Embedded Object Detection with Convolutional Neural Networks

Bachelor Thesis in Electrical Engineering and Information Technology
Lucerne University of Applied Sciences and Arts - Engineering & Architecture

Author: Cyrill Durrer

Supervisor: Prof. Dr. Jürgen Wassner
Expert: Thomas Schmidiger

Spring semester 2020

Bachelor-Thesis an der Hochschule Luzern - Technik & Architektur

Titel	**Embedded Object Detection with Convolutional Neural Networks**
Diplomandin/Diplomand	**Durrer Cyrill**
Bachelor-Studiengang	**Bachelor Elektrotechnik und Informationstechnologie**
Semester	**FS20**
Dozentin/Dozent	**Wassner Jürgen**
Expertin/Experte	**Schmidiger Thomas**

Abstract Deutsch
Convolutional Neural Networks (CNN) werden häufig für die Bildverarbeitung, speziell für die Detektion von Objekten, eingesetzt. Ein an der HSLU entwickelter Low-Cost CNN Accelerator soll dabei helfen, diese auf eingebettete Systeme mit wenig Rechenleistung zu bringen. Die Zielhardware besteht aus einer Kombination aus Processing System (PS) und Programmierbarer Logik (PL).
Ein einfacher CNN Single-Shot-Detektor (SSD-7) wurde ausgewählt, trainiert und getestet. Er erreichte einen mAP-Wert («Mean Average Precision») von 0.308. Nach der Umwandlung in eine binär approximierte Form erreichte der Algorithmus nahezu gleich gute Resultate (0.303). Ein Abschätzung der Ausführungsgeschwindigkeit dieses Netzwerks auf einer Mid-Range sowie einer Low-End Hardware ergab Verarbeitungsraten von 99.23 beziehungsweise 6.54 Bilder pro Sekunde. Dies ist ausreichend für viele Echtzeit-Anwendungen.

Abstract Englisch
Convolutional Neural Networks (CNN) are widely used for image processing and especially object detection. A Low-Cost CNN Accelerator developed at HSLU aims to bring these algorithms onto embedded systems with limited computational power. The target hardware consists of a combination of processing system (PS) and programmable logic (PL).
A lightweight CNN single-shot detector (SSD-7) was selected, trained and tested, achieving a mean average precision (mAP) of 0.308. This network was converted to a binary approximated form and tested again, achieving almost the same performance as the original (0.303). For this network, the maximum inference speed for a mid-range as well as a low-end hardware was estimated. With inference speeds of 99.23, respectively 6.54 inferences per second, this approach could be useful for many embedded systems which require a low-cost, low-power object detector with real-time capabilities.

Ort, Datum Horw, 07.06.2020

Contents

1. Introduction

This thesis is written in the context of a bachelor degree at the Lucerne University of Applied Sciences and Arts (HSLU). It is part of a project aiming to provide a framework which allows to convert a neural network such that it can be used on an embedded system consisting of a combination of processing system (PS) and programmable logic (PL). The key to running a computationally expensive task, such as object classification or detection, on an embedded system is resource optimization and hardware acceleration.
This is done in two stages. The first stage is an evolutionary algorithm, which allows the user to generate a resource optimized network architecture with a minimum reduction in performance[16]. In the second stage, the optimized network is converted to a binary approximated model (BinArray[11]), in which most of the computation at inference time can be executed on a hardware accelerator implemented on the PL.

1.1 Outline

Prior to the publication of this thesis, the Low-Cost CNN Accelerator framework was only tested for image classification tasks. The objective of this work is to assess if this framework can also be applied to object detection tasks. The chosen network architecture is a Single Shot MultiBox Detector[18] (SSD, chapters 3 and 4). The question of interest is how many binary filters (M) are required for the binary approximation to keep the performance of the network and whether or not it is even possible to maintain the performance on a satisfactory level. After assessing the performance of the binary approximated network, the inference speed of the network on an embedded system can be estimated. This is an indication for the maximum frame rate of an image stream which can be processed in real-time. Because this is highly dependent on the available hardware resources, the results are always set in relation to the amount of resources required. This enables a potential user to determine the optimum values for himself and provides an estimate of what performance and inference speed can be achieved.

1.2 Hardware and Tools

1.2.1 Software

The code in this thesis is written in Python. The parts dealing with deep neural networks use the Keras[3] library and the TensorFlow-framework[6].

1.2.2 Version Control

To keep track of the progress of the project and to be able to share it with others, Git and GitLab/EnterpriseLab are used. Because the code needs to fulfill various requirements like training and predicting in floating point precision (git-branch: SSD_Standard) and converting the model to the binary approximation (git-branch: SSD_BA), multiple branches are used in parallel.

1.2.3 Development and Training Hardware

The code was mainly executed on two machines:

Machine 1: "Lenovo ThinkPad X1 Extreme" notebook running on Windows 10 equipped with an "Intel Core i7-8750H" hex-core CPU, a "Nvidia GeForce GTX 1050 Ti" GPU and 32 GB of RAM. This machine was mostly used to write the code, test it and check predictions of the network.

Machine 2: Workstation running on Ubuntu equipped with an "Intel Core i7-8700" hex-core CPU, a "Nvidia RTX2080 Ti" GPU and 64 GB of RAM. Due to the powerful GPU most of the neural network training documented in this thesis was executed on this machine.

1.2.4 Target Hardware

In this project, a general feasibility of this approach for hardware accelerators with field-programmable gate arrays (FPGA) is investigated. However, to be able to give concrete numbers on the efficiency of the implementation, two possibilities are presented. As an example of a mid-range hardware, the Xilinx Zynq-7045 (XC7Z045) with an Kintex-7 FPGA and a dual-core ARM Cortex-A9 CPU serves as target hardware. To investigate the performance which can be achieved using low-end hardware, the similar but much cheaper Zynq-7010 (XC7Z010) with the same ARM CPU, but with a smaller Artix-7 FPGA is chosen. Both are built as system on a chip (SoC) an belong to the Zynq-7000 family. Based on [11], 400 MHz serves as maximum clock frequency of the FPGA. It is assumed that the Artix-7 can run at the same clock frequency as the Kintex-7. This is probably not possible, but with newer devices of a similar price range a clock frequency of 400 MHz or more can be achieved.
The hardware constraints limiting parallel processing, and thus inference speed, are taken from the official datasheets of the respective hardware (appendix, section 12.2).

2. Object Detection

2.1 Overview

In order to understand object detection it is crucial to know what image classification and object localization are. This section describes the two concepts and transitions to the combination of the two.

2.1.1 Image Classification

In image classification, the objective is to find the class best fitting the image. If there is a car in the foreground of the image, an image classification network should output the class "car". It does not matter that there is a pedestrian somewhere in the background, because the classification network's task is only to classify the most obvious object in the picture.

2.1.2 Object Localization

If it is important to know where in the image an object is, object localization must be applied. A combination of image classification and object localization gives information about where in the image the object is and finds the corresponding class[12]. The localization information typically consists of a set of coordinates (section 2.1.5).

2.1.3 Object Detection: Recognizing Multiple Objects

Algorithms which are able to classify and localize multiple objects in an image are called object detectors. In fields such as autonomous driving, industrial machine control and robotics this is of great interest, because in many situations it is essential to get information about more than one object in an image or video.

2.1.4 Class Encoding

To train a neural network, a one-hot encoding for classes is often used. This means that every possible class which occurs in the dataset gets its own cell in a matrix or tensor. One-hot encoding allows the network to assign a confidence value to every class, indicating how high the network predicts the probability that this object is actually there. Based on these probabilities, the non-maximum suppression stage (NMS, section 3.1.3) is able to filter the predictions for the ones with the highest probability of being true.
The one-hot encoding in this thesis consists of six values: Five for the object classes and one for the background class.

2.1.5 Bounding Boxes

In this thesis, rectangular bounding boxes are used to describe the position of the detected objects. There are multiple ways to represent a rectangle as a vector. In this work, bounding box coordinates are encoded in the "centroid" format:

$$centroid\,coordinates = [cx, cy, w, h] \tag{2.1}$$

where cx and cy represent the center coordinates and w and h the width and height of the box. Coordinates are scaled to a number between zero and one, representing the proportion of the total image size instead of the number of pixels (equation 2.2).

$$[cx_{rel}, cy_{rel}, w_{rel}, h_{rel}] = \frac{[cx_{abs}, cy_{abs}, w_{abs}, h_{abs}]}{[x_{image}, y_{image}, x_{image}, y_{image}]} \tag{2.2}$$

To convert the coordinates back into absolute pixel values, equation 6.9 is applied.

2.2 Datasets

To assess the applicability of the Low-Cost CNN Accelerator framework on object detection tasks, a fitting dataset needs to be chosen. This dataset should contain a rather small amount of different classes to allow a relatively small network to effectively differentiate between them. It should also contain enough labelled images to train the network appropriately and the labels should contain as few errors and inaccuracies as possible.

2.2.1 Pascal Visual Object Classes (VOC)

Pascal VOC was an object detection challenge lasting from 2005 to 2012. In 2007, a dataset containing about 10'000 images with 20 classes was released. It was increased until it contained more than 20'000 images in 2012[8].
This dataset is considered one of the most important in the field. It helped to set the conditions for modern object detection. With 20 classes it is already quite challenging for a simpler network and thus not suited for this project.

2.2.2 Common Objects in Context (COCO)

COCO is a large-scale object detection, segmentation and captioning dataset originally created by Microsoft in 2014. It consists of about 328'000 labelled images with 80 object classes. Apart from object detection it can also be used for image segmentation[17].
Because it is newer and far bigger than Pascal VOC, nowadays this dataset is very common to train more complex object detectors. With as many as 80 classes, the network used for detection needs to be large and trained on big amounts of data in order to effectively discriminate between classes. Because this project is about the general possibility of running an object detection network on an embedded system, a simpler dataset is more suitable.

2.2.3 "Udacity" - Dataset

Udacity is an educational platform which offers online courses in various topics of engineering. One of those courses aims to help students become self-driving-car engineers. For the purpose of teaching the students how to build an object detection algorithm, the officially available datasets seemed too large and too complicated, thus Udacity decided to build their own open-source self-driving-car dataset with only five categories (car, truck, pedestrian, bicyclist, traffic light). It consists of approximately 20'000 labeled images with a resolution of 1920x1200 pixels[24].
As proposed in [9], a downscaled version of this dataset was first used to train the network. It contains all the five categories and images with a resolution of 300 x 480 pixels. After some tests with this dataset, it became apparent that there are lots of errors and missing objects in the labels.

2.2.4 "Roboflow" Improvements on the "Udacity" Dataset

The dataset-provider Roboflow noticed this errors in the Udacity self-driving-car dataset as well. Errors in 4'986 (33%) of the images were reported[4]. Roboflow improved and re-released the dataset[5].
To minimize errors from wrong labels in the dataset, the improved version is used in this project. It is denoted as Udacity/Roboflow dataset. A downscaled version of the images to 512x512 pixels

is chosen to limit the complexity of the network. The code written to edit the label formats to fit the network implementation and to split the dataset into training and validation set can be found in the appendix, section 12.3.1.

2.3 Performance Metric

2.3.1 Overview

To determine the performance of a deep learning algorithm, there are many possible metrics which can be used. The choice of the metric depends mainly on the area of application. For the sake of comparability, there is a standard metric in most fields of application, allowing comparisons between different solutions for the same or a similar problem. In object detection, the commonly used metric is mean average precision (mAP)[23]. The computation of the mAP in this project was implemented from scratch, because the available implementations were mostly suited for the Pascal VOC or COCO datasets and required a very different format of the ground truth and prediction data. The source code is in the appendix, section 12.3.2.

2.3.2 Intersection over Union

The first step to measure the performance of an algorithm is to define when it is correct and when it is not. In object detection, intersection over union (IoU, also known as Jaccard index) is often used to determine if a predicted bounding box has enough overlap with the ground truth box to be considered correct. IoU is calculated by dividing the area of the intersection of both boxes by the area of the union of those boxes:

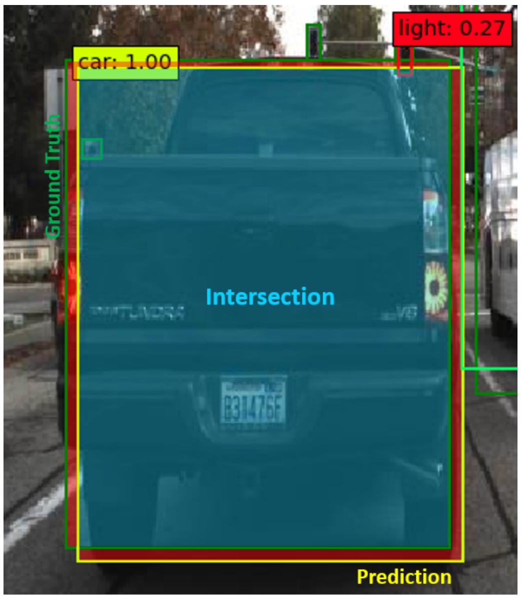


Figure 2.1: Visual explanation of IoU with an example from the Udacity[24]/Roboflow[5] dataset. The blue area is the intersection and the blue and the red areas together form the union.

$$IoU = \frac{Area\ of\ Intersection}{Area\ of\ Union} \tag{2.3}$$

The green box in figure 2.1 is the ground truth given from the dataset and the yellow box is predicted by the algorithm. In this example, the IoU value is very high and the prediction is marked as correct. The most commonly used IoU threshold is 0.5, meaning every predicted bounding box with the correct class and a IoU above 0.5 is considered a true positive. The higher this threshold is chosen, the more accurate the prediction boxes have to be. A more detailed explanation is provided in [15]. This metric is also used to remove overlapping boxes predicting the same object (section 3.1.3).

2.3.3 Confusion Matrix

The confusion matrix is a way of describing the performance of a classifier in a table with four cells. The horizontal dimension represents the actual (ground truth) values from the dataset and the vertical dimension the predicted values from the network under test. Both dimensions are split into positive and negative, where positive refers to the existence of this object or prediction in the picture and negative to the absence of the respective object or prediction.

		Actual value		
		p	**n**	**total**
Predicted value	**p′**	True Positive	False Positive	P′
	n′	False Negative	True Negative	N′
	total	P	N	

This table reveals the following information about the algorithm:

- True positive (TP): How many classifications were actually correct
- False positive (FP): How many classifications were wrong
- False negative (FN): How many ground truth objects were not classified
- True negative (TN): How many classifications were correctly predicted as negatives (no object)

This is useful for image classification, where for example a network has to decide whether or not some object is in the image. However, this representation can not be directly used for object detection, because there are no actual negatives, but rather objects at a certain location and background everywhere else. A consequence of this is that there are no TN in object detection. For a prediction to be marked as TP, its bounding box needs to achieve an IoU above a certain threshold with the ground-truth box and the predicted class has to be correct. If either the class is incorrect or the predicted bounding box is not fulfilling the IoU criterion, it is marked as FP. Conversely, when there is a ground truth box which does not match with any predicted box, it counts as FN. This is done for every class separately.

2.3.4 Precision and Recall

From the values in the confusion matrix, the metrics precision and recall can be calculated. Precision describes, what percentage of the predicted object is actually correct:

$$Precision = \frac{TP}{TP + FP} \tag{2.4}$$

Recall describes what percentage of the ground truth objects in the image were detected:

$$Recall = \frac{TP}{TP + FN} \tag{2.5}$$

While both of these metrics describe some quality of the algorithm, no single one of them is a good measure for performance by itself. A combination of these values has to be applied.

2.3.5 Precision-Recall Curve and Average Precision

To calculate the mAP of an algorithm, the computation of the precision-recall curve (PR curve) is necessary. As the name says, this curve plots precision (mostly on the y-axis) against recall (mostly on the x-axis). The various pairs of precision and recall values are obtained by using various confidence threshold values in the NMS stage (section 3.1.3).
The higher the chosen confidence threshold, the higher the precision gets, because only the most confident of all predictions are evaluated. As a trade-off the recall value will be very low. With a low confidence threshold the recall value is high, because many predictions will be evaluated and some of them will most likely be correct. As a consequence of all the wrong predicted boxes the precision will now be very low.
For every class, the precision values are plotted against the respective recall values for various confidence thresholds to obtain the PR curve[2][21] (section 4.2.2). The area under this curve is called the average precision (AP). Because the PR curve is different for every class, so is the AP value. Figure 4.5 is an example of such a curve with the corresponding AP values in table 4.3. The chosen confidence thresholds to generate the PR curve are mentioned in section 4.2.2.

2.3.6 Mean Average Precision

The mean of the APs of all different classes in the dataset is computed to obtain the mAP. This metric finally represents the performance of the algorithm in one number.
It is not completely standardized how many points of the PR curve should be used and how they should be interpolated. Among the challenges in the field of object detection, different evaluation metrics like Pascal VOC and COCO (sections 2.3.7 and 2.3.8) have been developed.

2.3.7 Pascal VOC 2007 & 2012 Evaluation Metrics

For the Pascal VOC 2007 evaluation metric, the PR curve needs to consist of 11 values with confidence thresholds chosen to achieve recall values of 0, 0.1, 0.2, ..., 1.0. The mean of the 11 corresponding precision values represents the AP, and the mean of the AP over all classes (20 in the case of the original Pascal VOC challenges since 2007) quantifies the algorithm performance as mAP with Pascal VOC evaluation metric[8].
For Pascal VOC 2012, instead of the interpolated values, the true area under the PR curve has to be calculated. Both metrics use a constant IoU threshold of 0.5.
In this thesis, the Pascal VOC 2012 evaluation metric is used to calculate the mAP, because it has proved to be an accurate metric for many projects and it is easier to implement than the COCO metric which consists of 12 metrics. Also, the widespread use of this metric makes the performance comparable to other object detectors.

2.3.8 COCO Evaluation Metric

The newer COCO challenge uses 12 metrics for characterizing the performance of an object detector. Six of those are mAP metrics, the other six are mean average recall (mAR) metrics.
The primary challenge metric evaluates the mAP with 10 different IoU thresholds of 0.5 to 0.95 in steps of 0.05. The mean of these 10 values gives the desired value. This adjustment of the metric rewards detectors with a better localization, which are predicting the bounding boxes more accurately.
One of the remaining mAP metrics is a standard Pascal VOC evaluation metric. Another is the same with another IoU threshold of 0.75 instead of 0.5. The remaining three mAP metrics assess the performance of the algorithm for detecting small, medium and large objects[7].
The mAR metrics base on a curve where the recall is plotted against various IoU thresholds (from 0.5 to 0.95). The mAR value corresponds to two times the area under this curve. This metric is not applied in this thesis.

3. Single-Shot MultiBox Detector

In this thesis some familiarity with neural networks and especially CNNs is presumed. An overview of these algorithms and an explanation to many technical terms used here can be found in [20]. Here is a very short explanation of the most important terms used:

- Tensor: A multi-dimensional matrix (3-dimensional in this thesis)
- Feature map/tensor: Intermediate result in the network after the data is processed by some layer
- Pooling: A method of downscaling an image/feature map (to a quarter the size in this thesis)
- Weight: Element of the filter tensor whose value is adjusted during training

3.1 Concept and Architecture

Single-Shot MultiBox Detector[18] (SSD) is an algorithm developed for efficient object detection and localization. The base network used in the original paper is a VGG-16[22] network without the fully connected layers. In contrast to most other CNN models such as R-CNN[13] and YOLO[19], SSD is fully convolutional and does not contain any fully connected layers. The predictions of the network go through a NMS stage to find the most accurate predictions (figure 3.1).

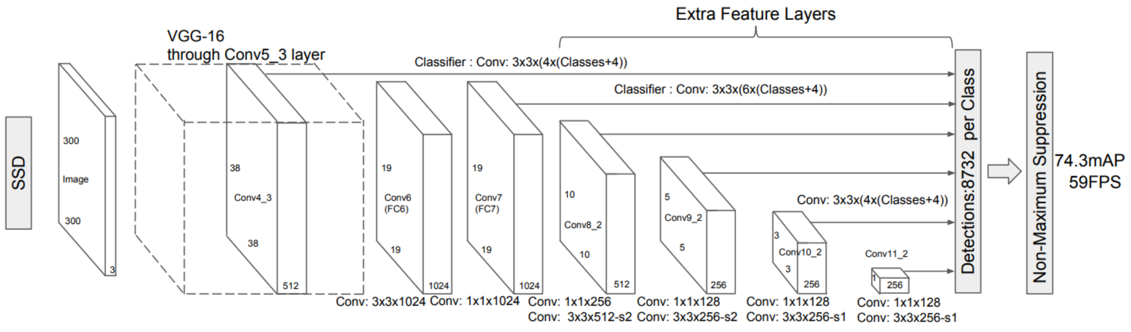


Figure 3.1: SSD architecture, input: 300x300x3 (image source: [18])

3.1.1 Predictor Layers

The predictor layer (detections layer in figure 3.1) is the part where SSD differentiate themselves from other object detectors like R-CNN or YOLO. In contrast to most other CNN architectures, the SSD uses feature maps from multiple layers to compute the prediction instead of only using the last layer. These unconventional connections allow the network to detect objects of various sizes in the input image more easily than most other approaches. The feature maps of some layers in the network are forwarded to the predictor layer before applying pooling (figure 4.1).
There are two predictor layers for every feature map used for prediction: one to predict the class and the other to predict the bounding box offset. The class prediction layer is responsible for assigning a confidence value to every possible class in a one-hot encoding, representing the probability of this object being in the corresponding bounding box. The boxes predictor layer estimates the offsets of the center coordinates as well as the offsets (scaling) of the width and height of the related anchor box. Figure 4.2 provides a detailed view of the predictor layer in the SSD-7.

3.1.2 Anchor Boxes

Every value of every feature map passed on to the predictor layers represents the center of a perception field on the input image. For every center point, multiple possible anchor boxes are generated with different height/width ratios and scales, depending on the position of the value in the feature map and in the network. The bigger the feature map, the smaller the related anchor boxes.
To map the outputs of the boxes layers (figure 4.2) to coordinates in the image, an anchors tensor is required. This tensor contains the information of every anchor box in every feature map which is forwarded to the predictor layer. The anchors tensor only depends on the network architecture and input image dimensions. It does not depend on the values passed through the network and thus can be generated as soon as the architecture is fixed. The anchors tensor remains static during training and inference (grey area in figure 4.2).

3.1.3 Non-Maximum Suppression Stage

Independent of the number of objects in the image, the SSD always predicts the same amount of bounding boxes and respective class confidence values. Therefore, the shape of the output tensor (before the NMS stage) is fixed. In the case of the original SSD300 it puts out 8732 predictions per class. To find the boxes with the highest probability of being true, the output is forwarded to an NMS stage consisting of three parts: confidence threshold, intersection filter and top-k filter.
To drastically reduce the number of predictions, only those with a confidence level above a certain threshold are kept. This does not include the confidence for the background class. Because there are still many overlapping boxes left which predict the same object, all those with an IoU greater than an IoU overlap threshold of 0.45 with some prediction box with a higher confidence are discarded as well. This value is taken from the original paper[18]. This is done for every class separately to prevent the algorithm from discarding for example a pedestrian standing in front of a car. To limit the maximum number of possible predictions per image, only some predefined number (k) of predictions with the highest confidence values are kept. This only affects the predictions if there were more than k predictions left after the intersection filter. Figure 4.3 shows this section of the SSD-7 in further detail.

3.1.4 Activation Functions

In order to introduce non-linearity into the network, activation functions are used. The outputs of the convolution are forwarded to the activation function, which computes the output (activation) being passed on to the next layer.
In classification networks, the softmax function is most common in the output layer. It normalizes the values into a finite interval to be easily interpreted as probability distribution, which represents the confidence of the classification. Following this scheme, the classification of the SSD also uses the softmax function to convert the non-normalized outputs of the class prediction layers to a probability between zero and one. In the hidden layers, exponential linear units (ELU) serve as activation functions in the original implementation. For the tests in section 5.2.3, the ELU are replaced by rectified linear unit (ReLU) activation functions.

4. SSD in Floating Point Precision

4.1 SSD-7 with Keras

4.1.1 Network Architecture

In this thesis a stripped down version of the original SSD is used in order to assess if SSD-style networks can be used for embedded object detection with FPGA-hardware. The chosen network architecture is called SSD-7 and was originally implemented by Pierluigi Ferrari[9]. It is written in Keras with TensorFlow as backend.

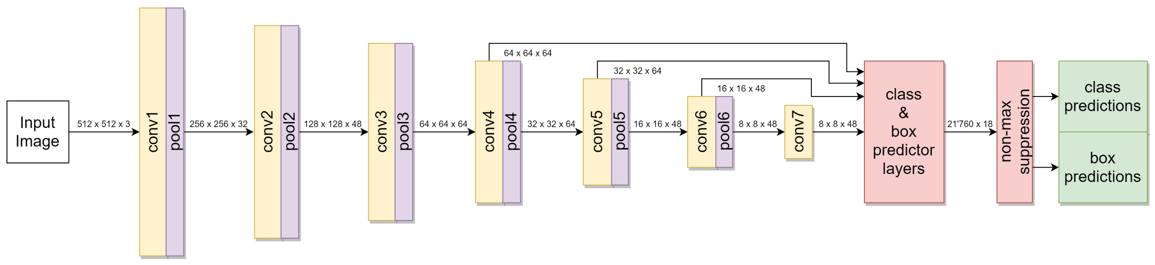


Figure 4.1: SSD-7 architecture

The SSD-7 consists of only seven convolutional layers in the base network and a convolutional predictor layer containing four classes layers and four boxes layers (figures 4.1 and 4.2). It contains 213'232 trainable and 672 non-trainable parameters, so it is fairly small for an object detection network.

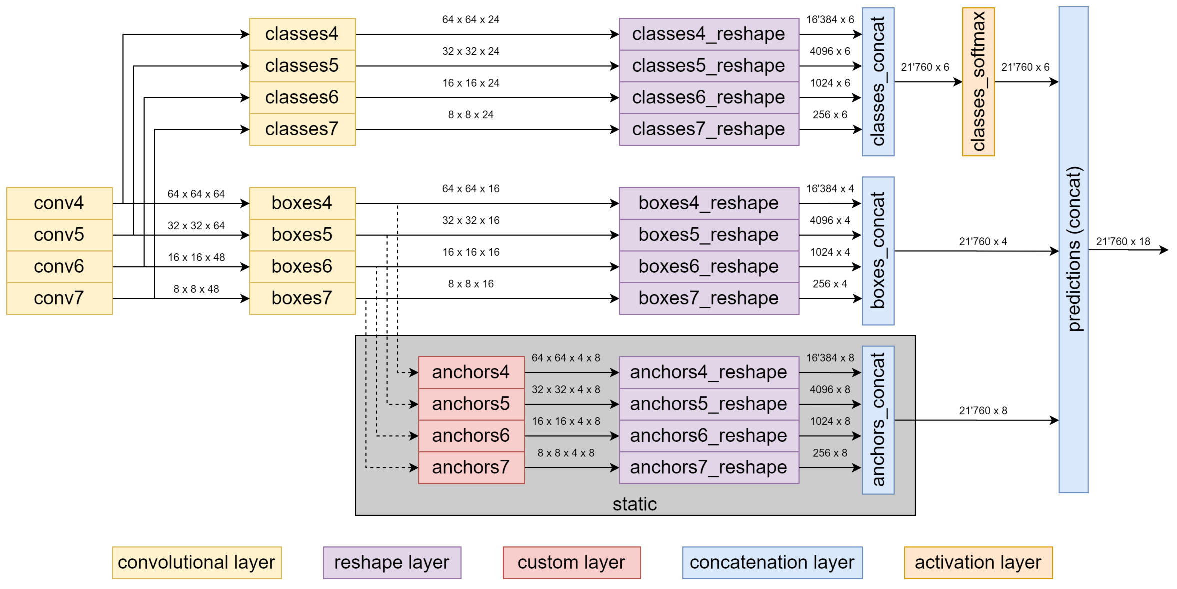


Figure 4.2: Detailed view of the SSD-7 class & box predictor layers

The layers *conv4-7* are directly used for prediction (figure 4.2). The anchors layers in the

grey area only need to be calculated once and remain static as long as there is no change in the network architecture or input image size. The arrows from the box offset tensors only indicate that the anchors layers depend on the size of these tensors, not on the actual data during training or inference.
For every feature tensor passed on to the predictor layers there is a convolutional layer for class prediction and another for the bounding box offsets. There are four corresponding anchor boxes for every value in the feature map, applying the aspect ratios 2, 1, 0.5 and a second, bigger box for the aspect ratio 1. Before being reshaped, the dimensions of the classes tensors correspond to the size of the input feature map with a depth of 24, because there are six classes (including background) for each of the four anchor boxes. This tensor gets reshaped into a list of all the possible anchor boxes with a channel for every class and then concatenated with the tensors of the other classes layers. The same is done with the box offset predictions, but instead of six classes there are four offset values, leading to a depth of 16 before the reshape.
In order to convert the confidence predictions to a value between zero and one, a softmax activation function is applied after the concatenation of the class predictions. After this, the resulting classes tensor is concatenated with the box offset tensor and the static anchors tensor (section 3.1.2). This forms the predictions tensor containing all relevant information about the prediction confidences and prediction box coordinates. The predictions tensor is then passed on to the NMS stage.

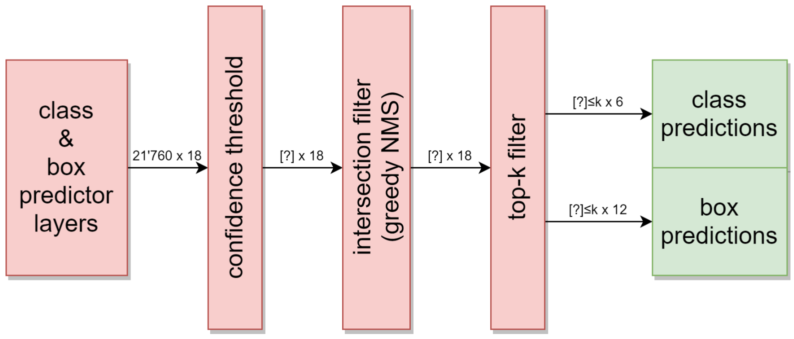


Figure 4.3: Detailed view of the SSD-7 NMS stage

The NMS stage of the SSD-7 can be used with various confidence threshold values. To determine the mAP-score, many different thresholds have to be used. As explained in section 4.3.3, a different threshold for each class has been used in order to get decent results. For the intersection filter, the IoU value has to exceed 0.45 to be regarded as overlapping. Lastly, the top-k filter is set to $k = 200$, ignoring the lowest confidence predictions if there happen to be more than 200 for a certain image.
In figures 4.1, 4.2 and 4.3, the batch size component of the tensor shapes has been omitted to improve readability.

4.1.2 Hyperparameters

Table 4.1 shows the hyperparameter configuration of SSD-7.

Input image dimensions	512 x 512 x 3
Optimizer	Adaptive moment (Adam)
Classification loss function	Logarithmic loss
Localization loss function	Smooth L1 loss
Aspect ratios	[0.5, 1, 2]
Scaling factors	[0.08, 0.16, 0.32, 0.64, 0.96]
Normalize coords	true
NMS: Confidence threshold	various values
NMS: IoU overlap threshold	0.45
NMS: k	200

Table 4.1: Hyperparameters overview

Most of the hyperparameter choices were left as implemented by Pierluigi Ferrari[9] and they correspond to the settings used in the original paper[18], adapted for the smaller version of the network.

4.1.3 Training

The network was trained on Machine 2 (section 1.2.3) due to the better performance. It took about eight minutes per epoch leading to a total of 24 hours training time. After 176 epochs the early stopping mechanism was triggered because there was no more improvement of the performance on the validation set. Some tests with a less sensitive early stopping and therefore more training epochs did not improve the performance. This network state serves as the "original" for all subsequent tests except for the ReLU implementation (section 5.2.3).

# training images	9239
# validation images	4000
Initial learning rate	0.001
# epochs (early stopping)	176

Table 4.2: Training parameters overview

Reduce learning rate on plateau: This function automatically reduces the learning rate, when the training is stuck at a certain validation loss value. In this case, the learning rate was multiplied by a factor of 0.2 whenever there was no progress (<0.001) for eight epochs in a row.

Early stopping: This function interrupts the training when no more decrease of the validation loss value is detected. In this implementation, training stopped automatically when no decrease of the validation loss occurred for 10 epochs in a row.

4.2 Tests and Results

4.2.1 Prediction Observations

Out of the five classes, cars are much better detected by the network than trucks, bicyclists, pedestrians and traffic lights. This is most likely a consequence of the very unbalanced dataset (figure 4.6). Tests proved that the other classes get detected as well, but with a much lower confidence. More on that in section 4.3.3.

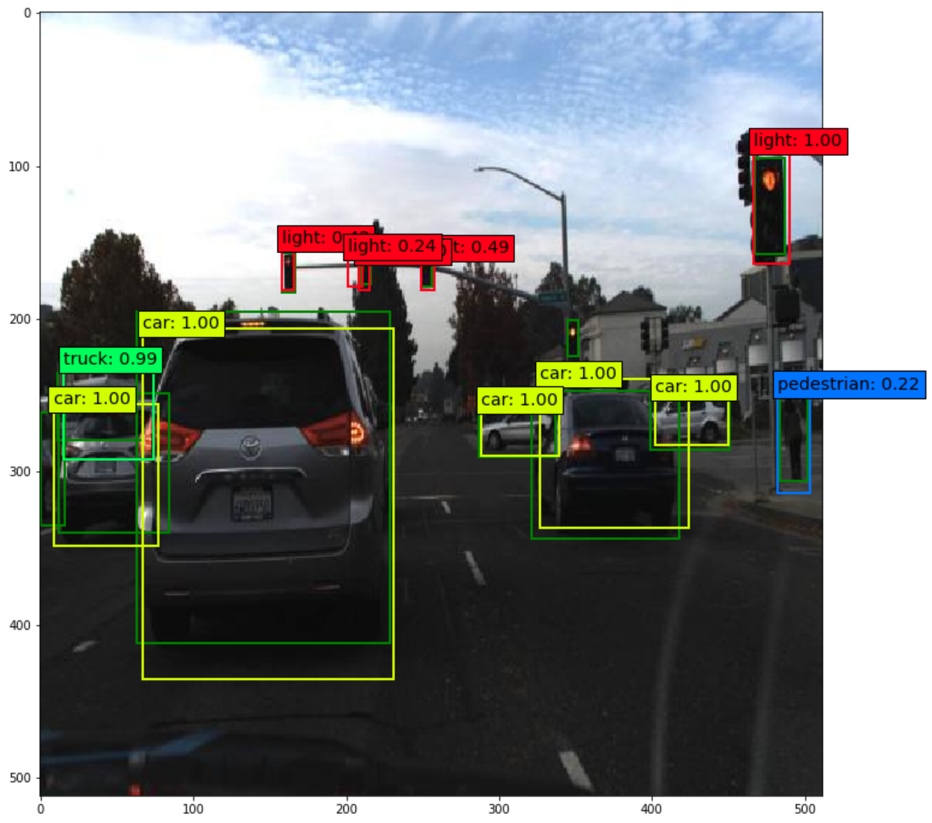


Figure 4.4: Predictions of the SSD-7 on an image of the Udacity[24]/Roboflow[5] dataset

The processing of one image on machine 1 (section 1.2.3) took about 30ms, which corresponds to a rate of approximately 33 frames per second (fps).

4.2.2 Prediction Performance

The performance of the trained network was measured based on Pascal VOC 2012 mAP[8] (section 2.3.7). For this computation the 4000 images of the validation set were used. The IoU threshold is set to 0.5 as specified in Pascal VOC.
To compute the mAP, precision and recall values for each class have been determined at 14 different confidence levels reaching from 0.01 (high recall) to 0.999999 (high precision). From these values the PR curves were generated (figure 4.5).

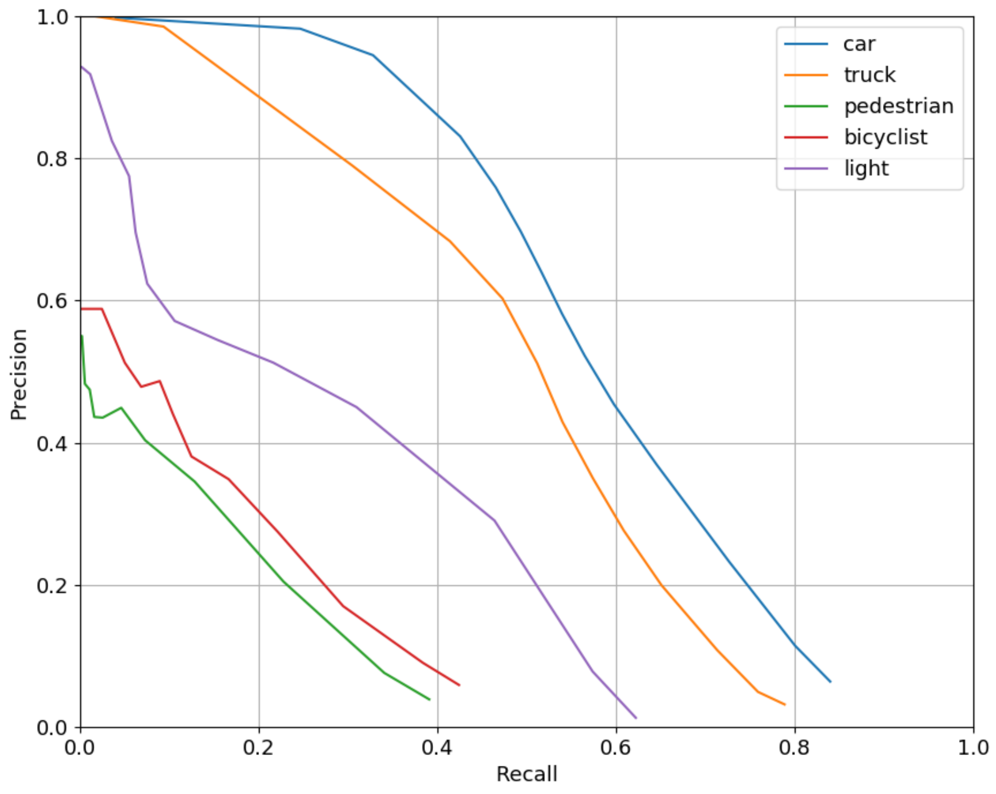


Figure 4.5: Precision-recall curves of all classes

car	truck	pedestrian	bicyclist	light	**mAP**
0.578	0.481	0.096	0.123	0.261	**0.308**

Table 4.3: Average precision values of the different classes and resulting mAP

The average precision (AP) of each class corresponds to the area under the PR curve. The mean of these AP-values is the performance metric mAP (table 4.3).

4.3 Interpretation

4.3.1 Achieved Performance

The achieved performance of mAP=0.308 is far from the performance of the best state-of-the-art networks tested on the Pascal VOC dataset, which achieve mAP values of 0.8 and higher. On the more complex datasets such as COCO, even the winning detectors hardly exceed 0.5, most of them only achieve about 0.25[14]. This shows that the mAP does strongly depend on the dataset. The Udacity/Roboflow dataset is rather simple, consisting of only five different classes, and the pictures are always traffic situations. This suggests that a high mAP value could be achieved. However, a glimpse into the dataset shows that, even though it has already been improved by Roboflow, there are still many inaccurate, missing, wrong or redundant labels in the dataset. Furthermore, many labels mark cars and other object in great distance, which are, on the downscaled image, hard to recognize even for a human. All this together makes it very difficult for an object detector trained and tested on this dataset to achieve a high mAP.
That some of the best object detectors score only about 0.25 on the COCO dataset also shows that this is already a decent performance. Compared to a classification task with five classes, where

random guessing already achieves an accuracy of 0.2, object detection is much harder. Random guessing of bounding boxes and classes would not provide many correct predictions. To achieve an mAP of 0.3 the detector needs to recognize the features quite well in order to correctly predict the class and position of many objects (figure 4.4).
The performance of the SSD-7, which is a much simpler and computationally less expensive network than the state-of-the-art, will most likely be sufficient for some applications in industry and robotics. However, it should not be used for safety-critical applications like autonomous driving or other tasks which require very accurate detections.

4.3.2 Class Performance Difference

The results of the AP-tests reveal big differences in the performance for the five classes (table 4.3). This is caused primarily by the non-balanced nature of the used dataset (figure 4.6).

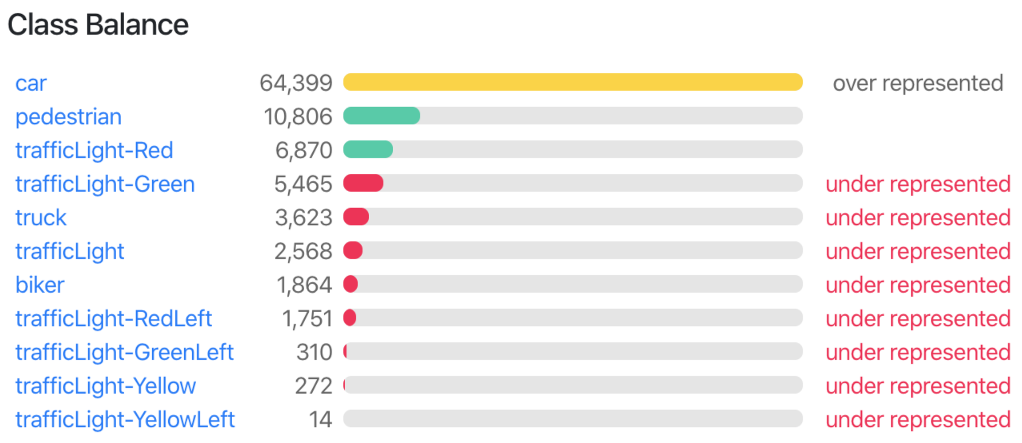


Figure 4.6: Class balance of the Udacity self-driving car dataset[5]

The under-representation of some classes leads to very low prediction confidences. To address this problem, a class-dependent confidence threshold was implemented (section 4.3.3). Due to the fact that the mAP-metric includes predictions at many different confidence levels, this improvement has no effect on the achieved mAP score. In this thesis, all traffic light classes distinguished in figure 4.6 are treated as one to keep the number of classes small and because there is not enough data for some of these classes.
With an AP score of 0.578, the object detector works best for detecting cars, followed by 0.481 for trucks (table 4.3). In the dataset, by far the most occurring class is cars, so the network has the most examples of cars to learn from. Because trucks have very similar optical features as cars, they get detected almost as well. In contrast, the algorithm has the most problems detecting pedestrians (AP: 0.096) and bicyclists (AP: 0.123). Even though there are more than 10'000 labeled pedestrians in the dataset, a glimpse in the images shows that they are sometimes very hard to see even for a human. This is caused by the down-scaled resolution of the images as well as bad light situations and partially hidden locations. Furthermore, to differentiate between pedestrians and bicyclists is obviously very challenging for the network, because in many cases the bicycles are very inconspicuous.
The AP scores show that the network does not reach a level to be applied in the area of self-driving cars, but considering the very small and simple architecture it does a decent job, especially in detecting cars and trucks.

4.3.3 Class-Dependent Confidence Thresholds

The original code used the same confidence threshold to suppress improbable detection boxes for every class. When using a high confidence threshold (>0.5) almost only cars are detected. Lowering the confidence threshold (0.1) revealed that there are many correct detections of other classes than cars, but with lower confidence. To counteract this, a class-dependent confidence threshold is introduced. The values for these class-dependent thresholds were found in an empirical way by choosing a value slightly above the confidence of most wrong detections. Classes appearing less in the dataset generally get lower confidences, even though the respective predictions are often correct.

	car	truck	pedestrian	bicyclist	light
confidence threshold	0.8	0.7	0.2	0.15	0.2

Table 4.4: Applied confidence thresholds for each class

An even better way to determine the ideal confidence threshold for every class would be to derive it from the PR curve (figure 4.5). One possibility is to determine the datapoint with the maximum value for $precision * recall$ and take the respective confidence threshold. This method achieves a good balance between precision and recall. If the application needs a high recall, because no object should be missed, but some wrong detections do not matter much, a value with high recall can be chosen. Vice-versa, if every prediction needs to be true, but some missing predictions are acceptable, a value with high precision is suitable.
For the computation of the mAP of an algorithm, this choice of confidence threshold has no relevance because multiple confidence thresholds are applied. However, when applying the object detector to a certain task this choice is very important.

5. SSD for Embedded Systems

5.1 Binary Approximated Neural Networks

5.1.1 Choosing the Number of Binary Filters

To run this object detector on an embedded hardware it is converted into a form which is suited for FPGAs. The BinArray[11] method is applied to ensure that the weight tensors are converted to a suitable form for the HA. In this form, the weight tensors of the network layers are approximated by a linear combination of M binary tensors. The larger M is chosen, the more accurate the approximation gets. The downside of a large M is the increased hardware consumption. The main benefit of choosing M small is the possibility to stack them in the FPGA due to low hardware usage. This allows a parallelization which greatly increases the computation speed of the inference. The goal when analyzing the binary approximated convolutional neural network (BACNN) is to find the minimal M for which the prediction performance (in this thesis measured by the mAP) is sufficiently close to the performance achieved by the original CNN.

5.1.2 Binary Retraining

In preceding works such as [11], binary retraining has proven to be a very effective method to improve the performance of the binary approximated network. Binary retraining means training the binary approximated network for some more epochs on the dataset to adjust the approximated weights and therefore reduce the decrease in performance, especially for small M. Like the training of the floating point SSD, retraining of the BA-SSD was executed on machine 2.

5.1.3 Change of Activation Function

SSD-7 uses ELU as activation functions for the convolutional layers, except for one softmax layer to normalize the classification confidence to a value between zero and one. The fact that the ELU requires much more computational effort indicates that it would be better to use another activation function instead. The most efficient activation function would be the ReLU, which has proven to be very effective for neural networks[1]. Furthermore, the hardware accelerator framework only supports ReLU in the current state.

5.2 Tests and Results

5.2.1 Without Retraining

To determine how much the mAP drops when the CNN is converted to a binary form and to estimate the optimal choice for M, seven BA-SSDs with values $M = [2, 3, 4, 5, 6, 7, 8]$ are generated and the respective mAP computed.

M	2	3	4	5	6	7	8	original
mAP	0.005	0.045	0.147	0.264	0.294	0.286	0.295	0.308

Table 5.1: Performance (mAP) of the BA-SSDs without retraining

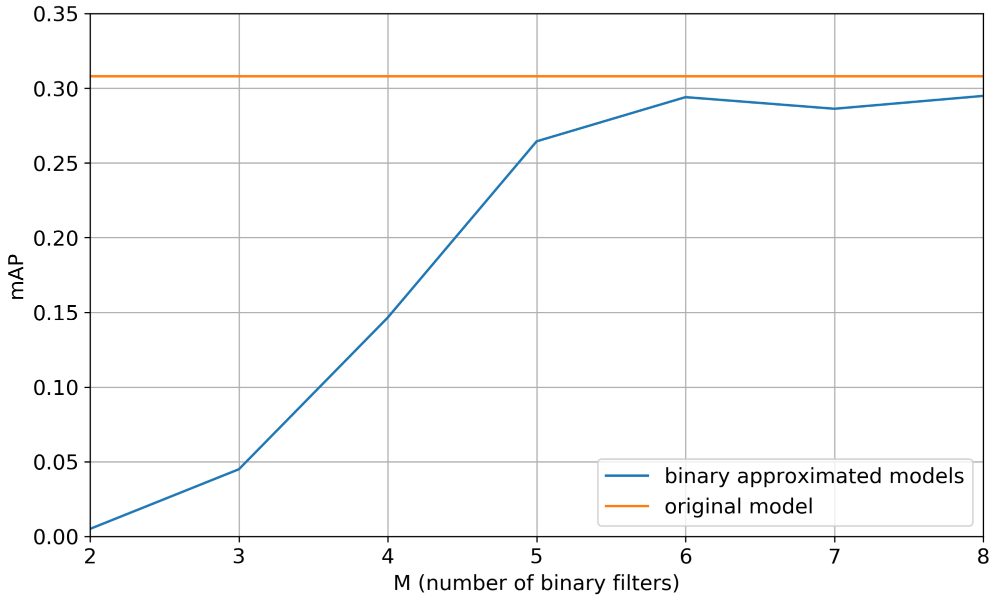


Figure 5.1: Performance (mAP) of the BA-SSDs as a function of M

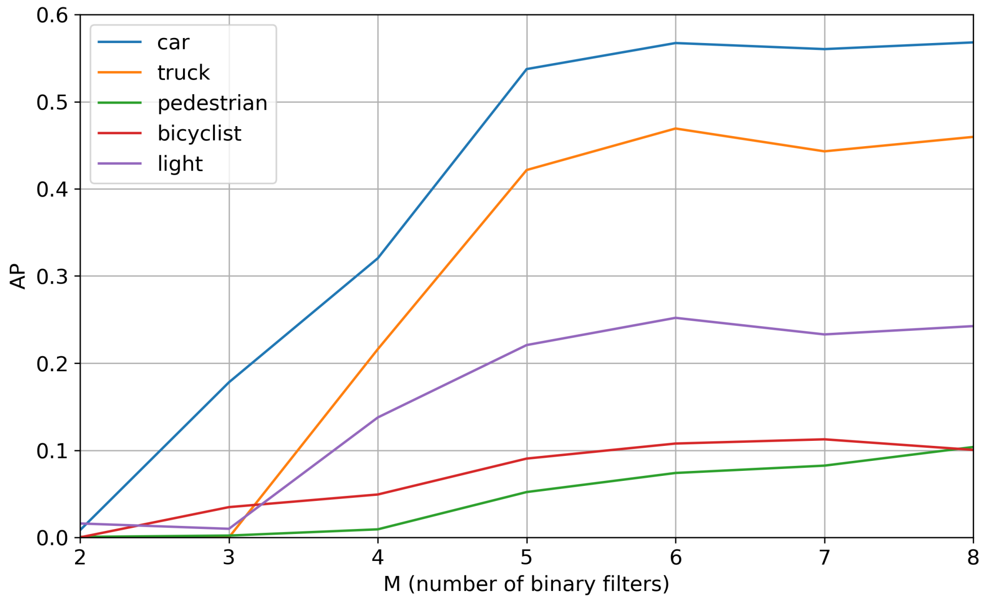


Figure 5.2: AP values of the different classes as a function of M

5.2.2 With Retraining

To assess the effectiveness of retraining for this network, all the BA-SSDs with $M = [2, 3, 4, 5, 6, 7, 8]$ are retrained for 5, 10, 25 and 50 epochs. For the retraining, the initial learning rate was reduced from 0.001 to 0.0001 to prevent the learning algorithm from getting stuck. This is a manual way to correct for the learning rate reduction already applied when training the original network (section 4.1.3).

M	2	3	4	5	6	7	8
no retraining	0.005	0.045	0.147	0.264	0.294	0.286	0.295
5 epochs	0.191	0.272	0.293	0.294	0.296	0.296	0.295
10 epochs	0.204	0.276	0.294	0.300	0.301	0.297	0.299
25 epochs	0.216	0.274	0.301	0.299	0.307	0.301	0.306
50 epochs	0.255	0.284	0.303	0.299	0.308	0.306	0.307

Table 5.2: Performance (mAP) of the BA-SSDs with retraining

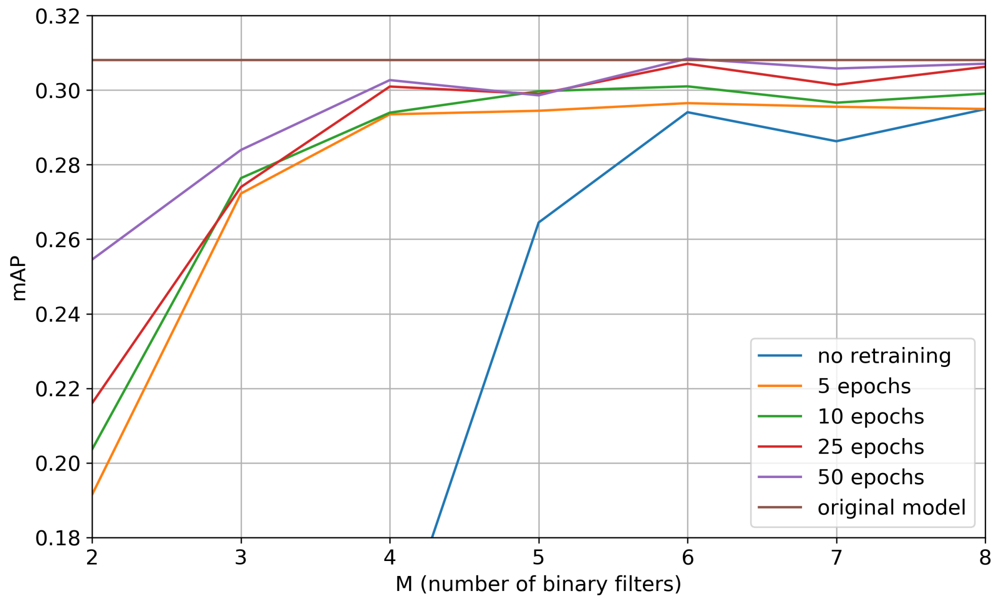


Figure 5.3: Performance (mAP) of the retrained BA-SSDs as a function of M

5.2.3 ReLU as Activation Function

To investigate the performance change, the same network has been trained using ReLU instead of ELU activation functions.

car	truck	pedestrian	bicyclist	light	**mAP**
0.589	0.505	0.093	0.106	0.253	**0.309**

Table 5.3: Average precision values of the different classes and resulting mAP with ReLU

With ReLU as activation functions the algorithm in floating point precision achieved an even higher mAP of 0.309 (table 5.3) than with ELU as implemented originally (table 4.3). The results of the binary approximated models using ReLU are shown in table 5.4 and figure 5.4.

M	2	3	4	5	6	7	8
no retraining	0.015	0.146	0.268	0.273	0.304	0.306	0.308
5 epochs	0.214	0.280	0.295	0.294	0.307	0.308	0.299
10 epochs	0.221	0.287	0.294	0.300	0.301	0.297	0.299
25 epochs	0.216	0.274	0.301	0.299	0.307	0.301	0.306
50 epochs	0.255	0.284	0.303	0.299	0.308	0.306	0.307

Table 5.4: Performance (mAP) of the retrained BA-SSDs with ReLU activation functions

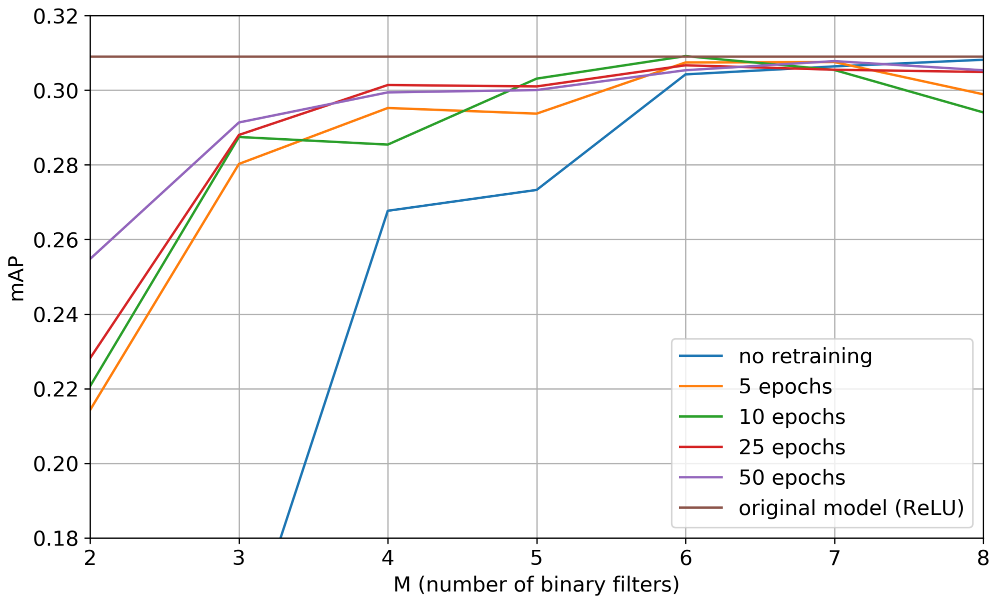


Figure 5.4: Performance (mAP) of the retrained BA-SSDs as a function of M with ReLU activation functions

5.3 Interpretation

5.3.1 Without Retraining

Figure 5.1 shows the performance in mAP of the seven BACNN models without retraining. With $M = 6$ the performance of the BACNN ($mAP_{M=6} = 0.294$) is close to the performance of the original CNN ($mAP_{original} = 0.308$). Even for $M = 5$ ($mAP_{M=5} = 0.264$) the reduction is rather small, but still significant.

5.3.2 With Retraining

Figure 5.3 shows that retraining efficiently improves the performance of the binary approximated models. All the retrained models reach a higher mAP than the approximated model without retraining. $mAP_{M=6} = 0.308$ and $mAP_{M=8} = 0.307$ still give the best results, but $mAP_{M=4} = 0.303$ achieves almost the same performance. The minimum acceptable mAP depends on the application. For the hardware implementation estimations in chapter 6, the number of binary filters is set to $M = 4$ because a decrease in mAP of 0.005 is acceptable for this project.

5.3.3 ReLU as Activation Function

Figure 5.4, where the network performance is measured with ReLU as activation function, shows even slightly better results than the original network with ELU (figures 5.3 and 5.4). This indicates that the network will perform as well with ReLU as activation function, which has a lower computational complexity and is already implemented in the Low-Cost CNN Accelerator framework.

6. Hardware Implementation

6.1 Hardware Accelerator Concept

To achieve the goal of executing an object detector algorithm on an embedded system with a speed that allows real-time applications, a combination of processing system (PS), in this case an ARM-CPU, and hardware accelerator (HA) implemented on a programmable logic (PL) is used. The PS manages the input data stream (e.g. a video stream from a camera) as well as the output data, the NMS stage and the control of the HA[11]. The PL computes all the convolutional layers and is able to access the memory directly to fetch the required data and write the results back.

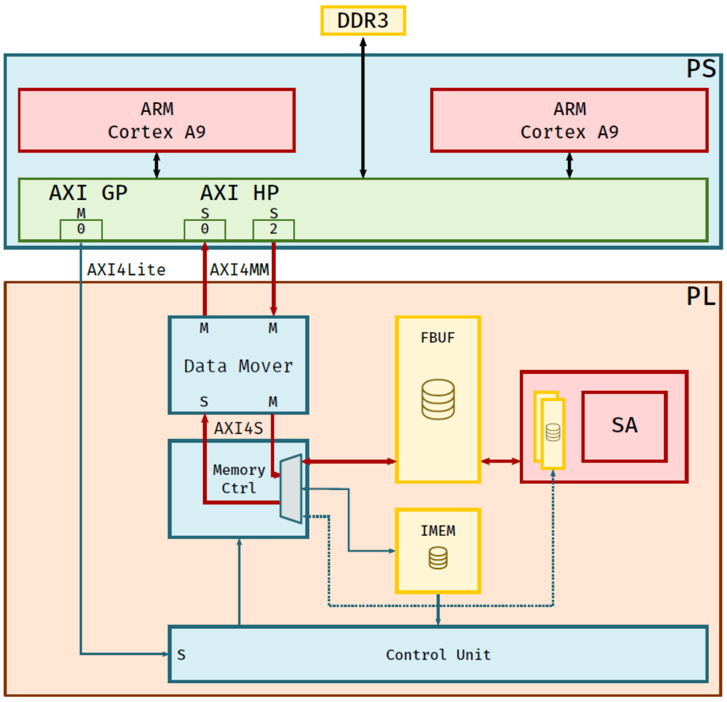


Figure 6.1: BinArray System: a combination of PS (ARM-CPU) and HA implemented on a PL (image source: [11])

An AXI bus connects the PS and the PL, enabling the PS to send instructions to the HA. The data mover of the HA has direct memory access (DMA). This means it is directly connected to AXI bus as master and can therefore access the external memory (DDR3) without requiring computation time of the CPU. This is the architecture of the target hardware (Xilinx Zynq-7000) introduced in section 1.2.4.
The systolic array (SA) consists of a collection of processing elements (PEs). The PEs are arranged in 2-dimensional grid with height D_{arch} and width M_{arch}. The size of the SA must be selected to match the network architecture. If M_{arch} is smaller than the number of binary arrays used for approximation (M), it will need multiple systolic unit cycles (SUC) to compute a certain layer. In this thesis M_{arch} is chosen to be equal to M in order to maximize performance. The SA height

D_{arch} determines how many convolution filters can be applied simultaneously. If D_{arch} is equal to or greater than the number of filters, only one SUC is need for the computation. The highest number of filters in the SSD-7 network is 64 and the lowest is 16, accordingly it is not possible to increase the performance by choosing D_{arch} lower than 16 or higher than 64. Because of the lower hardware consumption, a small D_{arch} allows more parallel SAs implemented in the HA. The downside of choosing D_{arch} to be low is the trade-off in speed, because there are multiple SUCs need for some layers.

6.2 Hardware Constraints

On every FPGA chip only a limited number of logical components can be implemented. These basic components are block RAM (BRAM), digital signal processor (DSP), flip-flop (FF) and lookup table (LUT). As explained in section 1.2.4, the Xilinx XC7Z045 and XC7Z010 are used in this thesis.

Resource Type	**XC7Z045** (Mid-Range)	**XC7C010** (Low-End)	Fraction [%]
BRAM	545	60	11.01
DSP	900	80	8.89
FF	437200	35200	8.05
LUT	218600	17600	8.05

Table 6.1: Available hardware resources of a mid-range (XC7Z045) and a low-end (XC7Z010) FPGA

These numbers originate from the official datasheet provided by Xilinx (appendix, section 12.2).

6.3 Hardware Consumption Estimation

To find the optimal configuration, the resource requirements for five different values of D_{arch} are estimated. With the information about the available resources in the target hardware (table 6.1) the maximum number of SA implementations N_{SA-max} is determined. Formulas from [16] are used to compute the hardware consumption of an SA depending on the configuration of M_{arch} and D_{arch}. BRAM and DSP only depend on M_{arch} and therefore remain constant when M_{arch} is left unchanged.

D_{arch}	M_{arch}	BRAM [%]	DSP [%]	FF [%]	LUT [%]	N_{SA-max} (Mid-Range)
16	4	0.73	0.44	0.89	2.77	36
24	4	0.73	0.44	1.13	3.97	25
32	4	0.73	0.44	1.37	5.17	19
48	4	0.73	0.44	1.86	7.57	13
64	4	0.73	0.44	2.34	9.98	10

Table 6.2: Hardware usage of the mid-range FPGA (XC7Z045) for different SA heights D_{arch}

More detailed tables of these estimations are provided in the appendix, section 12.2. The estimations do not include the additional BRAM consumption from the FBUF implementation (section 6.4.2).

D_{arch}	M_{arch}	BRAM [%]	DSP [%]	FF [%]	LUT [%]	N_{SA-max} (Low-End)
16	4	6.67	5.00	11.07	34.44	2
24	4	6.67	5.00	14.07	49.35	2
32	4	6.67	5.00	17.07	64.26	1
48	4	6.67	5.00	23.07	94.08	1
64	4	6.67	5.00	29.07	*123.90*	0

Table 6.3: Hardware usage of the low-end FPGA (XC7Z010) for different SA heights D_{arch}

6.4 Inference Speed Estimation

6.4.1 Hardware Accelerator

Table 6.2 shows the estimated hardware usage of one systolic array relative to the available resources on the mid-range hardware. The lower this percentage is, the more SAs can be implemented to work in parallel (N_{SA}) and therefore speed up computation. The results show that LUTs are the limiting resource in this case and therefore define how many SA can be implemented. The formulas used to estimate the inference speed on the HA are derived from [10].

		Inferences / s		
D_{arch}	M_{arch}	$N_{SA} = 1$	N_{SA-max} (Mid-Range)	N_{SA-max} (Low-End)
16	4	2.76	**99.23**	4.86
24	4	3.54	85.15	**6.54**
32	4	4.77	87.24	4.77
48	4	6.22	78.96	6.22
64	4	7.29	72.93	-

Table 6.4: Resulting inference speed for different SA heights D_{arch} with one SA as well as with the maximum number of SA to fit into the mid-range (XC7Z045) or low-end (XC7Z010) FPGA

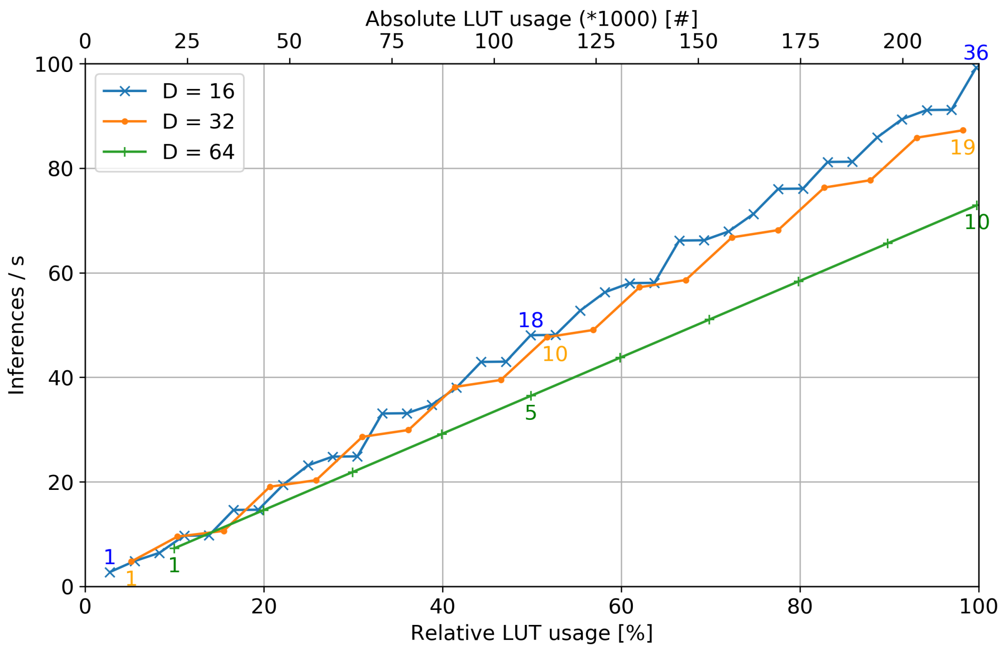


Figure 6.2: Inference speed as a function of the relative hardware usage (LUTs) of the mid-range hardware (XC7Z045), for three SA height values D_{arch} when N_{SA} (coloured numbers) is increased from one to N_{SA-max}

In figure 6.2 the three coloured graphs show the relation between inference speed and LUT usage for different values of D_{arch}. Every marked point on the graph from left to right represents one more implementation of an SA (e.g. N_{SA}) up to the maximum possible. The x-axis shows the LUTs used as a percentage of the LUTs available in the mid-range hardware (XC7Z045).
When working with the chosen target mid-range hardware, this figure is important when deciding which inference speed needs to be achieved and how many hardware resources have to be utilized. It also shows which values have to be chosen for D_{arch} and N_{SA}. Table 6.2 shows that there are plenty of the other hardware resources (FF, DSP, BRAM) available to implement other applications on the PL.
Figure 6.3 holds the relevant information to determine the possible inference speed from the available hardware resources in absolute numbers. Section 7.2 explains how to apply this by describing concrete examples.

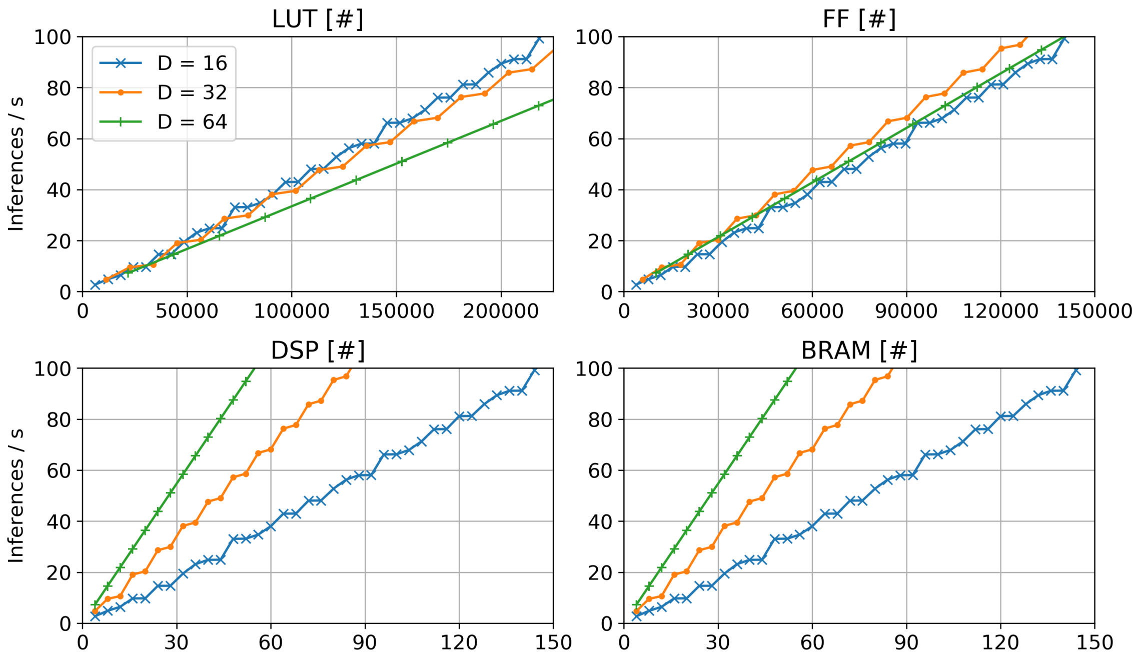


Figure 6.3: Inference speed as a function of absolute hardware usage, split into the four basic hardware resources

For every combination of compatible CNN and FPGA these values can be computed, allowing the user to accurately choose the amount of hardware resources to give to the accelerator to achieve a required inference rate. This figure does not include the additional BRAM consumption for the implementation of the FBUFs.

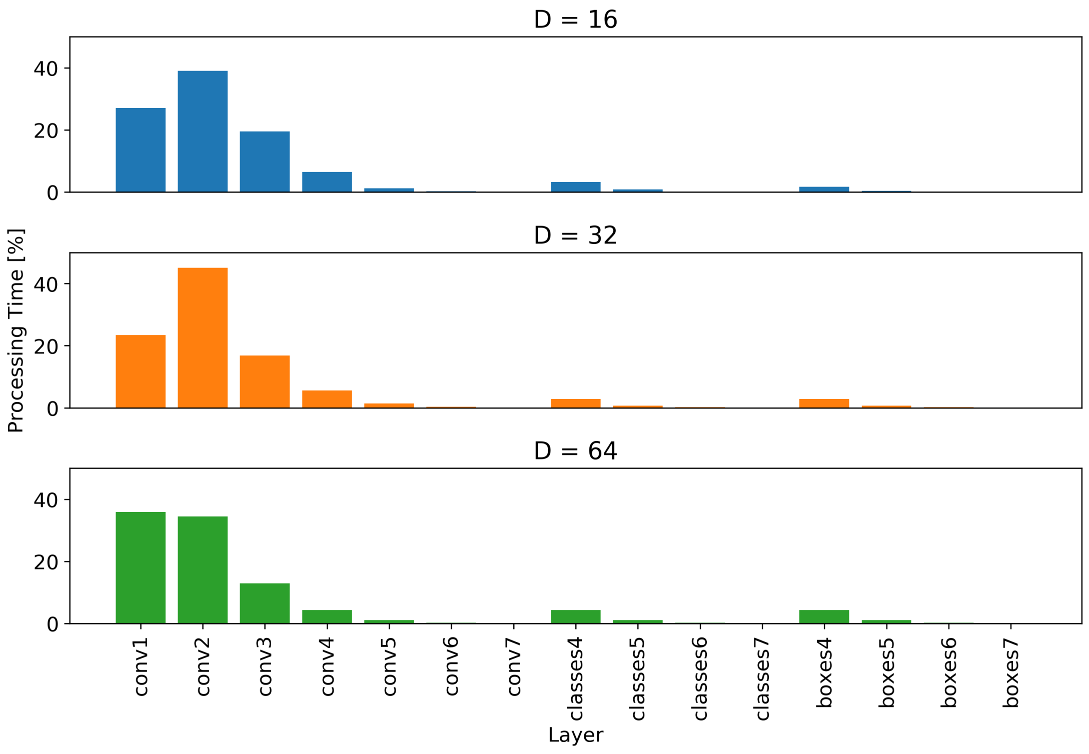


Figure 6.4: Processing time of the HA per convolutional layer for D_{arch} values 16, 32 and 64

To investigate the impact of each layer on the processing time as well as the effect of different D_{arch} values, figure 6.4 provides a graphical overview. It shows that most of the processing time (>66%) is consumed by the first two convolutional layers. This is caused by the large dimensions of the involved feature tensors. Furthermore the ability of higher values of D_{arch} to process layers with up to 64 filters simultaneously can be observed. This accelerates for example the computation of the layers *conv2* and *conv3* with 48 respectively 64 filters. Optimization efforts will be much more effective on the early layers. The later layers have little influence on the inference speed.

6.4.2 Feature Buffers and Memory Access

In many cases, memory access can be an issue for real-time applications because it is rather slow compared to other operations. To prevent the need for writing the feature tensor back to the memory after every layer, the HA provides feature buffers (FBUF) to store the feature tensors close to the SAs, where they have fast access to it[11].
There are two FBUFs in the HA. The first one is the global FBUF, which is implemented as ping-pong buffer. This means it holds the input as well as the output tensor and can be accessed simultaneously. To be able to hold both tensors it needs to provide enough memory. Through the control unit, the PS tells the HA which data to load. The data mover then accesses the DDR3-memory and loads the image or feature tensor into the global FBUF. The data mover has direct access to the memory via the AXI-bus and does not require CPU time for this operation. The ping-pong buffer (global FBUF) allows the data mover to fetch the next image while the SAs are busy. The SAs write the result back to the FBUF without interfering with the data mover already filling in the next tensor. To store the values which the SAs working in parallel need to perform their operations, there are local FBUFs embedded in every SA. The feature tensor to be processed is split into various tiles and distributed among all the SAs used for the operation.
The highest consumption of FBUF memory occurs when the SAs have finished computing the first convolutional layer and store the resulting feature tensor T_1 in the global FBUF. Then the global FBUF holds the input and the other part of the global FBUF holds T_1. All the local FBUFs together also hold T_1. The dimensions of the feature tensors are shown in figure 4.1. Every data

element in the network consumes 1 byte.

$$size(Input\,Image) = 512 * 512 * 3\,B = 786.432\,kB \tag{6.1}$$

$$size(T_1) = 64 * 64 * 32\,B = 2.097\,MB \tag{6.2}$$

$$Maximum\,FBUF\,memory\,consumption = size(Input\,Image) + 2 * size(T_1) = 4.98\,MB \tag{6.3}$$

The mid-range FPGA has enough BRAM to store up to 2.4 MB (19.2 Mb) (appendix, section 12.2). In the current architecture of the HA this is not sufficient. For the low-end FPGA with only 262.5 kB (2.1Mb) it is even more problematic. To be able to accelerate a network containing tensors this big, a strategy to split the processing into smaller tasks has to be integrated. For this to work, there could be an overhead of time for memory access. The extent of this additional computation time is assessed in section 6.4.3.
Because in the SSD architecture some feature maps bypass other convolutional layers, they need to remain in the FBUF. The maximum amount of occupied FBUF memory by these additional connections can be calculated from the sizes of the feature tensors (equation 6.4). The detailed equations are in the appendix, section 12.1.

$$size(T_4) + size(T_5) + size(T_6) + size(T_7) = 343.04\,kB \tag{6.4}$$

This is much less than the maximum usage calculated in equation 6.3 and should fit into the available BRAM of the mid-range, but not in the low-end FPGA. To store the feature tensors for some cycles and process them afterwards, the HA architecture has to be adjusted. At the moment it only supports direct reuse of the stored tensor for the next computation.

6.4.3 Reducing Memory Requirement: Tiling

To address the problem discussed in section 6.4.2, the biggest feature tensor has to be tiled into four smaller tensors. This ensures, that there is sufficient BRAM on the mid-range hardware to store the results and inputs of the affected layers *conv1* and *conv2*. As a consequence of this tiling, the layers *conv1* and *conv2* have to be executed four times instead of once. Because the tensors to be processed are four times smaller, there is no relevant computation overhead for the SAs (appendix, section 12.2). The overhead occurring due to overlap sections at the edge of the tiles is negligibly small. Some additional time consumption could occur due to the need to split the image, read every tile separately to the FBUF and write the results back to the memory. The extent of this additional time depends largely on the speed of the data mover and the AXI bus. The ping-pong design of the FBUF enables the data mover to load and store data while the SAs are busy. Because the problematic layers are also the ones which take longest to process in the SAs (figure 6.4), the data mover has more time for its task. If the load and store operations can be executed in the same or less time than the SAs need to process the data, there is no remarkable decrease in inference speed for the tiling.

6.4.4 CPU Workload

Even though most of the necessary operations can be done by the HA, some tasks need to be handled by the CPU. While the HA can execute all the convolutional layers including the classes and boxes layers, the PS needs to manage the input images and provide the memory addresses to the HA, send computation instructions to the control unit of the HA and deal with the output of the last convolutional layers of the network.
The layers after the convolutional layers are mostly reshape and concatenation layers (figure 4.2). Because there is no computation involved in these layers, by accessing the memory in a certain pattern this can be done without any additional clock cycles. To normalize the class prediction into a confidence value between zero and one, a softmax layer is needed. The softmax requires some CPU time. If this operation turns out to be too computationally expensive, this cost could be decreased by filtering the class predictions before applying the softmax. A possible way to implement this filter is by comparing the raw class prediction values to the value of the background class.

For example, if the value for the background class is more than twice as big as the highest class prediction value, the prediction is discarded. This comparison would require much less processing than applying the softmax to all the 21'760 predictions.
At the end of the network, the best prediction boxes are selected by the NMS stage. Of all the 21'760 possible boxes only those with a high confidence value contain relevant information, so only the predicted boxes with a confidence value higher than a chosen confidence threshold are passed on. These predictions will still contain multiple overlapping boxes detecting the same object, therefore all boxes with an IoU greater than a chosen IoU threshold (0.45) with a higher-confidence prediction box are discarded.
Finally, the predicted bounding box offsets need to be converted to prediction boxes using the constant anchors tensor. This gives relative coordinates which then have to be transformed into absolute coordinates in pixels. For every single one of the 21'760 predictions exists a corresponding anchor box which can be computed from the structure of the network. Every prediction further delivers a set of four offset values: center coordinate offsets as well as width and height offsets[18]. With the four coordinates of the anchor box, the four offset values and the resolution of the input image, the absolute coordinates of the predicted box can now be determined.

$$cx_{prediction,rel} = \Delta_{cx} * w_{anchor} + cx_{anchor} \tag{6.5}$$

$$cy_{prediction,rel} = \Delta_{cy} * h_{anchor} + cy_{anchor} \tag{6.6}$$

$$w_{prediction,rel} = e^{\Delta_w} * w_{anchor} \tag{6.7}$$

$$h_{prediction,rel} = e^{\Delta_h} * h_{anchor} \tag{6.8}$$

$$[cx_{abs}, cy_{abs}, w_{abs}, h_{abs}] = [cx_{rel}, cy_{rel}, w_{rel}, h_{rel}] \circ [x_{image}, y_{image}, x_{image}, y_{image}] \tag{6.9}$$

To get the relative coordinates of a predicted bounding box with the offset taken into account, four multiplications, two additions and two exponential functions are required (equations 6.5 - 6.8). To convert the relative to absolute coordinates, four more multiplications have to be carried out (equation 6.9).
To suppress overlapping boxes in the NMS stage, the IoU value of two boxes have to be calculated using the following equations:

$$w_{Intersection} = min(cx_1 + \frac{w_1}{2}, cx_2 + \frac{w_2}{2}) - max(cx_1 - \frac{w_1}{2}, cx_2 - \frac{w_2}{2}) \tag{6.10}$$

$$h_{Intersection} = min(cy_1 + \frac{h_1}{2}, cy_2 + \frac{h_2}{2}) - max(cy_1 - \frac{h_1}{2}, cy_2 - \frac{h_2}{2}) \tag{6.11}$$

$$A_{Intersection} = w_{Intersection} * h_{Intersection} \tag{6.12}$$

$$A_{Union} = w_1 * h_1 + w_2 * h_2 - A_{Intersection} \tag{6.13}$$

$$IoU = \frac{A_{Intersection}}{A_{Union}} \tag{6.14}$$

If the division by two is executed as shift operation, the computational cost of an IoU is 12 additions/subtractions, eight shift operations, four min/max operations, three multiplications and one division. With data reuse, such as keeping the area of the box with higher confidence, some of these operations could be avoided.
To determine if this is a bottleneck for the inference speed, a worst-case scenario is assumed. If there are 20 objects in the image, and for each object there are 20 overlapping boxes with a confidence higher than the confidence threshold, 400 boxes have to be processed. All the boxes predict the same class, so every overlap has to be taken into account. Converting the box offsets of 400 boxes into relative coordinates requires 800 exponential operations, 800 additions and 1600 multiplications.
To find the globally maximum box, a greedy algorithm is applied in the NMS stage. This algorithm compares a prediction to all the other remaining predictions of the same class, always keeping the one with maximum confidence and discarding those with lower confidence but a too high IoU value. The first prediction has to be compared to all other predictions. This step reveals the first local

maximum prediction and discards 19 overlapping predictions. The next prediction is found after the computation of 380 IoU values and so on.

$$\#\, IoU\ operations = \sum_{i=0}^{19} 400 - 20 * i = 4200 \tag{6.15}$$

This results in 4200 IoU computations. To achieve this, another 50'400 additions, 33'600 shift operations, 25'200 min/max operations, 12'600 multiplications and 4200 divisions have to be computed. For every IoU computation the greedy algorithm executes one min/max operation to compare the IoU values and another to compare the confidences, resulting in 8400 more min/max decisions. The additional computations to choose the next bounding boxes and store the maximum predictions at the end are neglected in this estimation. To convert the relative coordinates of the remaining 20 predictions into absolute coordinates, 80 multiplications are needed. Table 6.5 shows the overall required operations per image for this worst-case scenario.

operation	conf. threshold	rel. coords	intersection filter	abs. coords	total
additions	0	800	50'400	0	51'200
shift operations	0	0	33'600	0	33'600
min/max operations	21'760	0	25'200	0	46'960
multiplications	0	1'600	12'600	80	14'280
divisions	0	0	4'200	0	4'200
exp. operations	0	800	0	0	800

Table 6.5: CPU operations per image in a worst-case scenario

$$CPU time per Inference = (51'200+33'600+46'960)*1ns+(4200+800)*10ns = 181.76\mu s \tag{6.16}$$

$$HA\, time\, per\ Inference = \frac{1}{99.23}\, s = 10.08ms \tag{6.17}$$

$$CPU\ bottleneck\ margin\ (relative) = \frac{PL\, time\, per\ Inference}{CPU\ time\, per\ Inference} = 55.46 \tag{6.18}$$

Equation 6.16 shows the estimated computation time for all these operations on a CPU with 1 GHz clock frequency. This is the speed of the ARM Cortex embedded in the target hardware. For this estimation it is assumed that the basic operations (addition/subtraction, shift, min/max and multiplication) can be computed in one clock cycle (1ns) and the more complex operations (division and exponential) in ten clock cycles (10ns). The operations to discard images (e.g. change a pointer in a list) and to load the prediction data from the memory are neglected. If the resulting computation time is longer than the time the HA needs for one inference, the NMS computations on the CPU become a bottleneck.
Equation 6.17 shows the minimal time the HA requires when implemented on the mid-range hardware and equation 6.18 the resulting relative margin.

6.5 Interpretation

6.5.1 Real-Time Capability

The question if this system is capable of achieving an adequate inference speed for real-time applications is of central importance. Depending on the application, the minimum inference speed necessary can be very different. Nevertheless, it can be said that the results (table 6.4), indicating a possible inference speed of up to 99.23 inferences per second, will most likely be enough for most tasks. If a lower image processing speed is sufficient, N_{SA} can be set lower to reduce hardware usage on the FPGA allowing to choose a cheaper hardware.

6.5.2 Mid-Range vs. Low-End

The results show a remarkable difference in inference speed between the two target hardware. Due to the big gap in available resources (table 6.1), the maximum number of parallel implementations N_{SA} can only be one or two. For $D_{arch} = 64$ not even a single implementation is possible because the hardware does not provide enough LUTs.
With a maximum possible inference speed of 6.54 inferences per second (table 6.4) on the low-end target hardware, this can still be useful for many applications for which reaction time is not as critical.

6.5.3 Feature Buffer Bottleneck

Because the low-end hardware provides less BRAM than the mid-range, the tiling operation outlined in section 6.4.3 must be applied with even more tiles. This increases the number of memory operations further, which could limit the inference speed. The time consumption of memory operations needs to be analyzed and compared to time the SAs need to compute the respective layer (section 7.4.2).

6.5.4 CPU Workload Bottleneck

The result of the estimation (equation 6.18) shows that even in the worst-case scenario the computations required in the NMS stage only consume about one 55th of the available time for the fastest inference speed on the mid-range hardware. This margin should be sufficient even if the computation takes longer than estimated due to effects not taken into account. The other tasks which have to be performed by the CPU, such as managing the input stream from a camera and controlling the HA, are neglected in equation 6.16.
In case the NMS computation turns out to limit the inference speed, for example because a very slow PS is combined with a big PL, the problem can be addressed by increasing the confidence threshold in the NMS stage. This reduces the number of predictions to process, drastically decreasing the computational effort on the PS. If chosen optimal, the higher confidence threshold does not cause a worse performance (precision and recall). If the threshold is set too high, relevant predictions are discarded, reducing the performance of the object detector. An optimum value for the confidence thresholds per class could be determined before implementing the system on an embedded hardware by performing tests with different values and comparing the resulting precision and recall values or by using the PR curve (section 4.3.3). Another way to minimize the problem is to use a hardware with a faster PS.

7. Conclusion

7.1 Embedded Object Detector

Even though the object detector has not been implemented on the embedded hardware, the results presented in this thesis indicate that such an implementation would be possible. It is further shown what performance is to be expected from such a system and what the limitations might be. There are still some uncertainties in the estimations of this work, but most major challenges of such an implementation are addressed.
Chapters 3 and 4 describe the specific object detector network, on which all the results in this thesis are based. Even though some adjustments and changes would be required, similar results with comparable object detector networks of a similar complexity can be expected. The foundation is laid for more complex networks which achieve better mAP scores, but at the cost of higher hardware resource consumption or lower inference speed.
Chapter 5 explains the conversion to a binary form and provides the necessary results to prove the binary approximated network to behave very similar to the original. When one filter in floating point precision is approximated by only four binary filters, the mAP decrease from 0.308 to 0.303 (with retraining) is very small. With six binary filters the mAP is almost exactly the same as the original (0.308). The fact that this form of binary representation works very well with image classifiers[11] as well as object detectors supports the assumption that it can be used for other tasks as well.
In chapter 6 the possible inference speed of the object detector on a mid-range as well as a low-end hardware is estimated. A maximum speed of 99.23 inferences per second for mid-range hardware shows that this approach allows to build an object detector fast enough for most real-time applications. The maximum speed of 6.54 inferences per second on low-end hardware reflects the big difference in available hardware resources. Using cheaper hardware reduces the speed considerably, but it is still fast enough for many applications. Additionally, it shows that even the low-end implementation will outperform conventional embedded processors by far. The SSD-7 is estimated to achieve 0.435 inferences per second when executed purely on an 1 GHz CPU such as the ARM cortex A9 (appendix, section 12.2). Sections 6.4.2 and 6.4.3 investigate the challenges of memory access and storage. The ping-pong implementation of the FBUF, which allows simultaneous access from the SAs as well as the data mover, mitigates this problem. The memory operations can possibly be done completely simultaneously to the processing of the convolution operations in the SAs, if the available time is sufficient. The estimation provided in section 6.4.4 indicates that the operations to be performed by the CPU do not limit the inference speed on the target mid-range hardware.
The performance even on the low-end target hardware is sufficient to build a demonstrator aiming to present the capabilities of the Low-Cost CNN Accelerator. It could be implemented on an embedded CPU system as well to highlight the difference in processing speed.

7.2 Choosing the Parameters

This section describes how to use the findings of this thesis in two possible ways. In scenario 1, a user has a specific hardware and wants to know how fast he can run an SDD-7 object detector on it. In scenario 2, a user has a task which requires a minimum inference speed and wants to know which hardware to use. The instructions do not take into account the additional consumption of BRAM for the implementation of the FBUF.

7.2.1 Scenario 1: Target Hardware

In this example, someone wants to run an object detector on a hardware she already possesses. There are probably other designs implemented on the same hardware, but she knows exactly how much hardware resources are available. As a fictional example, there are 100'000 LUTs, 120'000 FFs, 140 DSPs and 120 BRAMs available.
In figure 6.3, respectively figure 7.1, the maximum inference for every resource can be extracted. The color of the line indicates the value of D_{arch} and the number of the datapoint in the graph is the number of systolic arrays N_{SA}. To find the maximum possible inference speed, the best values for D_{arch} and N_{SA} which can be implemented must be determined. For this, the most limiting resource needs to be identified. Then the parameters D_{arch} and N_{SA} with the highest inference speed for this resource are selected.

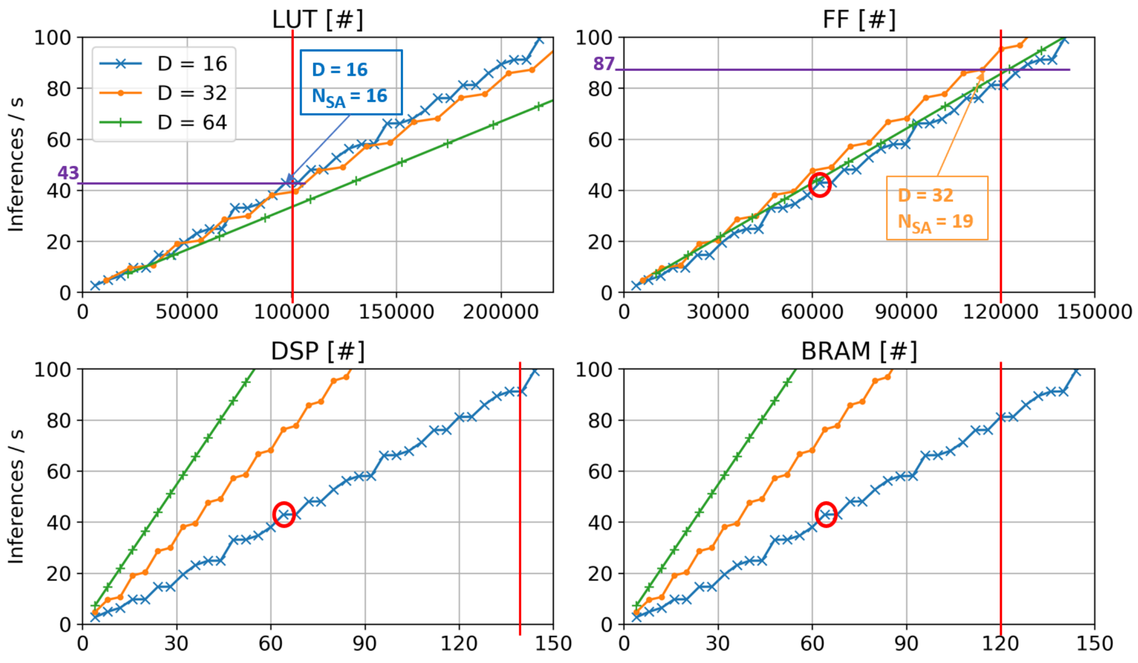


Figure 7.1: Figure 6.3 with markings for scenario 1

Figure 7.1 shows how to apply hardware constraints to determine the maximum possible inference speed and the corresponding values for D_{arch} and N_{SA}.

1. Mark the maximum available hardware resources in every graph (red lines).
2. In each graph, determine the point left of the red line with the highest inference speed (coloured arrows, purple line). In the DSP and BRAM graphs this would be a point on the green line ($D_{arch} = 64$) outside the boundaries of this figure. Because this configuration would consume more LUTs and FFs than provided, it can be neglected.
3. Identify the corresponding D_{arch} (color of the graph) and N_{SA} (number of the datapoint started from the leftmost point).
4. Identify the most limiting resource. This is the one with a maximum datapoint (D_{arch},N_{SA}) which is to the left of the red line in all the other graphs too (red circles). In this case it is LUT.
5. Read the inference speed (purple number), D_{arch} and N_{SA} values for this datapoint.

The best possible configuration for scenario 1 is $D_{arch} = 16$ and $N_{SA} = 16$ which achieves an inference speed of approximately 43 inferences per second. The exact numbers can be found in the appendix, section 12.2.

7.2.2 Scenario 2: Target Inference Speed

If there is a predefined task to be performed for an object detector on an embedded system, it will need a minimum inference speed to work properly. As a fictional example, the minimum inference speed is chosen to be 40 inferences per second.

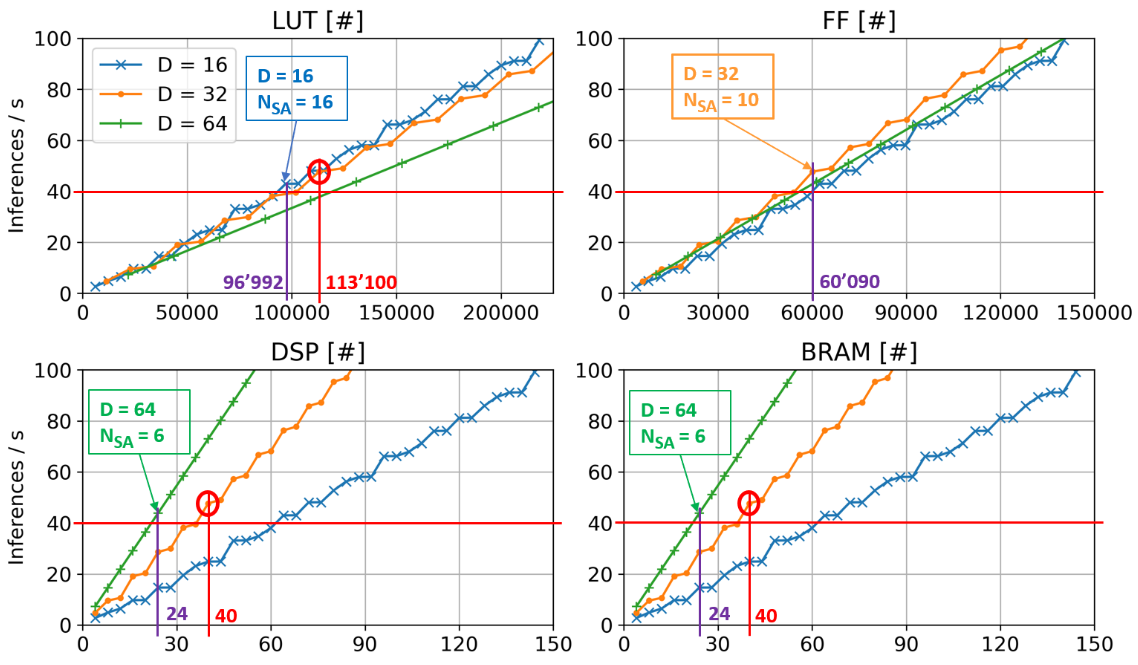


Figure 7.2: Figure 6.3 with markings for scenario 2

Figure 7.1 shows how to apply inference speed constraints to determine which hardware needs to be chosen to achieve it.

1. Mark the minimal inference speed in every graph (horizontal red lines)
2. Determine the leftmost point above the red line (coloured arrows, purple line).
3. Read the hardware requirement for this resource (purple number).
4. Choose the one resource which is most limited in the possible hardware choices (the one for which the corresponding purple number comes closest to the available resources of the same type). For scenario 2 we choose FFs to be the most limiting resource.
5. Identify the corresponding D_{arch} (color of the graph) and N_{SA} (number of the datapoint started from the leftmost point) for the chosen resource. In this scenario the values are $D_{arch} = 32$ and $N_{SA} = 10$.
6. Find the datapoints for the same configuration of D_{arch} and N_{SA} in the other graphs (red circles).
7. Read the hardware requirements for this configuration for every resource (red numbers). The resulting hardware requirements are: 113'100 LUTs, 60'090 FFs, 40 DSPs and 40 BRAMs.

The resource requirements found this way indicate the amount of resources an FPGA must provide in order to implement the architecture defined by the respective D_{arch} and N_{SA} values. If you choose a sufficiently big FPGA, the HA will achieve the required inference speed. The exact numbers can be found in the appendix, section 12.2.

7.3 Major Challenges

7.3.1 mAP Computation

The project[9] which provided the keras implementation of the SSD-7 network also contains code to determine the mAP of the trained network. However, this code did not work with the Udacity/Roboflow dataset. The error was hard to track down, so it seemed easier to implement it from scratch.
The complexity of this metric made it more challenging to implement than initially anticipated, but it also helped greatly in attaining knowledge of how it works and how object detectors work in general. The intermediate results in the computation could be used to analyze the performance in more detail and to provide figures such as figure 4.5.

7.3.2 Framework Incompatibility: Keras API vs. *tf.keras*

The SSD-7 from [9] is implemented in Keras (also know as Keras API) using TensorFlow as backend. Conversely, the implementation of the binary approximation and the necessary conversion function[11] uses TensorFlow with Keras as frontend. The term *tf.keras* indicates this method. Even though they seem to be the same and have almost the exact same features and capabilities, they are not compatible with each other. The reason for this is that Keras API is capable of working with other backend frameworks like Theano. With the term *tf.keras* another Keras version, which is built into TensorFlow, can be accessed.
This incompatibility became apparent when trying to load the model stored in .h5-file in the code to replace the convolutional layers with binary convolutional layers. One possible solution was to convert the whole SSD project to *tf.keras*. This turned out to be problematic because some custom objects (e.g. the anchors layers and the loss function) depended on functions which are different in the two frameworks. The applied solution was to build a similar network model in *tf.keras* without the custom objects, because they are unaffected by the binary approximation. The trained weights were loaded into the new model. Then the model and the trained weights were converted and stored as a new .h5-file.
It later turned out that the data generator in combination with the custom loss function were responsible for the incompatibility. The data generator takes images and labels from the dataset and prepares them for the network during training. This is usually done batch-wise in order to conserve memory space. When the data generator was removed from the model and executed in advance, storing all necessary data in the memory, the problem was solved. The cost of this workaround is a very high memory consumption on the training machine.
To solve this problem in a more effective way, the data generator needs to be adjusted to work in a *tf.keras* model.

7.4 Outlook

7.4.1 CPU Bottleneck Investigation

Even though this problem has been addressed in section 6.4.4, there were some uncertainties in the provided estimations. To determine the computation time which the CPU requires to control the HA and perform the necessary operations (primarily the NMS stage and the softmax activation), it should be implemented on the target CPU. Runtime measurements then provide with absolute certainty at which inference speed the CPU will become a bottleneck in the system.

7.4.2 Memory Access Bottleneck Investigation

Especially the necessary tiling of the feature maps described in section 6.4.3 indicates that the memory access speed is of central importance. An investigation into the rates at which data can be loaded from the memory into the FBUF and vice versa would provide valuable insight. Because the HA shares the AXI bus with the CPU it is important to consider which tasks are running on the CPU. If the CPU needs to access the memory at same time, the the data rate will be reduced.

7.4.3 Hardware Accelerator Adjustments

Up to this point, the HA has only been used for classification problems with one SA (e.g. $N_{SA} = 1$). To enable the HA to achieve the estimated inference speeds, the planned parallelization of multiple SAs has to be implemented and tested.
There are some architecture details specific to the SSD approach which need to be added as well. Most important are the unusual connections between the layers *conv4-7* with the later predictor layers. To achieve this, the convolutional layers have to be separated from the pooling layers and the feature tensor has to be stored in the FBUF for later use. At the current state, the pooling is applied directly in the output datastream of the convolutional layer and the FBUF only stores the data to be used in the next step.
A way to access the memory in a way which resembles the reshape and concatenation layers needs to be implemented as well. If this should be done by the HA or the PS is not yet clear.

7.4.4 High-Level Optimization

Another approach which is developed as part of the CNN accelerator framework is a high-level optimization[16]. Figure 6.4 shows that the first three layers contain the most potential to reduce the complexity of the network. With an evolutionary algorithm, the network architecture (e.g. filter sizes, number of layers etc.) can be optimized to consume a minimum amount of hardware resources but still achieve a decent performance. This could drastically reduce the hardware consumption of the network, allowing it to run on cheaper hardware or to increase N_{SA} and thus further increase the inference speed. This approach could also help reducing the complexity of bigger networks like the original SSD300 and SSD512[18] to be able to implement them with a smaller amount of hardware resources than otherwise required.

7.4.5 Different Datasets

To better assess the generalizability of the network as well as to better compare the results to other object detectors, it can be trained and tested on other datasets. Because of the similarity to the Udacity/Roboflow dataset and the widespread use, Pascal VOC would be a good choice for further tests. The fact that Pascal VOC contains 20 classes could mean that the SSD-7 is too simple to perform decently, if all the classes are included. For the much more complex COCO dataset it can be assumed that this network architecture would not be able to differentiate the 80 classes in various contexts. For this test, a more complex network like the original SSD300 or SSD512 should be considered.

7.4.6 Comparison with Competitors

From an economic perspective it is crucial to know how the Low-Cost CNN Accelerator compares to other embedded object detectors and hardware accelerators. To make the product attractive, it is important in which scenarios it outperforms its competitors. Implementing the same object detector, for example the SSD-7, on another hardware with a different approach of hardware acceleration could provide valuable insight in how to advertise the product once finished. It can also provide the necessary information to choose the best direction in which to focus the further development of the product.

8. List of Abbreviations

Adam	Adaptive Moment Optimization
AXI	Advanced eXtensible Interface
AP	Average Precision
API	Application Programming Interface
BACNN	Binary Approximated Convolutional Neural Network[11]
BA-SSD	Binary Approximated Single Shot Detector
BAT	Bachelor Thesis
BRAM	Block Random Access Memory
CNN	Convolutional Neural Network
COCO	Common Objects in Context[17]
CPU	Central Processing Unit
DDR3	Double Data Rate 3
DMA	Direct Memory Access
DQA	Data Quality Assessment
DSP	Digital Signal Processor
ELU	Exponential Linear Unit
FC	Fully Connected (Keras: ”dense” layer)
FF	Flip Flop
FN	False Negative
FP	False Positive
FPGA	Field Programmable Gate Array
FPS	Frames Per Second
FS20	Spring Semester 2020
GPU	Graphics Processing Unit
HA	Hardware Accelerator
HSLU	Lucerne University of Applied Sciences and Arts
LUT	Lookup Table
mAP	Mean Average Precision
NMS	Non-Maximum Suppression
NN	Neural Network
PL	Programmable Logic
PR Curve	Precision-Recall Curve
PS	Processing System
ReLU	Rectified Linear Unit
R-CNN	Region-CNN (CNN Object Detector Architecture)
SA	Systolic Array
SSD	Single-Shot MultiBox Detector[18]
SUC	Systolic Unit Cycles
TN	True Negative
TP	True Positive
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VOC	Visual Object Classes[8]
YOLO	You Only Look Once (CNN Object Detector Architecture)

List of Figures

List of Tables

Bibliography

[1] Jason Brownlee. A gentle introduction to the rectified linear unit (ReLU). https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/, 2019. (Accessed on 05/06/2020).

[2] Jason Brownlee. How to use ROC curves and precision-recall curves for classification in python. https://machinelearningmastery.com/roc-curves-and-precision-recall-curves-for-classification-in-python/, 2019. (Accessed on 03/19/2020).

[3] François Chollet et al. Keras. https://keras.io, 2015.

[4] Brad Dwyer. A popular self-driving car dataset is missing labels for hundreds of pedestrians. https://blog.roboflow.ai/self-driving-car-dataset-missing-pedestrians, 2020. (Accessed on 03/04/2020).

[5] Brad Dwyer. Udacity self driving car dataset. https://public.roboflow.ai/object-detection/self-driving-car, 2020. (Accessed on 03/04/2020).

[6] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems. https://www.tensorflow.org/, 2015.

[7] Tsung-Yi Lin et al. COCO - common objects in context. http://cocodataset.org/#detection-eval, 2019. (Accessed on 03/19/2020).

[8] Mark Everingham, Luc Gool, Christopher K. Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *Int. J. Comput. Vision*, 88(2):303–338, June 2010.

[9] Pierluigi Ferrari. SSD: single-shot multibox detector implementation in keras. https://github.com/pierluigiferrari/ssd_keras, 2018. (Accessed on 03/02/2020).

[10] Mario Fischer and Jürgen Wassner. BinArray: A flexible hardware architecture for CNNs with binary encoded weights. Thesis HSLU, 2019.

[11] Mario Fischer and Jürgen Wassner. BinArray: A scalable hardware architecture for binary approximated CNNs. MSE Thesis HSLU, 2020.

[12] Hao Gao. Object localization in overfeat - towards data science. https://towardsdatascience.com/object-localization-in-overfeat-5bb2f7328b62, 2017. (Accessed on 05/13/2020).

[13] Ross B. Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR*, abs/1311.2524, 2013.

[14] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, and Kevin Murphy. Speed/accuracy trade-offs for modern convolutional object detectors. *CoRR*, abs/1611.10012, 2016.

[15] Jonathan Hui. mAP (mean average precision) for object detection - jonathan hui - medium. https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173, 2018. (Accessed on 03/17/2020).

[16] Michael Kurmann. Optimization of neural networks for FPGA implementation. MSE Vertiefungsarbeit HSLU, 2020.

[17] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft COCO: Common objects in context, 2014.

[18] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: single shot multibox detector. *CoRR*, abs/1512.02325, 2015.

[19] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.

[20] Sumit Saha. A comprehensive guide to convolutional neural networks — the eli5 way. `https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53`, 2018. (Accessed on 06/02/2020).

[21] Tarang Shah. Measuring object detection models - mAP - what is mean average precision? `https://tarangshah.com/blog/2018-01-27/what-is-map-understanding-the-statistic-of-choice-for-comparing-object-detection-models`, 2018. (Accessed on 03/19/2020).

[22] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[23] Ren Jie Tan. Breaking down mean average precision (mAP) - towards data science. `https://towardsdatascience.com/breaking-down-mean-average-precision-map-ae462f623a52`, 2019. (Accessed on 03/16/2020).

[24] Udacity: Become a self-driving car engineer. `https://www.udacity.com/course/self-driving-car-engineer-nanodegree--nd013`, 2016. (Accessed on 05/13/2020).

12. Appendix

12.1 Equations

$$size(T_4) = A_{Fmap4} * N_{channels4} = 64 * 64 * 64\,B = 262.144\,kB \tag{12.1}$$

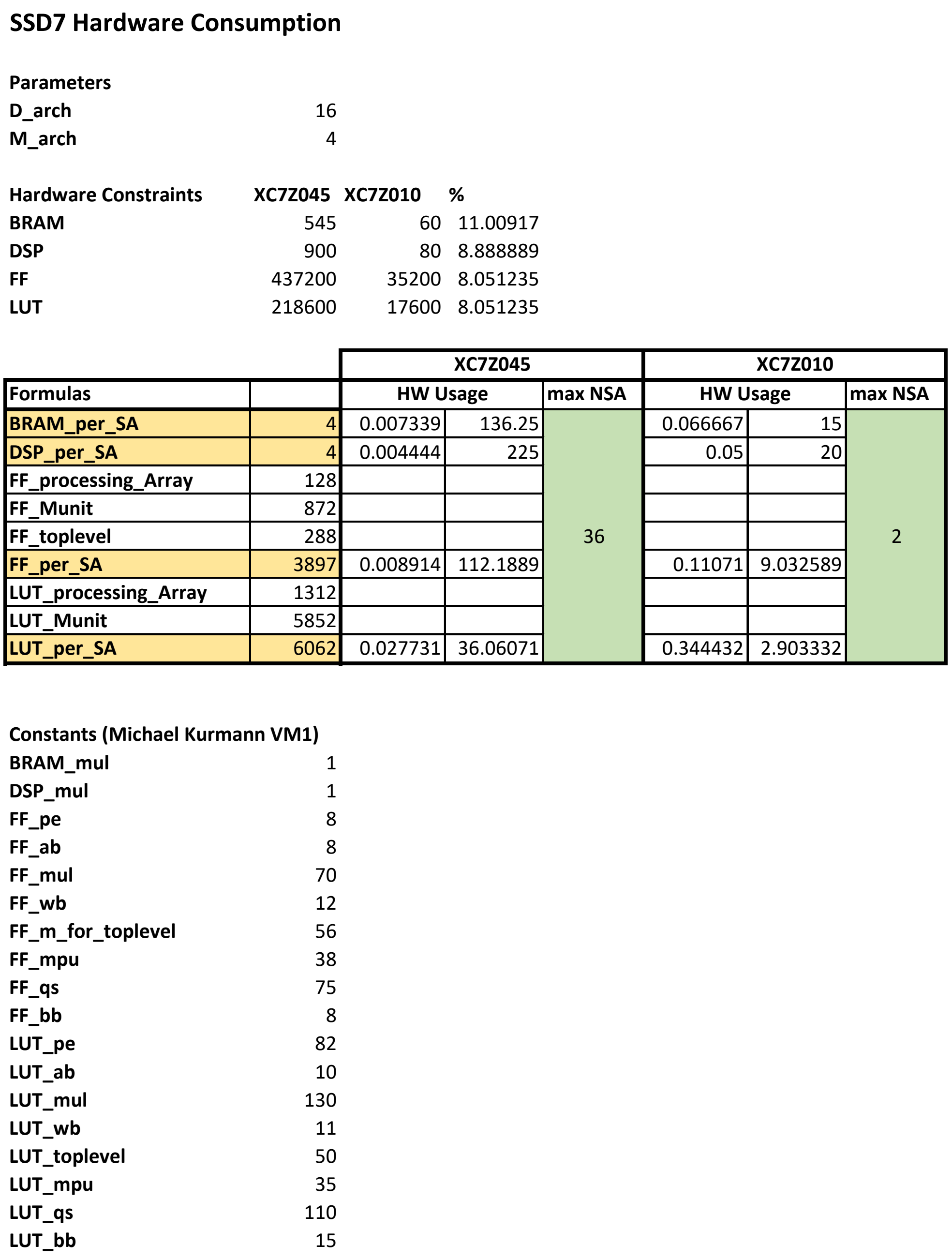
$$size(T_5) = A_{Fmap5} * N_{channels5} = 32 * 32 * 64\,B = 65.536\,kB \tag{12.2}$$

$$size(T_6) = A_{Fmap6} * N_{channels6} = 16 * 16 * 48\,B = 12.288\,kB \tag{12.3}$$

$$size(T_7) = A_{Fmap7} * N_{channels7} = 8 * 8 * 48\,B = 3.072\,kB \tag{12.4}$$

12.2 Additional Lists and Tables

12.2.1 Hardware Consumption



SSD7 Hardware Consumption

Parameters	
D_arch	16
M_arch	4

Hardware Constraints	XC7Z045	XC7Z010	%
BRAM	545	60	11.00917
DSP	900	80	8.888889
FF	437200	35200	8.051235
LUT	218600	17600	8.051235

		XC7Z045			XC7Z010		
Formulas		HW Usage		max NSA	HW Usage		max NSA
BRAM_per_SA	4	0.007339	136.25		0.066667	15	
DSP_per_SA	4	0.004444	225		0.05	20	
FF_processing_Array	128						
FF_Munit	872						
FF_toplevel	288			36			2
FF_per_SA	3897	0.008914	112.1889		0.11071	9.032589	
LUT_processing_Array	1312						
LUT_Munit	5852						
LUT_per_SA	6062	0.027731	36.06071		0.344432	2.903332	

Constants (Michael Kurmann VM1)	
BRAM_mul	1
DSP_mul	1
FF_pe	8
FF_ab	8
FF_mul	70
FF_wb	12
FF_m_for_toplevel	56
FF_mpu	38
FF_qs	75
FF_bb	8
LUT_pe	82
LUT_ab	10
LUT_mul	130
LUT_wb	11
LUT_toplevel	50
LUT_mpu	35
LUT_qs	110
LUT_bb	15

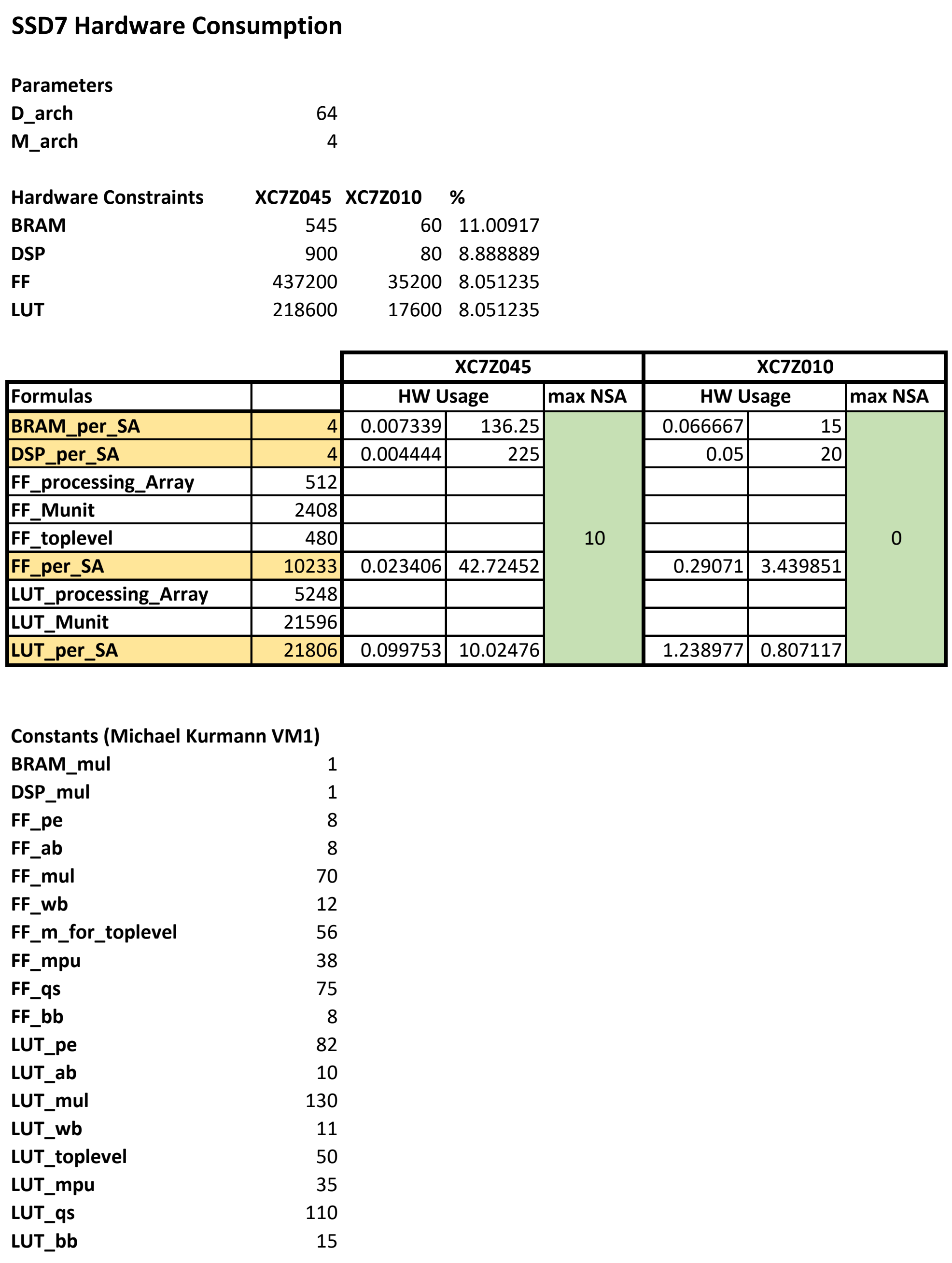
SSD7 Hardware Consumption

Parameters	
D_arch	32
M_arch	4

Hardware Constraints	**XC7Z045**	**XC7Z010**	**%**
BRAM	545	60	11.00917
DSP	900	80	8.888889
FF	437200	35200	8.051235
LUT	218600	17600	8.051235

		XC7Z045			**XC7Z010**		
Formulas		**HW Usage**		**max NSA**	**HW Usage**		**max NSA**
BRAM_per_SA	4	0.007339	136.25		0.066667	15	
DSP_per_SA	4	0.004444	225		0.05	20	
FF_processing_Array	256						
FF_Munit	1384						
FF_toplevel	352			19			1
FF_per_SA	6009	0.013744	72.75753		0.17071	5.85788	
LUT_processing_Array	2624						
LUT_Munit	11100						
LUT_per_SA	11310	0.051738	19.32803		0.642614	1.556145	

Constants (Michael Kurmann VM1)	
BRAM_mul	1
DSP_mul	1
FF_pe	8
FF_ab	8
FF_mul	70
FF_wb	12
FF_m_for_toplevel	56
FF_mpu	38
FF_qs	75
FF_bb	8
LUT_pe	82
LUT_ab	10
LUT_mul	130
LUT_wb	11
LUT_toplevel	50
LUT_mpu	35
LUT_qs	110
LUT_bb	15



SSD7 Hardware Consumption

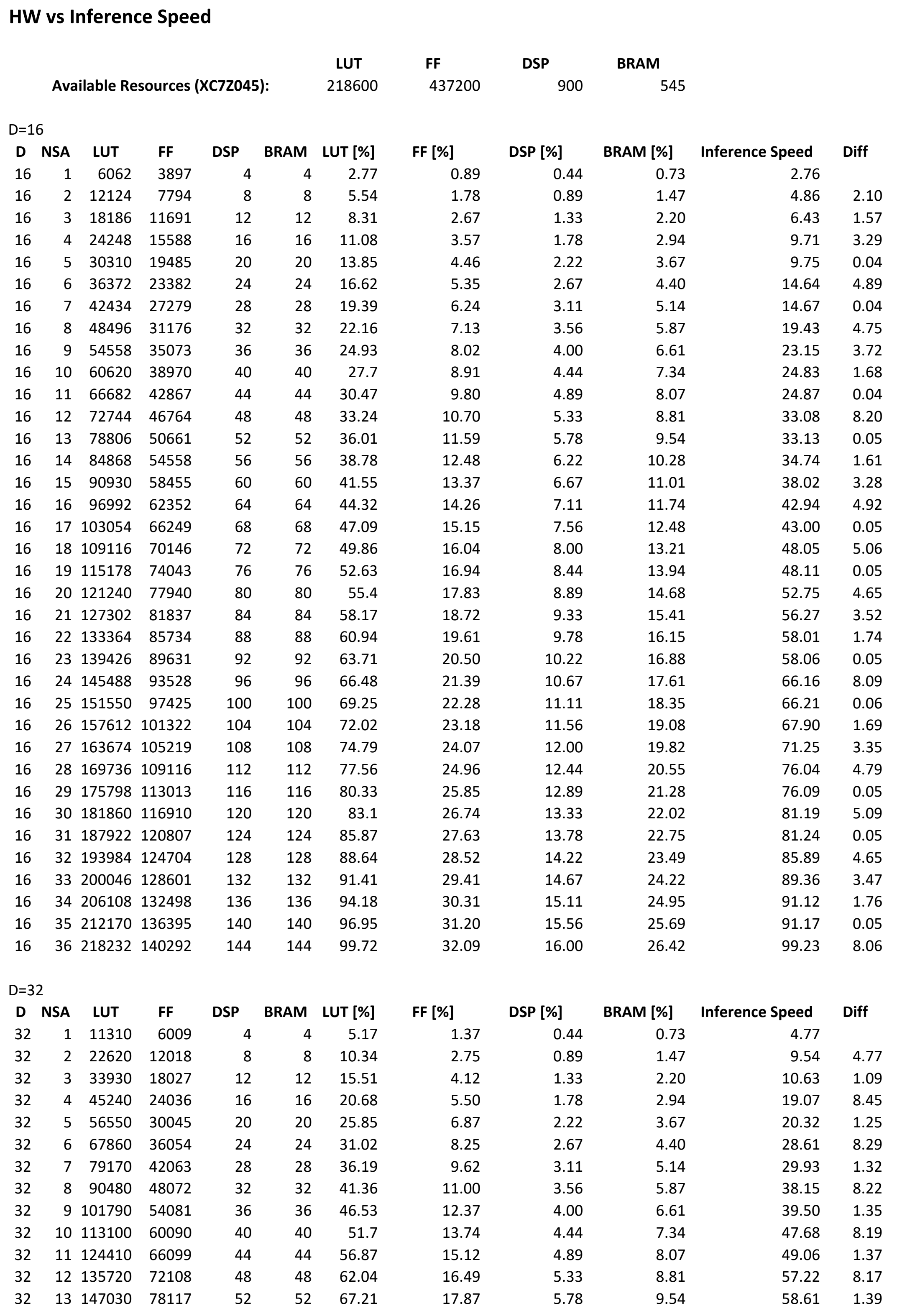
Parameters	
D_arch	64
M_arch	4

Hardware Constraints	XC7Z045	XC7Z010	%
BRAM	545	60	11.00917
DSP	900	80	8.888889
FF	437200	35200	8.051235
LUT	218600	17600	8.051235

		XC7Z045			XC7Z010		
Formulas		HW Usage		max NSA	HW Usage		max NSA
BRAM_per_SA	4	0.007339	136.25		0.066667	15	
DSP_per_SA	4	0.004444	225		0.05	20	
FF_processing_Array	512						
FF_Munit	2408						
FF_toplevel	480			10			0
FF_per_SA	10233	0.023406	42.72452		0.29071	3.439851	
LUT_processing_Array	5248						
LUT_Munit	21596						
LUT_per_SA	21806	0.099753	10.02476		1.238977	0.807117	

Constants (Michael Kurmann VM1)	
BRAM_mul	1
DSP_mul	1
FF_pe	8
FF_ab	8
FF_mul	70
FF_wb	12
FF_m_for_toplevel	56
FF_mpu	38
FF_qs	75
FF_bb	8
LUT_pe	82
LUT_ab	10
LUT_mul	130
LUT_wb	11
LUT_toplevel	50
LUT_mpu	35
LUT_qs	110
LUT_bb	15

12.2.2 Inference Speed Estimations



HW vs Inference Speed

	LUT	FF	DSP	BRAM
Available Resources (XC7Z045):	218600	437200	900	545

D=16

D	NSA	LUT	FF	DSP	BRAM	LUT [%]	FF [%]	DSP [%]	BRAM [%]	Inference Speed	Diff
16	1	6062	3897	4	4	2.77	0.89	0.44	0.73	2.76	
16	2	12124	7794	8	8	5.54	1.78	0.89	1.47	4.86	2.10
16	3	18186	11691	12	12	8.31	2.67	1.33	2.20	6.43	1.57
16	4	24248	15588	16	16	11.08	3.57	1.78	2.94	9.71	3.29
16	5	30310	19485	20	20	13.85	4.46	2.22	3.67	9.75	0.04
16	6	36372	23382	24	24	16.62	5.35	2.67	4.40	14.64	4.89
16	7	42434	27279	28	28	19.39	6.24	3.11	5.14	14.67	0.04
16	8	48496	31176	32	32	22.16	7.13	3.56	5.87	19.43	4.75
16	9	54558	35073	36	36	24.93	8.02	4.00	6.61	23.15	3.72
16	10	60620	38970	40	40	27.7	8.91	4.44	7.34	24.83	1.68
16	11	66682	42867	44	44	30.47	9.80	4.89	8.07	24.87	0.04
16	12	72744	46764	48	48	33.24	10.70	5.33	8.81	33.08	8.20
16	13	78806	50661	52	52	36.01	11.59	5.78	9.54	33.13	0.05
16	14	84868	54558	56	56	38.78	12.48	6.22	10.28	34.74	1.61
16	15	90930	58455	60	60	41.55	13.37	6.67	11.01	38.02	3.28
16	16	96992	62352	64	64	44.32	14.26	7.11	11.74	42.94	4.92
16	17	103054	66249	68	68	47.09	15.15	7.56	12.48	43.00	0.05
16	18	109116	70146	72	72	49.86	16.04	8.00	13.21	48.05	5.06
16	19	115178	74043	76	76	52.63	16.94	8.44	13.94	48.11	0.05
16	20	121240	77940	80	80	55.4	17.83	8.89	14.68	52.75	4.65
16	21	127302	81837	84	84	58.17	18.72	9.33	15.41	56.27	3.52
16	22	133364	85734	88	88	60.94	19.61	9.78	16.15	58.01	1.74
16	23	139426	89631	92	92	63.71	20.50	10.22	16.88	58.06	0.05
16	24	145488	93528	96	96	66.48	21.39	10.67	17.61	66.16	8.09
16	25	151550	97425	100	100	69.25	22.28	11.11	18.35	66.21	0.06
16	26	157612	101322	104	104	72.02	23.18	11.56	19.08	67.90	1.69
16	27	163674	105219	108	108	74.79	24.07	12.00	19.82	71.25	3.35
16	28	169736	109116	112	112	77.56	24.96	12.44	20.55	76.04	4.79
16	29	175798	113013	116	116	80.33	25.85	12.89	21.28	76.09	0.05
16	30	181860	116910	120	120	83.1	26.74	13.33	22.02	81.19	5.09
16	31	187922	120807	124	124	85.87	27.63	13.78	22.75	81.24	0.05
16	32	193984	124704	128	128	88.64	28.52	14.22	23.49	85.89	4.65
16	33	200046	128601	132	132	91.41	29.41	14.67	24.22	89.36	3.47
16	34	206108	132498	136	136	94.18	30.31	15.11	24.95	91.12	1.76
16	35	212170	136395	140	140	96.95	31.20	15.56	25.69	91.17	0.05
16	36	218232	140292	144	144	99.72	32.09	16.00	26.42	99.23	8.06

D=32

D	NSA	LUT	FF	DSP	BRAM	LUT [%]	FF [%]	DSP [%]	BRAM [%]	Inference Speed	Diff
32	1	11310	6009	4	4	5.17	1.37	0.44	0.73	4.77	
32	2	22620	12018	8	8	10.34	2.75	0.89	1.47	9.54	4.77
32	3	33930	18027	12	12	15.51	4.12	1.33	2.20	10.63	1.09
32	4	45240	24036	16	16	20.68	5.50	1.78	2.94	19.07	8.45
32	5	56550	30045	20	20	25.85	6.87	2.22	3.67	20.32	1.25
32	6	67860	36054	24	24	31.02	8.25	2.67	4.40	28.61	8.29
32	7	79170	42063	28	28	36.19	9.62	3.11	5.14	29.93	1.32
32	8	90480	48072	32	32	41.36	11.00	3.56	5.87	38.15	8.22
32	9	101790	54081	36	36	46.53	12.37	4.00	6.61	39.50	1.35
32	10	113100	60090	40	40	51.7	13.74	4.44	7.34	47.68	8.19
32	11	124410	66099	44	44	56.87	15.12	4.89	8.07	49.06	1.37
32	12	135720	72108	48	48	62.04	16.49	5.33	8.81	57.22	8.17
32	13	147030	78117	52	52	67.21	17.87	5.78	9.54	58.61	1.39

32	14	158340	84126	56	56	72.38	19.24	6.22	10.28	66.76	8.15
32	15	169650	90135	60	60	77.55	20.62	6.67	11.01	68.15	1.40
32	16	180960	96144	64	64	82.72	21.99	7.11	11.74	76.29	8.14
32	17	192270	102153	68	68	87.89	23.37	7.56	12.48	77.70	1.41
32	18	203580	108162	72	72	93.06	24.74	8.00	13.21	85.83	8.13
32	19	214890	114171	76	76	98.23	26.11	8.44	13.94	87.24	1.41
32	20	226200	120180	80	80	103.40	27.49	8.89	14.68	95.37	8.12
32	21	237510	126189	84	84	108.57	28.86	9.33	15.41	96.78	1.42
32	22	248820	132198	88	88	113.74	30.24	9.78	16.15	104.90	8.12

D=64

D	NSA	LUT	FF	DSP	BRAM	LUT [%]	FF [%]	DSP [%]	BRAM [%]	Inference Speed	Diff
64	1	21806	10233	4	4	9.98	2.34	0.44	0.73	7.29	
64	2	43612	20466	8	8	19.96	4.68	0.89	1.47	14.59	7.29
64	3	65418	30699	12	12	29.94	7.02	1.33	2.20	21.88	7.29
64	4	87224	40932	16	16	39.92	9.36	1.78	2.94	29.17	7.29
64	5	109030	51165	20	20	49.9	11.70	2.22	3.67	36.46	7.29
64	6	130836	61398	24	24	59.88	14.04	2.67	4.40	43.76	7.29
64	7	152642	71631	28	28	69.86	16.38	3.11	5.14	51.05	7.29
64	8	174448	81864	32	32	79.84	18.72	3.56	5.87	58.34	7.29
64	9	196254	92097	36	36	89.82	21.07	4.00	6.61	65.64	7.29
64	10	218060	102330	40	40	99.8	23.41	4.44	7.34	72.93	7.29
64	11	239866	112563	44	44	109.78	51.49	4.89	8.07	80.22	7.29
64	12	261672	122796	48	48	119.76	56.17	5.33	8.81	87.52	7.29
64	13	283478	133029	52	52	129.74	60.85	5.78	9.54	94.81	7.29
64	14	305284	143262	56	56	139.72	65.54	6.22	10.28	102.10	7.29

BA-SSD7

			Input Feature			Kernel						Output Feature			Performance Estimation							
Layer	Name	Type	Height	Width	Depth	Height	Width	Padding	Stride	Count	M	Height	Width	Depth	Processing Steps	N_LAS	N_SUC	N_T	N_CC	Time [s]	CPU CC	CPU Time [s]
0		Input	512	512	3																	
1	conv1	Convolution	512	512	3	5	5	2	1	32	4	512	512	32	2516582400	1	2	1	39321600	9.83E-02	629145600	6.29E-01
2	conv2	Convolution	256	256	32	3	3	1	1	48	4	256	256	48	3623878656	1	3	1	56623104	1.42E-01	905969664	9.06E-01
3	conv3	Convolution	128	128	48	3	3	1	1	64	4	128	128	64	1811939328	1	4	1	28311552	7.08E-02	452984832	4.53E-01
4	conv4	Convolution	64	64	64	3	3	1	1	64	4	64	64	64	603979776	1	4	1	9437184	2.36E-02	150994944	1.51E-01
5	conv5	Convolution	32	32	64	3	3	1	1	48	4	32	32	48	113246208	1	3	1	1769472	4.42E-03	28311552	2.83E-02
6	conv6	Convolution	16	16	48	3	3	1	1	48	4	16	16	48	21233664	1	3	1	331776	8.29E-04	5308416	5.31E-03
7	conv7	Convolution	8	8	48	3	3	1	1	48	4	8	8	48	5308416	1	3	1	82944	2.07E-04	1327104	1.33E-03
8	classes4	Convolution	64	64	64	3	3	1	1	24	4	64	64	24	226492416	1	2	1	4718592	1.18E-02	56623104	5.66E-02
9	classes5	Convolution	32	32	64	3	3	1	1	24	4	32	32	24	56623104	1	2	1	1179648	2.95E-03	14155776	1.42E-02
10	classes6	Convolution	16	16	48	3	3	1	1	24	4	16	16	24	10616832	1	2	1	221184	5.53E-04	2654208	2.65E-03
11	classes7	Convolution	8	8	48	3	3	1	1	24	4	8	8	24	2654208	1	2	1	55296	1.38E-04	663552	6.64E-04
12	boxes4	Convolution	64	64	64	3	3	1	1	16	4	64	64	16	150994944	1	1	1	2359296	5.90E-03	37748736	3.77E-02
13	boxes5	Convolution	32	32	64	3	3	1	1	16	4	32	32	16	37748736	1	1	1	589824	1.47E-03	9437184	9.44E-03
14	boxes6	Convolution	16	16	48	3	3	1	1	16	4	16	16	16	7077888	1	1	1	110592	2.76E-04	1769472	1.77E-03
15	boxes7	Convolution	8	8	48	3	3	1	1	16	4	8	8	16	1769472	1	1	1	27648	6.91E-05	442368	4.42E-04
																			Total	**0.36278016**	**Total**	**2.297094144**
																			Inference/s	**2.756**	**CPU Inference/s**	**0.435**

Parameters	
N_SA	**1**
M_arch	**4**
D_arch	**16**
T_cc CPU	**1.00E-09**
T_cc BinArray	**2.50E-09**

BA-SSD7																							
			Input Feature			Kernel						Output Feature			Performance Estimation								
Layer	Name	Type	Height	Width	Depth	Height	Width	Padding	Stride	Count	M	Height	Width	Depth	Processing Steps	N_LAS	N_SUC	N_T	N_CC	Time [s]	CPU CC	CPU Time [s]
0		Input	512	512	3																	
1	conv1	Convolution	512	512	3	5	5	2	1	32	4	512	512	32	2516582400	36	1	18	1092267	2.73E-03	629145600	6.29E-01
2	conv2	Convolution	256	256	32	3	3	1	1	48	4	256	256	48	3623878656	36	1	12	1572864	3.93E-03	905969664	9.06E-01
3	conv3	Convolution	128	128	48	3	3	1	1	64	4	128	128	64	1811939328	36	1	9	786432	1.97E-03	452984832	4.53E-01
4	conv4	Convolution	64	64	64	3	3	1	1	64	4	64	64	64	603979776	36	1	9	262144	6.55E-04	150994944	1.51E-01
5	conv5	Convolution	32	32	64	3	3	1	1	48	4	32	32	48	113246208	36	1	12	49152	1.23E-04	28311552	2.83E-02
6	conv6	Convolution	16	16	48	3	3	1	1	48	4	16	16	48	21233664	36	1	12	9216	2.30E-05	5308416	5.31E-03
7	conv7	Convolution	8	8	48	3	3	1	1	48	4	8	8	48	5308416	36	1	12	2304	5.76E-06	1327104	1.33E-03
8	classes4	Convolution	64	64	64	3	3	1	1	24	4	64	64	24	226492416	36	1	18	131072	3.28E-04	56623104	5.66E-02
9	classes5	Convolution	32	32	64	3	3	1	1	24	4	32	32	24	56623104	36	1	18	32768	8.19E-05	14155776	1.42E-02
10	classes6	Convolution	16	16	48	3	3	1	1	24	4	16	16	24	10616832	36	1	18	6144	1.54E-05	2654208	2.65E-03
11	classes7	Convolution	8	8	48	3	3	1	1	24	4	8	8	24	2654208	36	1	18	1536	3.84E-06	663552	6.64E-04
12	boxes4	Convolution	64	64	64	3	3	1	1	16	4	64	64	16	150994944	36	1	36	65536	1.64E-04	37748736	3.77E-02
13	boxes5	Convolution	32	32	64	3	3	1	1	16	4	32	32	16	37748736	36	1	36	16384	4.10E-05	9437184	9.44E-03
14	boxes6	Convolution	16	16	48	3	3	1	1	16	4	16	16	16	7077888	36	1	36	3072	7.68E-06	1769472	1.77E-03
15	boxes7	Convolution	8	8	48	3	3	1	1	16	4	8	8	16	1769472	36	1	36	768	1.92E-06	442368	4.42E-04
																			Total	**0.01007723**	**Total**	**2.297094144**
																			Inference/s	**99.234**	**CPU Inference/s**	**0.435**

Parameters	
N_SA	**36**
M_arch	**4**
D_arch	**16**
T_cc CPU	**1.00E-09**
T_cc BinArray	**2.50E-09**

BA-SSD7																						
			Input Feature			Kernel						Output Feature			Performance Estimation							
Layer	Name	Type	Height	Width	Depth	Height	Width	Padding	Stride	Count	M	Height	Width	Depth	Processing Steps	N_LAS	N_SUC	N_T	N_CC	Time [s]	CPU CC	CPU Time [s]
0		Input	512	512	3																	
1	conv1	Convolution	512	512	3	5	5	2	1	32	4	512	512	32	2516582400	1	1	1	19660800	4.92E-02	629145600	6.29E-01
2	conv2	Convolution	256	256	32	3	3	1	1	48	4	256	256	48	3623878656	1	2	1	37748736	9.44E-02	905969664	9.06E-01
3	conv3	Convolution	128	128	48	3	3	1	1	64	4	128	128	64	1811939328	1	2	1	14155776	3.54E-02	452984832	4.53E-01
4	conv4	Convolution	64	64	64	3	3	1	1	64	4	64	64	64	603979776	1	2	1	4718592	1.18E-02	150994944	1.51E-01
5	conv5	Convolution	32	32	64	3	3	1	1	48	4	32	32	48	113246208	1	2	1	1179648	2.95E-03	28311552	2.83E-02
6	conv6	Convolution	16	16	48	3	3	1	1	48	4	16	16	48	21233664	1	2	1	221184	5.53E-04	5308416	5.31E-03
7	conv7	Convolution	8	8	48	3	3	1	1	48	4	8	8	48	5308416	1	2	1	55296	1.38E-04	1327104	1.33E-03
8	classes4	Convolution	64	64	64	3	3	1	1	24	4	64	64	24	226492416	1	1	1	2359296	5.90E-03	56623104	5.66E-02
9	classes5	Convolution	32	32	64	3	3	1	1	24	4	32	32	24	56623104	1	1	1	589824	1.47E-03	14155776	1.42E-02
10	classes6	Convolution	16	16	48	3	3	1	1	24	4	16	16	24	10616832	1	1	1	110592	2.76E-04	2654208	2.65E-03
11	classes7	Convolution	8	8	48	3	3	1	1	24	4	8	8	24	2654208	1	1	1	27648	6.91E-05	663552	6.64E-04
12	boxes4	Convolution	64	64	64	3	3	1	1	16	4	64	64	16	150994944	1	1	1	2359296	5.90E-03	37748736	3.77E-02
13	boxes5	Convolution	32	32	64	3	3	1	1	16	4	32	32	16	37748736	1	1	1	589824	1.47E-03	9437184	9.44E-03
14	boxes6	Convolution	16	16	48	3	3	1	1	16	4	16	16	16	7077888	1	1	1	110592	2.76E-04	1769472	1.77E-03
15	boxes7	Convolution	8	8	48	3	3	1	1	16	4	8	8	16	1769472	1	1	1	27648	6.91E-05	442368	4.42E-04
																			Total	**0.20971776**	**Total**	**2.297094144**
																			Inference/s	**4.768**	**CPU Inference/s**	**0.435**

Parameters	
N_SA	**1**
M_arch	**4**
D_arch	**32**
T_cc CPU	**1.00E-09**
T_cc BinArray	**2.50E-09**

BA-SSD7																						
			Input Feature			Kernel						Output Feature			Performance Estimation							
Layer	Name	Type	Height	Width	Depth	Height	Width	Padding	Stride	Count	M	Height	Width	Depth	Processing Steps	N_LAS	N_SUC	N_T	N_CC	Time [s]	CPU CC	CPU Time [s]
0		Input	512	512	3																	
1	conv1	Convolution	512	512	3	5	5	2	1	32	4	512	512	32	2516582400	19	1	19	1034779	2.59E-03	629145600	6.29E-01
2	conv2	Convolution	256	256	32	3	3	1	1	48	4	256	256	48	3623878656	19	1	9	2097152	5.24E-03	905969664	9.06E-01
3	conv3	Convolution	128	128	48	3	3	1	1	64	4	128	128	64	1811939328	19	1	9	786432	1.97E-03	452984832	4.53E-01
4	conv4	Convolution	64	64	64	3	3	1	1	64	4	64	64	64	603979776	19	1	9	262144	6.55E-04	150994944	1.51E-01
5	conv5	Convolution	32	32	64	3	3	1	1	48	4	32	32	48	113246208	19	1	9	65536	1.64E-04	28311552	2.83E-02
6	conv6	Convolution	16	16	48	3	3	1	1	48	4	16	16	48	21233664	19	1	9	12288	3.07E-05	5308416	5.31E-03
7	conv7	Convolution	8	8	48	3	3	1	1	48	4	8	8	48	5308416	19	1	9	3072	7.68E-06	1327104	1.33E-03
8	classes4	Convolution	64	64	64	3	3	1	1	24	4	64	64	24	226492416	19	1	19	124174	3.10E-04	56623104	5.66E-02
9	classes5	Convolution	32	32	64	3	3	1	1	24	4	32	32	24	56623104	19	1	19	31044	7.76E-05	14155776	1.42E-02
10	classes6	Convolution	16	16	48	3	3	1	1	24	4	16	16	24	10616832	19	1	19	5821	1.46E-05	2654208	2.65E-03
11	classes7	Convolution	8	8	48	3	3	1	1	24	4	8	8	24	2654208	19	1	19	1456	3.64E-06	663552	6.64E-04
12	boxes4	Convolution	64	64	64	3	3	1	1	16	4	64	64	16	150994944	19	1	19	124174	3.10E-04	37748736	3.77E-02
13	boxes5	Convolution	32	32	64	3	3	1	1	16	4	32	32	16	37748736	19	1	19	31044	7.76E-05	9437184	9.44E-03
14	boxes6	Convolution	16	16	48	3	3	1	1	16	4	16	16	16	7077888	19	1	19	5821	1.46E-05	1769472	1.77E-03
15	boxes7	Convolution	8	8	48	3	3	1	1	16	4	8	8	16	1769472	19	1	19	1456	3.64E-06	442368	4.42E-04
																			Total	**0.01146234**	**Total**	**2.297094144**
																			Inference/s	**87.242**	**CPU Inference/s**	**0.435**

Parameters	
N_SA	**19**
M_arch	**4**
D_arch	**32**
T_cc CPU	**1.00E-09**
T_cc BinArray	**2.50E-09**

BA-SSD7																						
			Input Feature			Kernel						Output Feature			Performance Estimation							
Layer	Name	Type	Height	Width	Depth	Height	Width	Padding	Stride	Count	M	Height	Width	Depth	Processing Steps	N_LAS	N_SUC	N_T	N_CC	Time [s]	CPU CC	CPU Time [s]
0		Input	512	512	3																	
1	conv1	Convolution	512	512	3	5	5	2	1	32	4	512	512	32	2516582400	1	1	1	19660800	4.92E-02	629145600	6.29E-01
2	conv2	Convolution	256	256	32	3	3	1	1	48	4	256	256	48	3623878656	1	1	1	18874368	4.72E-02	905969664	9.06E-01
3	conv3	Convolution	128	128	48	3	3	1	1	64	4	128	128	64	1811939328	1	1	1	7077888	1.77E-02	452984832	4.53E-01
4	conv4	Convolution	64	64	64	3	3	1	1	64	4	64	64	64	603979776	1	1	1	2359296	5.90E-03	150994944	1.51E-01
5	conv5	Convolution	32	32	64	3	3	1	1	48	4	32	32	48	113246208	1	1	1	589824	1.47E-03	28311552	2.83E-02
6	conv6	Convolution	16	16	48	3	3	1	1	48	4	16	16	48	21233664	1	1	1	110592	2.76E-04	5308416	5.31E-03
7	conv7	Convolution	8	8	48	3	3	1	1	48	4	8	8	48	5308416	1	1	1	27648	6.91E-05	1327104	1.33E-03
8	classes4	Convolution	64	64	64	3	3	1	1	24	4	64	64	24	226492416	1	1	1	2359296	5.90E-03	56623104	5.66E-02
9	classes5	Convolution	32	32	64	3	3	1	1	24	4	32	32	24	56623104	1	1	1	589824	1.47E-03	14155776	1.42E-02
10	classes6	Convolution	16	16	48	3	3	1	1	24	4	16	16	24	10616832	1	1	1	110592	2.76E-04	2654208	2.65E-03
11	classes7	Convolution	8	8	48	3	3	1	1	24	4	8	8	24	2654208	1	1	1	27648	6.91E-05	663552	6.64E-04
12	boxes4	Convolution	64	64	64	3	3	1	1	16	4	64	64	16	150994944	1	1	1	2359296	5.90E-03	37748736	3.77E-02
13	boxes5	Convolution	32	32	64	3	3	1	1	16	4	32	32	16	37748736	1	1	1	589824	1.47E-03	9437184	9.44E-03
14	boxes6	Convolution	16	16	48	3	3	1	1	16	4	16	16	16	7077888	1	1	1	110592	2.76E-04	1769472	1.77E-03
15	boxes7	Convolution	8	8	48	3	3	1	1	16	4	8	8	16	1769472	1	1	1	27648	6.91E-05	442368	4.42E-04
																			Total	**0.13711872**	**Total**	**2.297094144**
																			Inference/s	**7.293**	**CPU Inference/s**	**0.435**

Parameters	
N_SA	**1**
M_arch	**4**
D_arch	**64**
T_cc CPU	**1.00E-09**
T_cc BinArray	**2.50E-09**

BA-SSD7

			Input Feature			Kernel						Output Feature			Performance Estimation							
Layer	Name	Type	Height	Width	Depth	Height	Width	Padding	Stride	Count	M	Height	Width	Depth	Processing Steps	N_LAS	N_SUC	N_T	N_CC	Time [s]	CPU CC	CPU Time [s]
0		Input	512	512	3																	
1	conv1	Convolution	512	512	3	5	5	2	1	32	4	512	512	32	2516582400	10	1	10	1966080	4.92E-03	629145600	6.29E-01
2	conv2	Convolution	256	256	32	3	3	1	1	48	4	256	256	48	3623878656	10	1	10	1887437	4.72E-03	905969664	9.06E-01
3	conv3	Convolution	128	128	48	3	3	1	1	64	4	128	128	64	1811939328	10	1	10	707789	1.77E-03	452984832	4.53E-01
4	conv4	Convolution	64	64	64	3	3	1	1	64	4	64	64	64	603979776	10	1	10	235930	5.90E-04	150994944	1.51E-01
5	conv5	Convolution	32	32	64	3	3	1	1	48	4	32	32	48	113246208	10	1	10	58983	1.47E-04	28311552	2.83E-02
6	conv6	Convolution	16	16	48	3	3	1	1	48	4	16	16	48	21233664	10	1	10	11060	2.77E-05	5308416	5.31E-03
7	conv7	Convolution	8	8	48	3	3	1	1	48	4	8	8	48	5308416	10	1	10	2765	6.91E-06	1327104	1.33E-03
8	classes4	Convolution	64	64	64	3	3	1	1	24	4	64	64	24	226492416	10	1	10	235930	5.90E-04	56623104	5.66E-02
9	classes5	Convolution	32	32	64	3	3	1	1	24	4	32	32	24	56623104	10	1	10	58983	1.47E-04	14155776	1.42E-02
10	classes6	Convolution	16	16	48	3	3	1	1	24	4	16	16	24	10616832	10	1	10	11060	2.77E-05	2654208	2.65E-03
11	classes7	Convolution	8	8	48	3	3	1	1	24	4	8	8	24	2654208	10	1	10	2765	6.91E-06	663552	6.64E-04
12	boxes4	Convolution	64	64	64	3	3	1	1	16	4	64	64	16	150994944	10	1	10	235930	5.90E-04	37748736	3.77E-02
13	boxes5	Convolution	32	32	64	3	3	1	1	16	4	32	32	16	37748736	10	1	10	58983	1.47E-04	9437184	9.44E-03
14	boxes6	Convolution	16	16	48	3	3	1	1	16	4	16	16	16	7077888	10	1	10	11060	2.77E-05	1769472	1.77E-03
15	boxes7	Convolution	8	8	48	3	3	1	1	16	4	8	8	16	1769472	10	1	10	2765	6.91E-06	442368	4.42E-04
																			Total	**0.01371189**	**Total**	**2.297094144**
																			Inference/s	**72.929**	**CPU Inference/s**	**0.435**

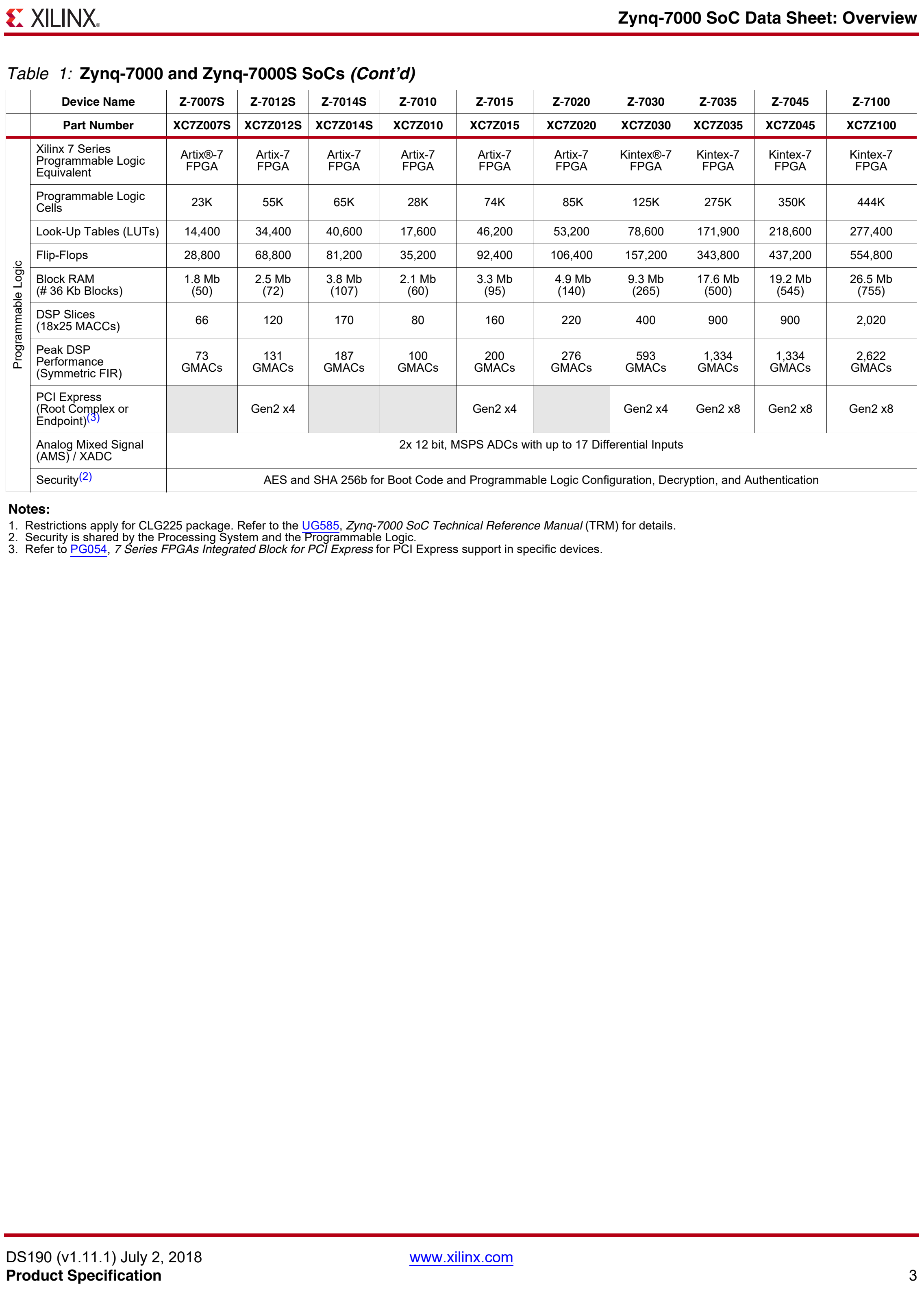
Parameters	
N_SA	**10**
M_arch	**4**
D_arch	**64**
T_cc CPU	**1.00E-09**
T_cc BinArray	**2.50E-09**

4 Tiles 4 Tiles

BA-SSD7 with Tiling																						
			Input Feature			Kernel						Output Feature			Performance Estimation							
Layer	Name	Type	Height	Width	Depth	Height	Width	Padding	Stride	Count	M	Height	Width	Depth	Processing Steps	N_LAS	N_SUC	N_T	N_CC	Time [s]	CPU CC	CPU Time [s]
0		Input	512	512	3																	
1	conv11	Convolution	256	256	3	5	5	2	1	32	4	256	256	32	629145600	1	2	1	9830400	2.46E-02	157286400	1.57E-01
2	conv12	Convolution	256	256	3	5	5	2	1	32	4	256	256	32	629145600	1	2	1	9830400	2.46E-02	157286400	1.57E-01
3	conv13	Convolution	256	256	3	5	5	2	1	32	4	256	256	32	629145600	1	2	1	9830400	2.46E-02	157286400	1.57E-01
4	conv14	Convolution	256	256	3	5	5	2	1	32	4	256	256	32	629145600	1	2	1	9830400	2.46E-02	157286400	1.57E-01
2	conv21	Convolution	128	128	32	3	3	1	1	48	4	128	128	48	905969664	1	3	1	14155776	3.54E-02	226492416	2.26E-01
3	conv22	Convolution	128	128	32	3	3	1	1	48	4	128	128	48	905969664	1	3	1	14155776	3.54E-02	226492416	2.26E-01
4	conv23	Convolution	128	128	32	3	3	1	1	48	4	128	128	48	905969664	1	3	1	14155776	3.54E-02	226492416	2.26E-01
5	conv24	Convolution	128	128	32	3	3	1	1	48	4	128	128	48	905969664	1	3	1	14155776	3.54E-02	226492416	2.26E-01
3	conv3	Convolution	128	128	48	3	3	1	1	64	4	128	128	64	1811939328	1	4	1	28311552	7.08E-02	452984832	4.53E-01
4	conv4	Convolution	64	64	64	3	3	1	1	64	4	64	64	64	603979776	1	4	1	9437184	2.36E-02	150994944	1.51E-01
5	conv5	Convolution	32	32	64	3	3	1	1	48	4	32	32	48	113246208	1	3	1	1769472	4.42E-03	28311552	2.83E-02
6	conv6	Convolution	16	16	48	3	3	1	1	48	4	16	16	48	21233664	1	3	1	331776	8.29E-04	5308416	5.31E-03
7	conv7	Convolution	8	8	48	3	3	1	1	48	4	8	8	48	5308416	1	3	1	82944	2.07E-04	1327104	1.33E-03
8	classes4	Convolution	64	64	64	3	3	1	1	24	4	64	64	24	226492416	1	2	1	4718592	1.18E-02	56623104	5.66E-02
9	classes5	Convolution	32	32	64	3	3	1	1	24	4	32	32	24	56623104	1	2	1	1179648	2.95E-03	14155776	1.42E-02
10	classes6	Convolution	16	16	48	3	3	1	1	24	4	16	16	24	10616832	1	2	1	221184	5.53E-04	2654208	2.65E-03
11	classes7	Convolution	8	8	48	3	3	1	1	24	4	8	8	24	2654208	1	2	1	55296	1.38E-04	663552	6.64E-04
12	boxes4	Convolution	64	64	64	3	3	1	1	16	4	64	64	16	150994944	1	1	1	2359296	5.90E-03	37748736	3.77E-02
13	boxes5	Convolution	32	32	64	3	3	1	1	16	4	32	32	16	37748736	1	1	1	589824	1.47E-03	9437184	9.44E-03
14	boxes6	Convolution	16	16	48	3	3	1	1	16	4	16	16	16	7077888	1	1	1	110592	2.76E-04	1769472	1.77E-03
15	boxes7	Convolution	8	8	48	3	3	1	1	16	4	8	8	16	1769472	1	1	1	27648	6.91E-05	442368	4.42E-04
																			Total	**0.3627802**	**Total**	**2.29709414**
																			Inference/s	**2.756**	**CPU Inference/s**	**0.435**

Parameters	
N_SA	**1**
M_arch	**4**
D_arch	**16**
T_cc CPU	**1.00E-09**
T_cc BinArray	**2.50E-09**

12.2.3 Datasheets



XILINX. **Zynq-7000 SoC Data Sheet: Overview**

Table 1: **Zynq-7000 and Zynq-7000S SoCs *(Cont'd)***

	Device Name	**Z-7007S**	**Z-7012S**	**Z-7014S**	**Z-7010**	**Z-7015**	**Z-7020**	**Z-7030**	**Z-7035**	**Z-7045**	**Z-7100**
	Part Number	**XC7Z007S**	**XC7Z012S**	**XC7Z014S**	**XC7Z010**	**XC7Z015**	**XC7Z020**	**XC7Z030**	**XC7Z035**	**XC7Z045**	**XC7Z100**
Programmable Logic	Xilinx 7 Series Programmable Logic Equivalent	Artix®-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Kintex®-7 FPGA	Kintex-7 FPGA	Kintex-7 FPGA	Kintex-7 FPGA
	Programmable Logic Cells	23K	55K	65K	28K	74K	85K	125K	275K	350K	444K
	Look-Up Tables (LUTs)	14,400	34,400	40,600	17,600	46,200	53,200	78,600	171,900	218,600	277,400
	Flip-Flops	28,800	68,800	81,200	35,200	92,400	106,400	157,200	343,800	437,200	554,800
	Block RAM (# 36 Kb Blocks)	1.8 Mb (50)	2.5 Mb (72)	3.8 Mb (107)	2.1 Mb (60)	3.3 Mb (95)	4.9 Mb (140)	9.3 Mb (265)	17.6 Mb (500)	19.2 Mb (545)	26.5 Mb (755)
	DSP Slices (18x25 MACCs)	66	120	170	80	160	220	400	900	900	2,020
	Peak DSP Performance (Symmetric FIR)	73 GMACs	131 GMACs	187 GMACs	100 GMACs	200 GMACs	276 GMACs	593 GMACs	1,334 GMACs	1,334 GMACs	2,622 GMACs
	PCI Express (Root Complex or Endpoint)[3]		Gen2 x4			Gen2 x4		Gen2 x4	Gen2 x8	Gen2 x8	Gen2 x8
	Analog Mixed Signal (AMS) / XADC	2x 12 bit, MSPS ADCs with up to 17 Differential Inputs									
	Security[2]	AES and SHA 256b for Boot Code and Programmable Logic Configuration, Decryption, and Authentication									

Notes:
1. Restrictions apply for CLG225 package. Refer to the UG585, *Zynq-7000 SoC Technical Reference Manual* (TRM) for details.
2. Security is shared by the Processing System and the Programmable Logic.
3. Refer to PG054, *7 Series FPGAs Integrated Block for PCI Express* for PCI Express support in specific devices.

DS190 (v1.11.1) July 2, 2018 www.xilinx.com
Product Specification 3

12.2.4 Project Definition

Lucerne University of
Applied Sciences and Arts

**HOCHSCHULE
LUZERN**

Technik & Architektur

Horw, 17. Februar 2020
Seite 1/3

Bachelor Thesis im Studiengang Elektrotechnik und Informationstechnologie

Aufgabe für Herrn Cyrill Durrer

Embedded Object Detection with Convolutional Neural Networks

Fachliche Schwerpunkte

Hardware bauen digital
Modellieren / Simulieren

Einleitung

Am CC ISN wird ein Low-Cost CNN Framework für embedded AI Echtzeit-Anwendungen entwickelt. Dieses Framework umfasst folgende Komponenten: 1. CNN-Optimierung auf Algorithmus-Ebene, 2. CNN-Optimierung auf Arithmetik-Ebene 3. FPGA-Beschleunigung der CNN-Inference, 4. FPGA-SoC Plattform.

Aufgabenstellung

Basierend auf diesem Framework soll ein Demonstrator entwickelt werden, welcher als Attraktor an Messen und Akquisitionsveranstaltungen eingesetzt werden kann. Demonstriert werden soll ein Single-Shot Detector (SSD) [1], welcher das gleiche CNN sowohl für die Detektion verschiedener Objekte in einem Bild als auch deren Klassifizierung einsetzt.

In der vorliegenden Arbeit sind dafür folgende Aufgaben zu bearbeiten:

- Erstellen und Trainieren eines SSD-Netzwerkes mit mindestens einem standardisierten Datensatz, wie z.B. [2] oder [3] in der TensorFlow-Umgebung [4]. Dabei kann von einem gegebenen Beispiel wie in [5] ausgegangen werden.
- Optimierung des SSD-Netzwerkes mittels des evolutionären Suchalgorithmus [6] für eine spätere Echtzeit-Implementierung auf einer FPGA-SoC Plattform mit dem HW-Beschleuniger [7] (die eigentliche Echtzeitimplementierung ist nicht Teil dieser Aufgabe).
- Bewertung des Detektions-Genauigkeitsverlustes durch Anwendung der binären Gewichtskodierung [8].
- Bewertung erreichbarer Detektions-Genauigkeiten und Frameraten in Abhängigkeit der zur Verfügung stehenden FPGA-Ressourcen.

FH Zentralschweiz

Hochschule Luzern
Technik & Architektur

Horw, 17.2.2020
Seite 2/3
Diplomarbeit im Fachbereich
Elektrotechnik und Informationstechnologie

Termine

Start der Arbeit:	Montag 17.2.2020
Zwischenpräsentation:	Zu vereinbaren im Zeitraum 6.4. – 1.5.2020
Abgabe Schlussbericht:	Freitag 5. Juni, vor 15:00 im Sekretariat
Abgabe Digitale Doku:	Gemäss separater Anweisung der Studiengangleitung
Abschlusspräsentation:	Zu vereinbaren im Zeitraum 8.6. – 27.6.2020
Diplomausstellung:	Freitag 3. Juli 2020 (Teilnahme obligatorisch!)

Dokumentation

Der gebundene Schlussbericht enthält keine Selbständigkeitserklärung und ist in 3-facher Ausführung zu erstellen. Er enthält zudem zwingend

- einen englischen Abstract mit maximal 2000 Zeichen.
- Ein Titelblatt, gleich hinter dem Deckblatt, gemäss Weisungen der Studiengangleitung
- Eine SD-Hülle, innen, auf der Rückseite des Berichtes für den Betreuer

Alle Exemplare des Schlussberichtes müssen komplett und termingerecht gemäss Angaben der Studiengangleitung abgeben werden. Zusätzlich muss eine SD-Speicherkarte mit dem Bericht (inkl. Anhänge), dem Poster und den Präsentationen, Messdaten, Programmen, Auswertungen, usw. unmittelbar nach der Präsentation abgeben werden.
Die gesamte Dokumentation ist zudem gemäss Anweisungen der Studiengangleitung elektronisch auf einen Server zu laden. Sämtliche abzugebende Teile der Dokumentation sind Bestandteile der Beurteilung.

Fachliteratur/Web-Links/Hilfsmittel

[1] SSD: Single Shot MultiBox Detector. W. Liu et al. https://arxiv.org/pdf/1512.02325.pdf
[2] Large Scale Visual Recognition Challenge (ILSVRC). http://www.image-net.org/challenges/LSVRC/
[3] Common Objects in Contest. http://cocodataset.org/#home
[4] TensorFlow - An end-to-end open source machine learning platform. https://www.tensorflow.org/
[5] SSD from scratch in Tensorflow. https://jany.st/post/2017-11-05-single-shot-detector-ssd-from-scratch-in-tensorflow.html
[6] M. Kurmann. Optimierung Neuronaler Netze für die FPGA Implementierung. MSE Vertiefungs-arbeit 1. Hochschule Luzern – Technik &Architektur 2020.
[7] M. Fischer. BinArray: A Scalable Hardware Architecture for Binary Approximated CNNs. Master Thesis. Hochschule Luzern – Technik &Architektur 2020.
[8] M. Fischer. Hardware-friendly Weight-Encoding of Convolutional Neural Networks. MSE Vertiefungsarbeit 1, Hochschule Luzern, 2019.

Hochschule Luzern
Technik & Architektur

Horw, 17.2.2020
Seite 3/3
Diplomarbeit im Fachbereich
Elektrotechnik und Informationstechnologie

Geheimhaltungsstufe

Öffentlich

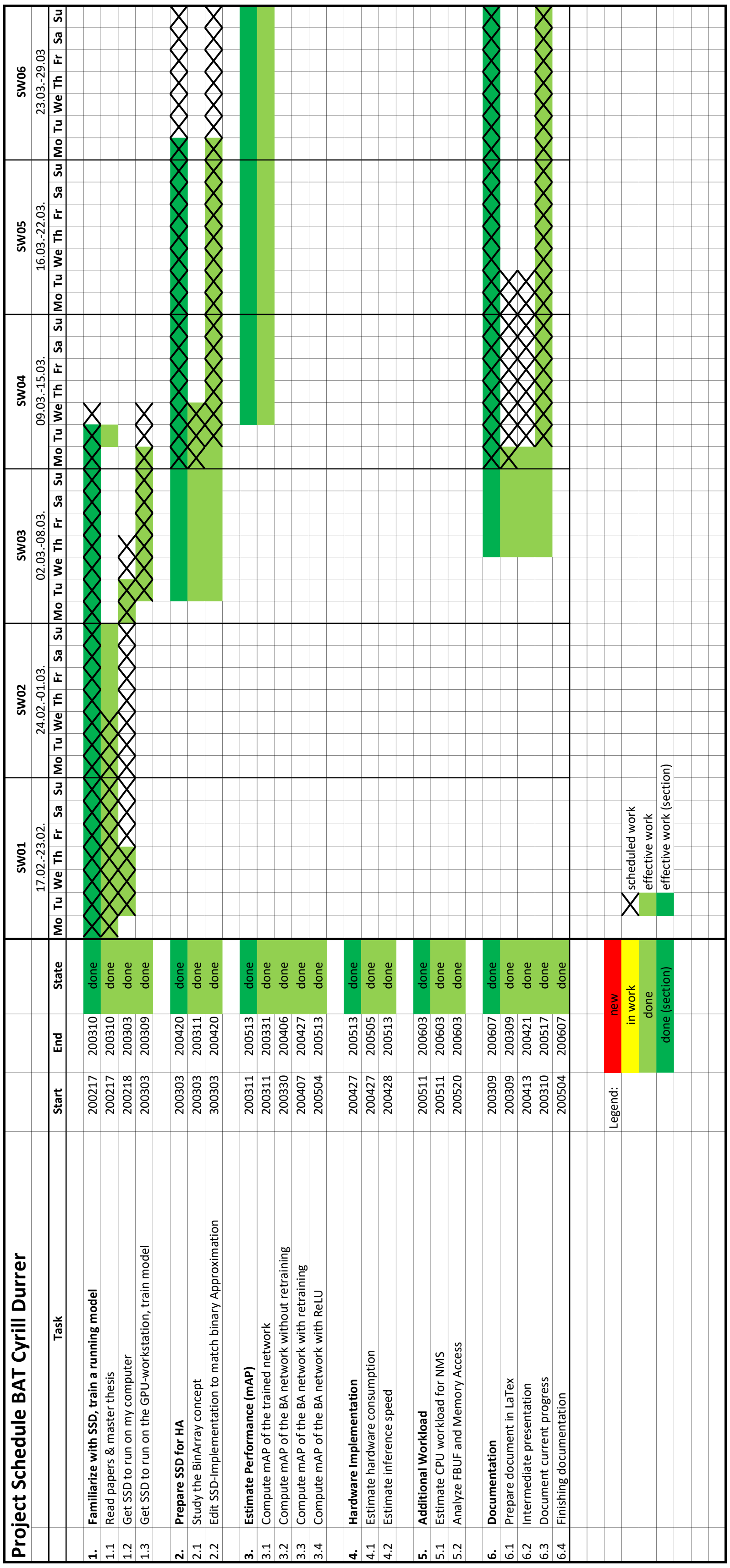
Verantwortlicher Dozent/Betreuungsteam, Industriepartner

Dozent	Prof. Jürgen Wassner	juergen.wassner@hslu.ch
Industriepartner	CC ISN Hochschule Luzern, T&A	
Experte	Thomas Schmidiger	thomas.schmidiger@maxon.ch

Hochschule Luzern
Technik & Architektur

Prof. Jürgen Wassner

12.2.5 Project Schedule

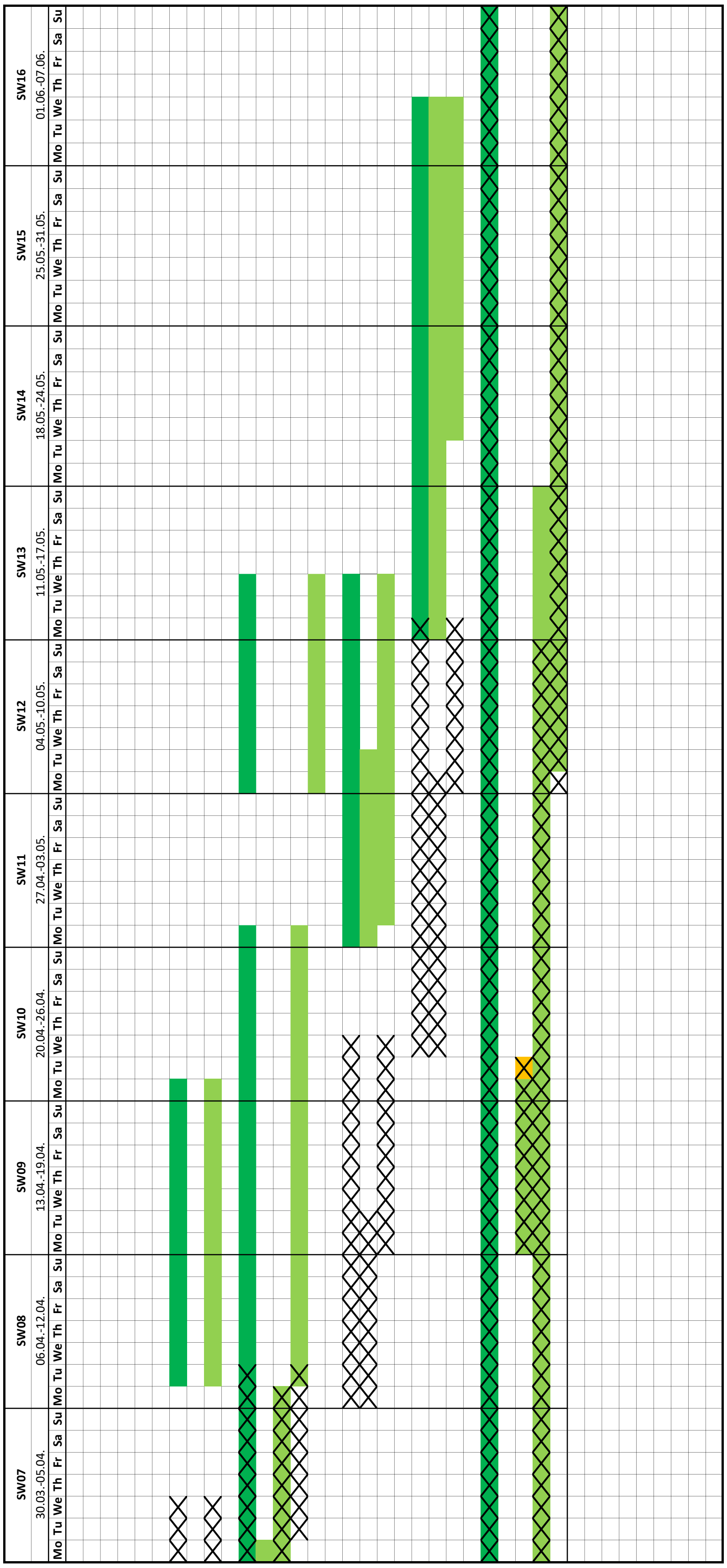


Project Schedule BAT Cyrill Durrer

	Task	Start	End	State
1.	**Familiarize with SSD, train a running model**	200217	200310	done
1.1	Read papers & master thesis	200217	200310	done
1.2	Get SSD to run on my computer	200218	200303	done
1.3	Get SSD to run on the GPU-workstation, train model	200303	200309	done
2.	**Prepare SSD for HA**	200303	200420	done
2.1	Study the BinArray concept	200303	200311	done
2.2	Edit SSD-Implementation to match binary Approximation	300303	200420	done
3.	**Estimate Performance (mAP)**	200311	200513	done
3.1	Compute mAP of the trained network	200311	200331	done
3.2	Compute mAP of the BA network without retraining	200330	200406	done
3.3	Compute mAP of the BA network with retraining	200407	200427	done
3.4	Compute mAP of the BA network with ReLU	200504	200513	done
4.	**Hardware Implementation**	200427	200513	done
4.1	Estimate hardware consumption	200427	200505	done
4.2	Estimate inference speed	200428	200513	done
5.	**Additional Workload**	200511	200603	done
5.1	Estimate CPU workload for NMS	200511	200603	done
5.2	Analyze FBUF and Memory Access	200520	200603	done
6.	**Documentation**	200309	200607	done
6.1	Prepare document in LaTex	200309	200309	done
6.2	Intermediate presentation	200413	200421	done
6.3	Document current progress	200310	200517	done
6.4	Finishing documentation	200504	200607	done

Legend:

🟥	new
🟨	in work
🟩	done
🟩	done (section)
☒	scheduled work
🟩	effective work
🟩	effective work (section)


SW07
30.03.-05.04.
SW08
06.04.-12.04.
SW09
13.04.-19.04.
SW10
20.04.-26.04.
SW11
27.04.-03.05.
SW12
04.05.-10.05.
SW13
11.05.-17.05.
SW14
18.05.-24.05.
SW15
25.05.-31.05.
SW16
01.06.-07.06.
Mo Tu We Th Fr Sa Su

12.3 Source Code

12.3.1 Dataset Preprocessing

File: Dataset_Preprocessing.ipynb

```
"""
Created on Mar 05 2020

@author: Cyrill Durrer

- Preprocessing of the Udacity/Roboflow dataset to match the format required by the
     SSD-7 network implementation.
- Merging all the different "trafficLight" classes into one
- Splitting the dataset into training and validation set
"""
import csv
import random
import numpy as np
from numpy import genfromtxt

dataset_path = r'./roboflow_small/'
filename = r'_annotations.csv'

target_training_filename=r'labels_train.csv'
target_validation_filename=r'labels_val.csv'
target_trainval_filename=r'labels_trainval.csv'

dataset_size=15000
validation_set_size=4000

#original annotations order
column_filename=0
column_width=1
column_height=2
column_class=3
column_xmin=4
column_ymin=5
column_xmax=6
column_ymax=7

#target annotations order
target_column_filename=0
target_column_xmin=1
target_column_xmax=2
target_column_ymin=3
target_column_ymax=4
target_column_class_id=5

with open(dataset_path+filename, newline='') as f:
    reader = csv.reader(f)
    csv_data = list(reader)

len(csv_data)

#extract columns from original annotations
image_filenames=[]
xmin=[]
xmax=[]
ymin=[]
ymax=[]
label_class=[]
for row in csv_data:
    #omit empty lines in csv
    if row!=[]:
        image_filenames.append(row[column_filename])
        xmin.append(row[column_xmin])
        xmax.append(row[column_xmax])
        ymin.append(row[column_ymin])
        ymax.append(row[column_ymax])
```

```python
        label_class.append(row[column_class])
len(label_class)

#change classes to class_id
class_id=[]
class_id.append('class_id')
del label_class[0]
for class_name in label_class:
    if class_name=='car':
        class_id.append('1')
    elif class_name=='truck':
        class_id.append('2')
    elif class_name=='pedestrian':
        class_id.append('3')
    elif class_name=='biker':
        class_id.append('4')
    else:#trafficLight
        class_id.append('5')
len(class_id)

target_order_list=[]
for i in range(0,len(class_id)):
    target_order_list.append([image_filenames[i],xmin[i],xmax[i],ymin[i],ymax[i],
    class_id[i]])
len(target_order_list)

#validation set split
validation_set_csv=[]
validation_set_csv.append(target_order_list[0])
i2=0
row_index=0
print("Generating validation set with size: ",validation_set_size)
for i in range(0,validation_set_size):
    rn=random.randrange(1, len(target_order_list)-1)
    current_filename=target_order_list[rn][0]
    while True:
        try:
            row_index = image_filenames.index(current_filename)
        except ValueError:
            break
        validation_set_csv.append(target_order_list[row_index])
        del image_filenames[row_index]
        del target_order_list[row_index]
print("Successfully generated validation set with ",len(validation_set_csv),"
    labels.")

#Save target csv-files
with open(dataset_path+target_training_filename, 'w', newline='') as training_csv:
    wr = csv.writer(training_csv, quoting=csv.QUOTE_ALL)
    for row in target_order_list:
        wr.writerow(row)
training_csv.close()

with open(dataset_path+target_validation_filename, 'w', newline='') as
    validation_csv:
    wr = csv.writer(validation_csv, quoting=csv.QUOTE_ALL)
    for row in validation_set_csv:
        wr.writerow(row)
validation_csv.close()

del target_order_list[0]
with open(dataset_path+target_trainval_filename, 'w', newline='') as trainval_csv:
    wr = csv.writer(trainval_csv, quoting=csv.QUOTE_ALL)
    for row in target_order_list:
        wr.writerow(row)
    for row in target_order_list:
        wr.writerow(row)
trainval_csv.close()
```

12.3.2 mAP Computation

File: BA-SSD7/mAP_computation_utils/mAP_utils

```python
"""
Created on Tue Mar 24 09:58:29 2020

@author: Cyrill Durrer

Mean Average Precision computation utils:
- IoU(): Intersection over Union calculator
- confusion_values(): calculates true positives, predicted positives and ground truth positives
- area_under_curve(): calculates area under curve as right riemann sum (for pr-curves usually smaller than the true area under the curve)
"""

import numpy as np

def IoU(ground_truth_coordinates,prediction_coordinates):
    xmin_p = prediction_coordinates[2]
    ymin_p = prediction_coordinates[3]
    xmax_p = prediction_coordinates[4]
    ymax_p = prediction_coordinates[5]

    xmin_gt = ground_truth_coordinates[1]
    ymin_gt = ground_truth_coordinates[2]
    xmax_gt = ground_truth_coordinates[3]
    ymax_gt = ground_truth_coordinates[4]

    if xmax_p<xmin_gt or ymax_p<ymin_gt or xmax_gt<xmin_p or ymax_gt<ymin_p:
        #IoU computation: no overlap, returning 0
        return 0.0

    area_p = (xmax_p-xmin_p)*(ymax_p-ymin_p)
    area_gt = (xmax_gt-xmin_gt)*(ymax_gt-ymin_gt)

    area_intersection = (min(xmax_gt,xmax_p)-max(xmin_gt,xmin_p))*(min(ymax_gt,ymax_p)-max(ymin_gt,ymin_p))

    area_union = (area_p+area_gt)-area_intersection

    IoU_value = area_intersection/area_union
    return IoU_value

def confusion_values(y_pred_decoded,gt_labels,IoU_threshold=0.45,nof_classes=5):
    true_positives=np.zeros(nof_classes)
    gt_boxes=np.zeros(nof_classes)
    pred_boxes=np.zeros(nof_classes)
    precision=np.zeros(nof_classes)
    recall=np.zeros(nof_classes)

    for i in range(len(y_pred_decoded)):
        pred_boxes[int(y_pred_decoded[i,0])-1]+=1

    for i in range (len(gt_labels)):
        gt_boxes[gt_labels[i,0]-1]+=1
        for j in range (len(y_pred_decoded)):
            if gt_labels[i,0]==y_pred_decoded[j,0]:
                if IoU(ground_truth_coordinates=gt_labels[i],prediction_coordinates=y_pred_decoded[j])>=IoU_threshold:
                    true_positives[gt_labels[i,0]-1]+=1
                    y_pred_decoded=np.delete(y_pred_decoded,j,0)
                    break;
    return true_positives, pred_boxes, gt_boxes

def area_under_curve(x, y):
    if len(x)!=len(y):
        print("Error: unable to calculate area under curve: arrays of different size. Returning 0.")
        return 0
    area_under_curve=x[len(x)-1]*(2*y[len(x)-1])/2
    for i in range(len(x)-2,0,-1):
        area_under_curve+=(x[i]-x[i+1])*(y[i]+y[i+1])/2
        last_x=x[i]
    return area_under_curve
```