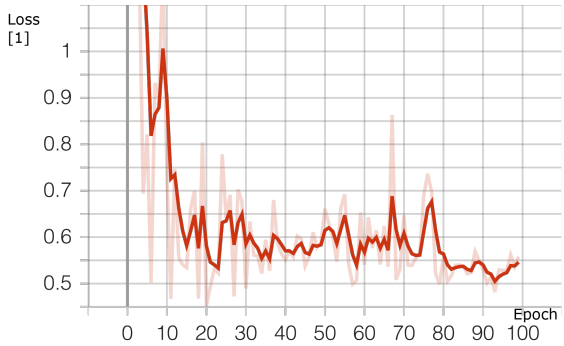

Figure 7.3: The loss of the model on the test set during training

Figures 7.3 and 7.2 plot the loss for the test and train set for each epoch. The light orange graphs show the real values, while the darker orange graphs are smoothed. One is able to see that the loss on the train set steadily decreases, while the loss on the test set also tends to decrease, but not as steadily as the train loss. This is because the model is only fit on the train set. See Section 7.2.3 on how this is used to select the model to use for inference. A training run of 100 Epochs, together with synthesizing the samples pre-training, takes approximately 45 minutes on an AMD Ryzen 9 5950X without a GPU.

7.2.3 Validation

To validate the performance of the model the raw input samples were split into train and test data.
A test/train split of 20% test data from the 401 samples, which were then synthesised to be 20000 samples, was used.
Train data is used to fit the model, while the test data is only used to measure accuracy, i.e. the model is never explicitly fit on the test data. This is used to detect overfitting of the model to the train data. If the test loss increases and accuracy decreases while the loss on the train data decreases, the model overfits to the train data. This would increase accuracy on the training data, but decrease accuracy during inference, which is where the model is used. This can be compared with memorizing results instead of learning how to compute them.
This simple architecture is able to produce an accuracy of about 90% on the used test set, meaning that 90% of the time the model correctly predicts the word, even after applying the sample processors to the base data.

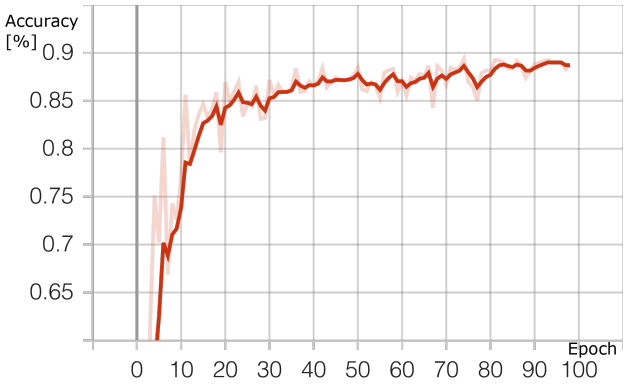

Figure 7.4: Accuracy of the model on the test set during training

The accuracy of the model on the test set was computed after each finished training epoch. Figure 7.4 plots the accuracy of the model on the y-axis with the epoch ID on the x-axis. The most accurate model was produced on epoch 93, where an accuracy of about 90% is reached. The graph in light orange shows the real values, the values plotted in dark orange have had jumps smoothed out.

The accuracy value is computed by getting the outputs of the model for the test set and counting the number of times where the prediction of the network matches the label. This is then divided by the number of elements in the test set. The loss of the model over the training set is also computed for each epoch (see Figure 7.3), but is not as relevant for selecting a model. While the loss and accuracy are correlated, having a slightly lower loss does not necessarily increase accuracy. The decreased loss could just be the symptom of increasing the confidence for some parts of the test set, which does decrease the difference between a theoretical 100% confidence for each sample and the output of the network. This does not imply that more labels produced by the network are now matching the true labels. This is why the accuracy was used to select the best model, which is an accurate representation of the models ability to predict the correct labels.

7.2.4 Inference

For inference the model with the highest validation accuracy (see Section 7.2.3) was used. When running on the PYNQ-Z1 board it is intended to use the FPGA for reading the data from the microphone and processing it to get the Mel Frequency Cepstral Coefficients. During testing and development a development machine was used to test inference. The repository contains code to be able to record samples with a builtin microphone and get the classification from the network. While the model has a 90% accuracy on the test set, the accuracy during inference is about 10%. This could be a symptom of 401 raw input samples not being enough input data, although they are altered through the data synthesis pipeline. The model is also very sensitive to using different microphones. Having more diversity in the input data should increase accuracy during inference.

Chapter 8

Evaluation

written by Roland Helmich

written by Fynn Hagen

coauthored by Jan Hartig

This chapter focuses on the evaluation of the results achieved within this project. It will be highlighted which aspects were successful/ approches were wrong to carry this knowledge into a possible follow-up project. In addition a review and evaluation the current project status/ findings will be done.

Due to the lack of final integration of the entire system, each component (neural network, recording and preprocessing of microphone data) are evaluated by themself, but no final system evaluation can be performed.

Processing Pipeline on Programmable Logic
Over the course of the project, a general concept for a processing pipeline on the Artix-7 FPGA has been developed. However, no real implementation on the target hardware could be tested due to time constraints. Generally, audio acquisition, filtering and conversion of the sampled data into the frequency domain should be handled by the FPGA - both because of the high performance requirements and the fact that the onboard microphone is connected only to the FPGA.

To handle audio acquisition, a prototype for a custom IP-Core (*pynq_mic*) was developed. This core is capable of controlling the onboard PDM microphone and transmitting recorded data via an AXI-Stream interface. During the early phases of the project, some tests were conducted with the Xilinx provided *audio_direct* core. However, the core proved to be limited to recording fixed sized samples to pre-allocated memory locations (instead of a continuous recording). It also only supports normal AXI transactions, not the stream type used by all other parts of the pipeline, making integration rather difficult.

To filter the recorded audio signal and convert it into the easier to process PCM format, the *FIR Compiler* IP-Core from the Xilinx library should be used. This core handles data input and output via AXI-Stream interfaces and is thus well suited to processing a continuous data stream. It provides configurable filtering and decimation capabilities. While the decimation factor can be easily calculated, finding correct filter parameters (coefficients) for the wanted behavior is more difficult. During research, it was discovered that the MATLAB filter designer [FIRD] can be used to create these parameters. The desired behavior for the *FIR Compiler* as part of the pipeline is that of a low-pass filter, followed by a decimation to the target PCM sampling frequency. Handling a single audio stream should be well within the capabilities of the core, this however could not be verified because the project did not progress far enough to test the actual design.

As the last step of the audio processing pipeline, we intend for MFCC feature extraction to take place on the FPGA. It has been proven by other research ([Dao+17]) that implementing feature extraction on a FPGA is possible using a custom IP-Core. Instead of implementing a custom IP-Core, it might also be

possible to chain together existing cores though AXI-Stream to handle the data processing. For example, Xilinx provides the *xfft* core for FFT processing - and the earlier mentioned *FIR Compiler* core for many different filtering operations. Because of the use of AXI-Stream interfaces in the pipeline, adding new processing steps with new IP-Cores should be relatively easy.

The results of the MFCC feature extraction would be committed to the main memory of the ZYNQ SoC. This can best be achieved by using the Xilinx provided *AXI DMA* IP-Core. It can receive data via an AXI-Stream subordinate interface, write it to a predefined memory location all while being relatively CPU efficient through the use of interrupts.

Processing Pipeline on Processing System

As seen as in 7.2.3, when testing the network with a random sample (generated the same way as test data 7.1 but not included in the sample set) an accuracy of $\sim 90\%$ can be achieved.

When testing the neural network in a simulated "real" environment, not on the PYNQ-Z1 board itself, but with the builtin microphone of a development machine the accuracy drops to $\sim 10\%$. This value is way to low for a real world deployment but could be improved in future projects by adding more variation to the training (microphone, voice, pitch, emphasis, etc.) sample base.

Chapter 9

Conclusion and Outlook

written by Jan Hartig

coauthored by Philipp Wittje

This project was focused on creating a simple speech regocnition for five commands. In order to achive this a preprocessing pipeline for (raw) audiodata, conversion to NN workable data and the NN itself is needed.

For now only the theoretical data processing (pipeline) is finished. The implementation has been started but is lacking. Audio data from the on-board microphone can be sampled with the created IP-Core *pynq_mic* (see 5.1). The needed filtering via FIR and decimation (5.2) is not yet implemented. Signal samples are part of a AXI-Stream which should make processing straight forward. Documentation of contemplated pipeline steps as far as it was possible to plan can be found in 5. Before planning more features these should be implemented and tested thourougly.

In the current scope, the NN should not yet be on the Programmable Logic part of the PYNQ-Z1 board. The needed transport for MFCC-Features through DMAs has been planned and prepared in form of python snippets (see 6).

The neural network for evaluating audio signals, is ready to interpret the audio signals accordingly. A generator of test audio data has been implemented. To make the network more robust and viable it should be trained with data from the onboard microphone. This depends on the previous implementation, making the completion of preprocessing a very high priority.

To implement this project in the context of an autonomous space craft with limited computational resources, the results of the NN have to be mapped to motor pins on the board, bringing together the entire system.

Appendix A

Used Material

A.1 Software Packages

written by Florian Sommerfeld

Resource type	Description	Usage	Version
Board	PYNQ-Z1	Platform for the embedded system	Z1
Computer	MacBook (macOS): M1 chip	Development of NN	2020
	PC (Win 10 Pro): AMD Ryzen 9 5950X	Training of NN	-
Boot Image	PNYQ-Z1 SD card image	PYNQ software environment	2.7
Flash memory	micro SDHC 8 GiB Class 10	Boot PYNQ-Z1	-
Power supply	12V, 3A power supply	Power source for the PYNQ-Z1	-
Cable	Ethernet CAT 5E	Communication with PYNQ-Z1	-
Development tool	PyCharm	Python IDE for NN development	2021.*
	Vivado	Development of FPGA code	2021.2
	Git	Version control	*
Programming language	Python	Development of NN	3.8
	VHDL	Hardware description of the system	VHDL-2008
Python package	numpy	Math	1.22.1
	pydub	Audio manipulation	0.25.1
	tomli	TOML configuration parser	2.0.0
	SQLAlchemy	Object Relational Mapping	1.4.31
	python-speech-features	MFCCs and filterbank energies	0.6
	scipy	Math	1.8.0
	torch	Machine learning	1.10.2
	tensorboard	Visualisation for machine learning	2.8.0
	brevitas	PyTorch research library for NN	0.7.1
	pyaudio	Python bindings for PortAudio	0.2.11
Audio sample	401 .mp3 files	NN training samples	-

Standard notations for Deep Learning

This document has the purpose of discussing a new standard for deep learning mathematical notations.

1 Neural Networks Notations.

General comments:

· superscript (i) will denote the i^{th} training example while superscript [l] will denote the l^{th} layer

Sizes:

·m : number of examples in the dataset

·n_x : input size

·n_y : output size (or number of classes)

·$n_h^{[l]}$: number of hidden units of the l^{th} layer

In a for loop, it is possible to denote $n_x = n_h^{[0]}$ and $n_y = n_h{}^{[\text{number of layers } +1]}$.

·L : number of layers in the network.

Objects:

·$X \in \mathbb{R}^{n_x \times m}$ is the input matrix

·$x^{(i)} \in \mathbb{R}^{n_x}$ is the i^{th}example represented as a column vector

·$Y \in \mathbb{R}^{n_y \times m}$ is the label matrix

·$y^{(i)} \in \mathbb{R}^{n_y}$ is the output label for the i^{th} example

·$W^{[l]} \in \mathbb{R}^{\text{number of units in next layer} \times \text{number of units in the previous layer}}$ is the weight matrix,superscript [l] indicates the layer

·$b^{[l]} \in \mathbb{R}^{\text{number of units in next layer}}$ is the bias vector in the l^{th} layer

·$\hat{y} \in \mathbb{R}^{n_y}$ is the predicted output vector. It can also be denoted $a^{[L]}$ where L is the number of layers in the network.

Common forward propagation equation examples:

$a = g^{[l]}(W_x x^{(i)} + b_1) = g^{[l]}(z_1)$ where $g^{[l]}$ denotes the l^{th} layer activation function

$\hat{y}^{(i)} = softmax(W_h h + b_2)$

· General Activation Formula: $a_j^{[l]} = g^{[l]}(\sum_k w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]}) = g^{[l]}(z_j^{[l]})$

· $J(x, W, b, y)$ or $J(\hat{y}, y)$ denote the cost function.

Examples of cost function:

· $J_{CE}(\hat{y}, y) = -\sum_{i=0}^{m} y^{(i)} \log \hat{y}^{(i)}$

· $J_1(\hat{y}, y) = \sum_{i=0}^{m} \mid y^{(i)} - \hat{y}^{(i)} \mid$

Appendix B

Repository Structure

written by Mattheo Mahnke

B.1 General Repository Structure

The project is divided into three repositories:

- The documentation repository (`https://gitlab.com/embedsprojekt_wise21/dokumentation`)
- The FFT repository (`https://gitlab.com/embedsprojekt_wise21/fft`)
- The neural network repository (`https://gitlab.com/embedsprojekt_wise21/ki`)

They can all be found under the group `https://gitlab.com/embedsprojekt_wise21`.

B.2 Documentation Repository

The documentation repository contains the sources to this document which is created by LaTeX. It also contains a few markdown files on how to do certain things, e.g. using the microphone of the PYNQ Z1 board, starting a Jupyter Notebook and some other stuff that can be helpful when recreating this project.

B.3 FFT Repository

The FFT repository was created for recording with the on-board microphone and pre-processing the data for the neural network. It contains the Vivado projects for reading and processing the data. Additionally the python code used to read the results of the pre-processing (the Mel Frequency Cepstral Coefficients) from the FPGAs to be able to use them in the neural network, which would currently run on the microcontroller, is also there.

B.4 Neural Network Repository

This repository contains a python script to be able to generate training data and use it to train the network. The script has a few command line arguments for easy configuration but reads most of its configuration from the config located in the *config* folder. Call the script with *–help* for further info on the command line interface. The configuration file uses Tom's Obvious Minimal Languages (TOMLs) and is separated into the steps of the data pipeline.

"input_data" contains the configuration for the raw input samples. Each input section has to have its own section defined which contains the name of the category and the path where the samples are located. These are the categories the neural network will use for classification.
The synthesis block is used to configure the data synthesis. It allows configuration of the individual sample processors, which are used to diversify samples as mentioned in 7.1. It can also be used to overwrite the configuration of a sample processor for a specific synthesis step, which is done by including that configuration in the configuration block of the step. This is mostly used to set the apply probabilities for processors, but could also be used to e.g. change out the backgrounds to load for the *BackgroundNoiseProcessor*. Note that apply probabilities are normalized and use a default value of 0.5. This results in the values not being in percent, because they are changed based on the apply probabilities of the other processors. The processor to use the config for is resolved by the name of the class, which has to match the name of the section. The next section allows configuration of parameters used during training, e.g. the learning rate. The last block configures the small inference script built into the program. The configuration file is heavily commented and its recommended to have a look at these to understand the configureability of the pipeline.

B.5 Extending the Pipeline

The code was written with extension of the data preparation pipeline in mind.
Adding a new word category that the network shall be able to detect can be done by adding a configuration block for it under the input data configuration and providing input samples. Note that this will break compatibility with any previously trained networks since it changes the topology of the network.
The folder "*src*" in the neural network repository contains the sources that back the script for the training pipeline and inference.
Audio samples are loaded using an *AudioLoader*,
they are located under *trainingData/inputLoading/audioLoaders*. New loaders, other than MP3 which is already implemented, can be provided by creating a class that inherits from *AudioLoader*. The *AudioLoaderFactory* will automatically detect this class, after adding it to the *__all__* attribute of the *audioLoaders* package (note that this depends on the class being in the *audioLoaders* package). The factory will use the loader for the files with the extension it returns when calling *get_loader_extension*. To add a new sample processor one has to create a new class under *trainingData/synthesis/sampleProcessors* that inherits from *SampleProcessor*. The class has to be added to the *__all__* attribute of the *sampleProcessors* package. This new class is automatically loaded whenever sample processors are used, and is configured using the configuration block with the same name.
Under *trainingData/synthesis/steps* one is also able to create entirely new pipeline steps. A new pipeline step has to be added in the initialiser of the *SynthesisStepPipeline* in the same module, but configuration is autoloaded by the superclass, as its done for the sample processors.
If one wants to use sample processors in the new pipeline step, it is recommended to inherit from *SampleProcessorStep* which includes code to load the sample processors and configure them for the current step. It can also be used to just create new configuration contexts for the sample processors, as its done for the *PreStitching* and *PostStitching* steps.

B.6 Training Database

Synthesised samples are saved into a (SQLite) database for later use. The path to use for the database can be configured in the configuration file under the synthesis block with the config key *db_path*.
Functions for saving and loading samples are provided in the *trainingData/trainingDatabase* package. Samples are saved together with their labels and are automatically loaded for training if the script was called without the *–synthesis* flag. The number of samples to be used for training can be limited under the training configuration block using the key *max_number_of _samples*. This will cause the database to be randomly sampled.

List of Figures

List of Acronyms

Adam Adaptive Moment Estimation

AI Artificial Intelligence

API Application Programe Interface

ASIC application specific integrated circuit

ASR Automatic Speech Recognition

AXI Advanced eXtensible Interface Bus

CNN Convolutional Neural Network

CPU Central Processing Unit

DAW Digital Audio Workstation

DCT Discrete Cosine Transform

DFT discrete Fourier transform

DMA Direct Memory Access

FFT fast fourier transform

FIR finite impulse response

FPGA field programmable gate array

GAN Generative Adversarial Network

GPU Graphics Processing Unit

HMI Human Machine Interaction

IC integrated circuit

IP-Core intellectual property core

LPC Linear Predictive Coding Coefficient

LSTM Long Short Term Memory

LUT look-up table

MEMS Micro Electro-Mechanical System

MFC mel-frequency cepstrum

MFCC Mel Frequency Cepstral Coefficient

MNIST Modified National Institute of Standards and Technology

NN Neural Network

PCM Pulse Code Modulation

PDM Pulse Density Modulation

PL Programmable Logic

PS Processing System

QNN Quantized Neural Network

RAM random access memory

rc car radio-controlled car

ReLU Rectified Linear Unit

RGB Red Green Blue

RNN Recurrent Neural Network

SNR signal to noise ratio

SoC system-on-chip

SVM Support Vector Machine

TOML Tom's Obvious Minimal Language

WUW Wake-Up-Word

Bibliography

[17] *PYNQ-Z1 Board Reference Manual*. 2017. URL: https://digilent.com/reference/_media/reference/programmable-logic/pynq-z1/pynq-rm.pdf (visited on 03/01/2022).

[21] *PYNQ: Python productivity for Xilinx platforms*. 2021. URL: https://pynq.readthedocs.io/en/latest/index.html (visited on 03/05/2022).

[3Bl17] 3Blue1Brown. *What is backpropagation really doing? | Deep learning, chapter 3*. Nov. 2017. URL: https://www.youtube.com/watch?v=Ilg3gGewQ5U (visited on 03/08/2022).

[Alt+21] Philipp Altnickel et al. *Gesten- und Objekterkennung durch schwache FPGAs in autonomen Fahrzeugen mittels neuronaler Netze*. German. Tech. rep. Univ. of Applied Sciences Bremen, Germany, Sept. 1, 2021. 153 pp.

[Aul19] Sugondo Hadiyoso; Bayuaji Kurniadhani; Rita Magdalena; Suci Aulia. "FPGA-based implementation of speech recognition for robocar control using MFCC". In: *Universitas Ahmad Dahlan* Vol 17, No 4: August 2019 (2019). URL: http://journal.uad.ac.id/index.php/TELKOMNIKA/article/view/12615/6758.

[CoolTuk65] John W. Tukey James W. Cooley. "Math. Comput. 19, S. 297–301". In: *An algorithm for the machine calculation of complex Fourier series*. 1965.

[Dao+17] Van-Lan Dao et al. "Hardware Implementation of MFCC Feature Extraction for Speech Recognition on FPGA". In: Nov. 2017, pp. 248–254. ISBN: 978-3-319-49072-4. DOI: 10.1007/978-3-319-49073-1_27.

[digital-audio12] Ph.D. Thomas Kite. *Understanding PDM Digital Audio*. Tech. rep. Audio Precision, 2012. 9 pp. URL: http://users.ece.utexas.edu/~bevans/courses/realtime/lectures/10_Data_Conversion/AP_Understanding_PDM_Digital_Audio.pdf (visited on 03/02/2022).

[DMA-PYNQ] *Access of DMA in Python*. URL: https://pynq.readthedocs.io/en/v2.7.0/pynq_libraries/dma.html (visited on 03/08/2022).

[FFT] *Fast Fourier Transformation FFT*. German. URL: https://www.nti-audio.com/de/service/wissen/fast-fourier-transformation-fft (visited on 03/04/2022).

[FIR] *dsp.FirDecimator*. URL: https://www.mathworks.com/help/dsp/ref/dsp.firdecimator-system-object.html (visited on 03/06/2022).

[FIRD] URL: https://www.mathworks.com/help/signal/ug/fir-filter-design.html (visited on 03/07/2022).

[GD15] Kishori R. Ghule and Ratnadeep R. Deshmukh. "Feature Extraction Techniques for Speech Recognition: A Review". In: 2015.

[GEO19] ALAN GEORGE. *INSIDE THE FFT BLACK BOX : serial and parallel fast fourier transform algorithms*. S.l: CRC PRESS, 2019. Chap. 3. ISBN: 9780367399290.

[Hei17] Sebastian Heinz. *Deep Learning Grundlagen - Teil 1: Einführung | STATWORX*. 2017. URL: https://www.statworx.com/de/blog/deep-learning-teil-1-einfuehrung/ (visited on 03/05/2022).

[Hut+20] Colin von Huth et al. *Bericht zum Projekt "Neuronale Netze auf strahlungstoleranten FPGAs für die Raumfahrt"*. German. Tech. rep. Univ. of Applied Sciences Bremen, Germany, Feb. 14, 2020. 88 pp. URL: http://homepages.hs-bremen.de/~jbredereke/de/forschung/veroeffentlichungen/neuronale-netze-fpgas-projekt-1920.html (visited on 12/13/2021).

[Inc21] Coursera Inc. *Coursera*. 2021. URL: https://www.coursera.org/ (visited on 03/05/2022).

[JC14] Siddhant Joshi and A.N. Cheeran. "MATLAB Based Feature Extraction Using MFCC for ASR". In: June 2014, pp. 1820–1823.

[Jou08] Szu-Chen (Stan) Jou. "Automatic Speech Recognition on Vibrocervigraphic and Electromyographic Signals". PhD thesis. Carnegie Mellon University, 2008. URL: https://www.csl.uni-bremen.de/cms/images/documents/publications/PhD-Jou.pdf.

[kam22] *AXI-Stream FIFO. Homepage*. German. URL: https://www.kampis-elektroecke.de/fpga/zynq/axi-stream-fifo/ (visited on 03/03/2022).

[Kep11] Veton Kepuska. "Wake-Up-Word Speech Recognition". In: *Speech Technologies*. Ed. by Ivo Ipsic. Rijeka: IntechOpen, 2011. Chap. 12. DOI: 10.5772/16242. URL: https://doi.org/10.5772/16242.

[LCB] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. *THE MNIST DATABASE of handwritten digits*. URL: http://yann.lecun.com/exdb/mnist/ (visited on 03/08/2022).

[McD17] Conor McDonald. *Machine learning fundamentals (I): Cost functions and gradient descent*. Nov. 2017. URL: https://towardsdatascience.com/machine-learning-fundamentals-via-linear-regression-41a5d11f5220 (visited on 03/08/2022).

[McL09] Ian McLoughlin. *Applied speech and audio processing : with Matlab examples*. Cambridge New York: Cambridge University Press, 2009. ISBN: 978-0521519540.

[mfcc-core] *MFCC FPGA-Core*. URL: https://github.com/lambdaconcept/mfcc (visited on 03/07/2022).

[mic] Knowles Acoustics. *SPK0833LM4H-B*. Version Revision B. Knowles Electronics.

[mik21] *FPGA. Homepage*. URL: https://www.mikrocontroller.net/articles/FPGA#:~:text=FPGA%20ist%20die%20Abk%C3%BCrzung%20f%C3%BCr,von%20Schaltungen%20realisiert%20werden%20k%C3%B6nnen. (visited on 10/13/2021).

[Mot02] Petr Motlíček. *Feature Extraction in Speech Coding and Recognition*. Tech. rep. Portland, US, 2002, pp. 1–50. URL: https://www.fit.vut.cz/research/publication/7069.

[Mül+21] Felix Müller et al. *Applying Binarized Neural Networks on FPGAs to an Autonomous Driving Problem*. Tech. rep. Univ. of Applied Sciences Bremen, Germany, Mar. 31, 2021. 50 pp. URL: http://homepages.hs-bremen.de/~jbredereke/de/forschung/veroeffentlichungen/bnns-on-fpgas-driving-projekt-2021.html (visited on 12/13/2021).

[Mül21] Felix Müller. "Dynamisches Tiling auf schwachen FPGAs zur Objekterkennung mithilfe kleiner neuronaler Netze". German. Bachelorthesis. Univ. of Applied Sciences Bremen, Germany, June 23, 2021. URL: http://homepages.hs-bremen.de/~jbredereke/de/forschung/veroeffentlichungen/mueller-bsc-thesis-2021.html (visited on 12/13/2021).

[NG] Andrew NG. *Standard notations for Deep Learning*. permission to course material required. URL: https://www.coursera.org/learn/neural-networks-deep-learning/resources/YsZjP (visited on 03/05/2022).

[NKM] Andrew Ng, Kian Katanforoosh, and Younes Bensouda Mourri. *Deep Learning | Coursera*. URL: https://www.coursera.org/specializations/deep-learning (visited on 03/05/2022).

[nyquist-shannon49] C.E. Shannon. "Communication in the Presence of Noise". In: *Proceedings of the IRE* 37.1 (1949), pp. 10–21. DOI: 10.1109/JRPROC.1949.232969.

[Pap21] Alessandro Pappalardo. *Xilinx/brevitas*. 2021. URL: https://github.com/Xilinx/brevitas (visited on 03/01/2022).

[pdm] *PDM Microphones and Sigma-Delta A/D Conversion*. URL: https://tomverbeure.github.io/2020/10/04/PDM-Microphones-and-Sigma-Delta-Conversion.html (visited on 03/02/2022).

[Pog03] Kai Poguntke. *IP-Cores Proseminar 2003*. German. Tech. rep. Universitaet Ulm, 2003. 9 pp. URL: https://www.informatik.uni-ulm.de/ni/Lehre/SS03/ProSemFPGA/IP-Cores.pdf (visited on 03/03/2022).

[PWS18] Pasd Putthapipat, Chutitep Woralert, and Phumiphat Sirinimnuankul. "Speech recognition gateway for home automation on open platform". In: Jan. 2018, pp. 1–4. DOI: 10.23919/ELINFOCOM.2018.8330715.

[PZ1] *PYNQ-Z1 Reference Manual. Homepage*. URL: https://www.nti-audio.com/portals/0/pic/news/FFT-Time-Frequency-View-540.png (visited on 03/02/2022).

[RF16] Joseph Redmon and Ali Farhadi. "YOLO9000: Better, Faster, Stronger". In: *CoRR* abs/1612.08242 (2016). arXiv: 1612.08242. URL: http://arxiv.org/abs/1612.08242 (visited on 03/05/2022).

[Ros07] Thomas D. Rossing. *Springer handbook of acoustics*. Springer, 2007. ISBN: 9780387304465.

[Sch14] Jürgen Schmidhuber. "Deep Learning in Neural Networks: An Overview". In: *CoRR* abs/1404.7828 (2014). arXiv: 1404.7828. URL: http://arxiv.org/abs/1404.7828 (visited on 03/05/2022).

[SJ08] Xuancheng Shao and Steven G. Johnson. "Type-II/III DCT/DST algorithms with reduced number of arithmetic operations". In: *Signal Processing* 88.6 (2008), pp. 1553–1564. ISSN: 0165-1684. DOI: https://doi.org/10.1016/j.sigpro.2008.01.004. URL: https://www.sciencedirect.com/science/article/pii/S016516840800008X.

[stream-spec10] ARM. *AMBA 4 AXI4-Stream Protocol*. ARM. 2010.

[Tur10] Clay S. Turner. "A Fast Binary Logarithm Algorithm [DSP Tips amp; Tricks]". In: *IEEE Signal Processing Magazine* 27.5 (2010), pp. 124–140. DOI: 10.1109/MSP.2010.937503.

[wake-word-verification] *Requirements for Cloud-Based wake word verification*. URL: https://developer.amazon.com/en-US/docs/alexa/alexa-voice-service/implement-ww-verification.html (visited on 03/06/2022).

[xFFt21] Xilinx. *Fast Fourier Transform v9.1*. Version PG109. Xilinx. 2021.

[Xil21] Xilinx. *FIR Compiler v7.2*. Version PG109. Xilinx. 2021.