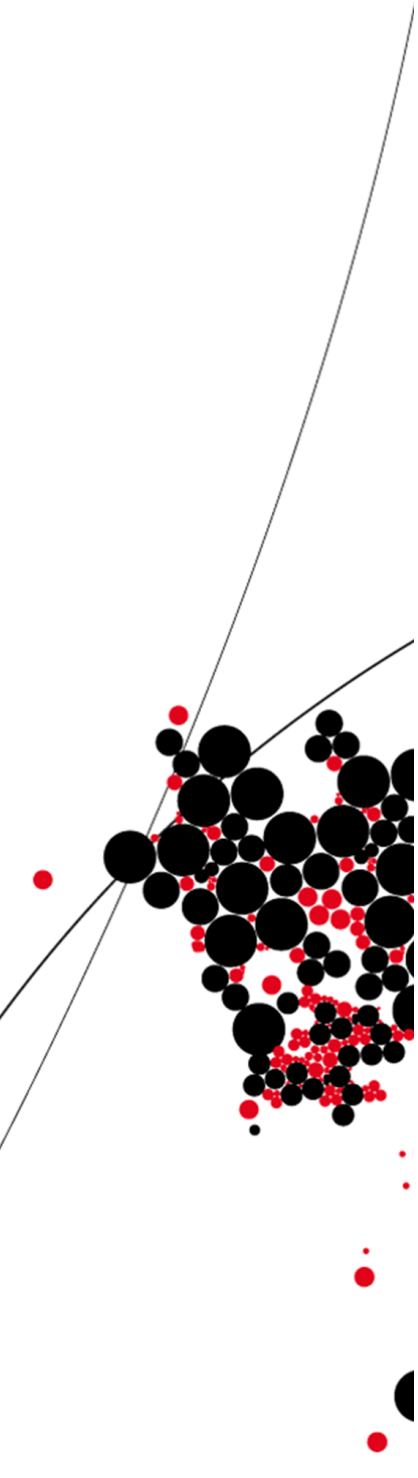
UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering,
Mathematics & Computer Science

Embedded Neural Network Design on the ZYBO FPGA for Vision Based Object Localization

Konstantinos Fatseas

M.Sc. Thesis
July 2018



Supervisors:
Prof. dr. ir. M. J. G. Bekooij
Ir. J. Scholten
O. Meteer, MSc

Computer Architectures for
Embedded Systems Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
7522 NH Enschede
The Netherlands

Abstract

During recent years we have been witnessing great advancements in the field of computer vision due to the utilization of machine learning. Those achievements are attributed to the evolution of Convolutional Neural Networks (CNNs) as nowadays they are the basic building block of every state-of-the-art object detector and classifier. There is an abundance of possible applications for CNNs and especially in cyber-physical systems, but the utilization of a CNN comes with a very high computation and memory requirements.

Therefore the objective of this work is to study the suitability of machine learning and more specifically of a CNN in a visual servoing application. The application itself consists of two robots where one has the role of prey and the other of the predator. The CNN is defined as a classifier and trained on a generated training dataset in order to avoid manual annotation. Then it is integrated into an object detection pipeline in order to extract the precise location of the prey robot. Furthermore, a pre-trained neural network is utilized so as to improve the overall performance of the object detection pipeline. Finally, an accelerator is developed in order to port the object detector into the ZYBO FPGA board and to meet the real-time constraint.

Robust performance from both the object tracking algorithm and the accelerator is reported. Although the generated training dataset is the limiting factor of this work, the methodology overall offers a lot of potential for future improvements.

Acknowledgments

I would first like to thank my supervisor Marco Bekooij. He consistently allowed this project to be my own work, but steered me in the right direction whenever I needed it. Furthermore, the proposed project was very challenging and gave me the opportunity to practice the hardware design skills I've acquired during my study in the Embedded Systems M.Sc. program. While, it also acted as a gentle introduction to the increasingly interesting field of Machine Learning.

I would also like to thank all the colleagues that kept me company throughout my study. More specifically, Viktorio, Oguz, Kiavash, Zhiyuan, Matheus, Stefano and Giovanni.

Finally, I must express my very profound gratitude to my parents Ineke and Yannis, and to my partner Nicoleta for providing me with unfailing support and continuous encouragement throughout my years of study. This accomplishment would not have been possible without them. Thank you.

Konstantinos Fatseas
Enschede, July 15th 2018

Contents

List of Figures

List of Tables

Nomenclature

ANN	Artificial Neural Network
ASIC	Application-Specific Integrated Circuit
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DDR	Double Data Rate
DMA	Direct Memory Access
DNN	Deep Neural Network
DSP	Digital Signal Processing
FPGA	Field-Programmable Gate Array
FPS	Frames Per Second
GPU	Graphic Processing Unit
IC	Integrated Circuit
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
RAM	Random-Access Memory
ReLU	Rectified Linear Unit
SDRAM	Synchronous Dynamic Random-Access Memory
SoC	System on Chip
SVM	Support Vector Machine
VDMA	Video Direct Memory Access

1 Introduction

During recent years we have been witnessing great advancements in the field of computer vision due to the utilization of machine learning. More specifically, image classification and object detection algorithms have been improved and are now delivering even super human accuracy in some tasks. The driving force has been the improvement of deep learning methods that led to the evolution of Convolutional Neural Networks (CNN). CNNs have been the core element of all the state-of-the-art image classifiers and object detectors after the year 2012 when the winning method of the ImageNet Large Scale Visual Recognition Challenge [28] (ILSVRC) was based on a artificial neural network for the first time.

After the initial success of Krizhevsky et al.[16] at the 2012 ILSVRC, CNNs have been extensively studied resulting to an abundance of new tools, methods and applications. For this reason CNNs are nowadays a strong candidate for any task that involves computer vision like image based search on the Internet or autonomous cars where image sensors can provide information about the surroundings. Other areas that CNNs have been applied in order to achieve state-of-the-art performance include object tracking, semantic segmentation, pose estimation, text detection and recognition, scene labeling, image restoration etc. In fact, CNNs provide such a robust platform that they have been utilized in numerous applications even out of the field of computer vision due to their ability to exploit correlations not only in the spatial domain but in the time and frequency domains as well. For example a CNN can be part of a system in areas like natural language processing, speech processing and speech synthesis.

The main advantage of machine learning is that it shifts the effort from the developing of application specific algorithms to the creation of a suitable training dataset for each task. On one hand, the collection of the training material and its annotation is a relatively simple task. On the other hand, the vast amount of examples that are usually needed require a lot of man-hours to be manually annotated. The importance of the training dataset is such that the spring of machine learning we are witnessing can be partially attributed to the compilation of large training datasets like the ImageNet that were not available in the past. The other two main reasons are the overall improvement of the training algorithms and the availability of Graphic Processing Units (GPUs) for general purpose computing.

1.1 Motivation

There is an abundance of possible applications for CNNs and especially in cyber-physical systems but the utilization of a CNN comes with a very high computation and memory requirements. For this reason a lot of attention is being paid in applying CNNs on resource limited platforms. The main approach as for today is to let remote servers handle the computation of the CNN. This is the case for many applications that exist on the iOS and Android platforms and use deep learning algorithms for natural language processing, object detection etc. Except of the need for a connection between the embedded platform and the server, this approach is not only increasing latency but also introduces latency variations due to possible traffic on the network or bad connection quality in

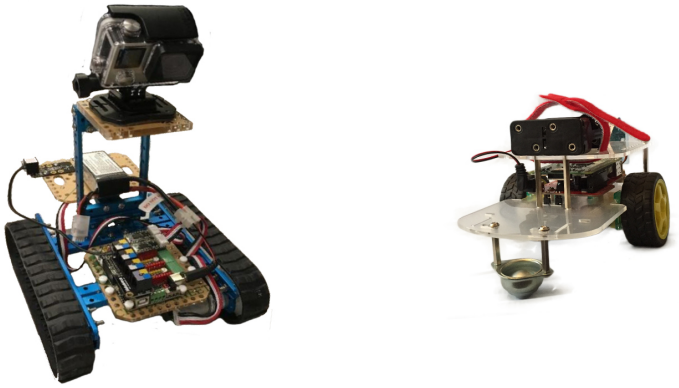


Figure 1: Predator (left) and prey (right) robots. The GoPro action camera which is mounted on top of the predator robot, is used to provide the input video stream.

case of wireless transmission. Furthermore, wireless data transmission requires excessive energy consumption, especially on a cellular network.

Applications like autonomous driving or industrial automation are subject to tight real-time constrain and low latency requirements, thus the solution of remote processing is not applicable in all cases. For real-time systems there is a need for reliable processing platforms that can provide guarantees about the execution time and also introduce low latency while being energy efficient. Meeting those requirements on platforms with limited resources is challenging but the availability of modern SoCs that include a portion of FPGA fabric lets us exploit the parallelism of CNN computation[22] and deliver enough computing power with a very good performance per watt ratio.

1.2 Problem Definition

The goal of this thesis has been the utilization of machine learning and more specifically of a CNN in a visual servoing application. The application itself consists of two robots (figure 1) were one has the role of prey and the other of the predator, thus one robot (the predator) has to track the other and follow closely. Although initially the neural network was intended to take full control of the robot by being trained from end-to-end, after the involvement of a fellow student in the project it was decided to develop a more conventional control system. The controller would combine the output of the CNN with information about the ego motion by performing sensor fusion. As a result the CNN would be used only for tracking of the prey robot by using frames coming from a single image sensor.

It is known that employing a CNN on a resource limited platform is a challenging task so from the beginning of the project the ZYBO board was chosen as the platform to facilitate the tracking and control algorithms. The reason is that the ZYBO board is build around the Xilinx Zynq 7010 or 7020 SoC which not only provides an abundance of peripherals in order to easily attach an image sensor. It can also provide enough computational power by combining a dual

core ARM processor together with FPGA fabric.

Furthermore, from the early stages of development it was known that the Raspberry Pi 3 board (which is already available on the prey-predator robots) is incapable of performing the amount of computation that state-of-the-art CNNs require in real-time. Additionally, in order to meet the real-time constrain with even a smaller CNN there was need for specialized performance-oriented software that would make use of the DSP capabilities of the ARM CPUs or of the VideoCore which is the equivalent of a GPU and is integrated into the BCM2837 SoC.

Those options were not taken into consideration due to the lack of scalability that such a specialized software solution would suffer from. This is in contrast to the FPGA based SoC where the CNN computation can be accelerated on the FPGA thus leaving room for the control and navigation algorithm to run on the ARM cores. Finally, another issue of the Raspberry Pi is the latency that is introduced from the camera driver, this is completely omitted on the ZYBO board by synthesizing dedicated hardware for the image acquisition and using direct memory access (DMA) to directly save pixel values into the DDR SDRAM memory.

To summarize, the research question can be defined as follows:

Can a CNN be trained in order to act as a single-object detector, accelerated to meet the real-time constrain and integrated into an object tracking system on the ZYBO board ?

To answer the question, we need to brake it down into smaller objectives as such a system will require algorithmic, software and hardware development. Additionally, every step and decision taken will have consequences on the following steps. The objectives that arise from the research question are the following:

1. Create a training dataset suitable for this application
2. Research and evaluate existing CNN architectures
3. Develop a CNN based object detection algorithm
4. Define a subset of functions to be used for the CNN that are suitable for acceleration from a complexity point of view
5. Develop software so as to study the underlying computation of the detection algorithm and use as reference for the hardware implementation
6. Realize a real-time object detection system on the ZYBO board

1.3 Related work

The initial approach to the problem was to investigate the end-to-end training possibilities. This is to train a convolutional neural network to process the video stream from an image sensor and to directly predict the correct steering wheel angle while maintaining a constant speed. End-to-end autonomous driving was firstly achieved by Pomerleau in 1989[25] but was based on a fully connected neural network, thus was not optimal for high resolution images. LeCun et al.[23] did some further work in 2005 by training a small robotic vehicle to avoid obstacles on rough terrain. This time a convolutional neural network was

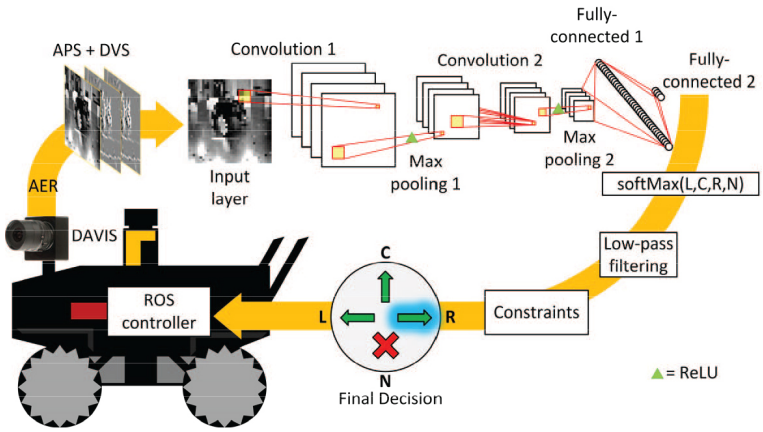


Figure 2: The closed loop system developed by Moeys et al.[21]

used in order to process color images of higher resolution coming from a stereo camera setup.

More recently, in 2016 Moeys et al.[21] in a very similar application (figure 2) to the one of this work, trained a CNN to make a robotic vehicle follow another one by relying only on visual information. Although, the work of Moeys provided a good starting point regarding the convolutional neural network architecture, there is a substantial difference regarding the computing platform that was used. All the aforementioned studies rely on external stationary platforms for the computation of the neural network. In contrast to this project were the goal is to perform all the computation on-board on a small sized SoC FPGA board and in an energy efficient manner.

This brings us to the second part of this thesis, the hardware acceleration of the CNN. Because of the potential that vision based applications have, especially on embedded devices, there already exist numerous CNN accelerators. The existing accelerators target both ASICs and FPGAs as the first can deliver sheer performance on high clock frequencies and be energy efficient but the later can be more flexible and adapt to the CNN that is to be accelerated or to future architectures. ShiDianNao[3] and Eyeriss[2] are two of the ASIC CNN accelerators found in the literature. They are complex structures though, so it was unreasonable to base the design of this thesis's accelerator on those studies, mainly due to time limitation. There also exist a lot of accelerators for FPGAs but again the goal of those studies is to deliver state-of-the-art performance, resulting to complex architectures that are synthesized for high range FPGAs.

Another area that researchers have been studying is the minimization of operations and parameters of CNNs, this is important for embedded applications as it will close the gap between state-of-the-art CNNs and limited resource platforms. This can be done by defining architectural structures that require less computation like the SqueezeNet[13] and MobileNets[10] that do not result into performance degradation. Other ways to minimize the computational burden and the size of CNNs is the use of binary weights[12], for example only 1 and -1. This technique has a lot of potential because it will be very easy to implement multiplierless accelerators but the training procedure is not supported by the

tools that are available at the moment.

This work is focusing on both the development of the CNN and its accelerator. Therefor, it is necessary to extract useful features from the literature of both the deep learning and CNN acceleration domains and combine them in order to create a simple workflow that will allow the training of small CNNs and its acceleration on the ZYBO board. The process of transferring the structure and the coefficients to the embedded platform is also a challenging task but examples of this process can be found in studies like the Angel-Eye[8].

1.4 Contributions

Due to lack of experience on neural networks within the chair of Computer Architectures for Embedded Systems, this thesis mainly contributes as a starting point for further research on the topic and emphasizes on minimizing the computational cost and latency of the inference. For this reason the use of commercial frameworks or High-Level Synthesis is avoided and instead only open source libraries are used and the register-transfer level description is written with VHDL.

The points that this thesis contributes to can be specified as follows:

- Developing a method that can produce a large synthetic training dataset with only a small number of actual images.
- Training of a relatively small convolutional neural network on the automatically generated training set and compare it with existing architectures.
- Extracting the object's location by inspecting the output of the last convolutional layer in order to avoid complex algorithms with high computational cost.
- Design and develop a re-configurable and lightweight CNN accelerator for an FPGA that meets the real-time constrain and introduces acceptable latency.
- Integrate the accelerator into a SoC and develop the appropriate driver in order to evaluate the accelerator and the neural network.

1.5 Report outline

In order to present the basic concepts that the reader needs to be aware, the 2nd Chapter provides general information on neural networks and presents some of the accelerators that have already been developed. Additionally, this chapter discusses the convolution operation as defined in the field of computer vision and the other functions that can be used in feed-forward neural nets. Moreover the state of the art CNN based object detectors are presented and their advantages and drawbacks are discussed.

The 3rd Chapter is focused on object detection. First, the training set generator is presented together with the training procedure of the CNN. Then we evaluate the training dataset. This chapter also presents the simplest way to extract an object's location by using the output of the last convolutional layer. Additionally, it shows how this method can also be used together with a state of the art classifier to create a high accuracy and general purpose object detector.

Chapter 4 documents the architecture that was developed and the reasoning behind the choices that were made throughout the process. Furthermore, the communication between the software and hardware is explained together. Finally, the overall performance of the accelerator and of the convolutional neural network itself are discussed..

Finally, Chapter 5 concludes this thesis and discusses possible future work.

2 Background

Machine learning is a sub field of Artificial Intelligence and refers to computer systems that have the ability to learn how to solve a problem rather than being explicitly programmed to do so. Artificial neural networks are only one of the approaches of machine learning, some of the other approaches are the Support Vector Machines, and data clustering.

Artificial neural networks are computing systems loosely inspired by nature. A neuron in nature typically consists of the dendrites (Input connections), the main body and the axon (output connection). Its output is related to its inputs so a neuron can be thought as a system that receives, processes and transmits information. In a neural network like a brain, different kind of neurons interconnected with each other create a complex system capable of perceiving the world and exhibit intelligence. This is a simplification though, because the human brain for example is far more complex. It is still unknown what is the exact functionality of a neuron and how exactly they contribute in this high form of intelligence that a human can display.

2.1 Artificial Neural Networks

A simple neuron in mathematics will have a set of n inputs, $(x_1,x_2,...,x_n)$ and a single output y. The neuron then has to learn a set on n weights, each of them will be multiplied with the corresponding input and the sum of the results will be the output. Thus, the functionality of a simple artificial neuron can be described as a function $f(x,w) = x_1w_1 + x_2w_2 + ... + x_nw_n$. Being a linear function though, such a simple neuron cannot form networks that are able to be trained to solve difficult problems. The solution to this is to add an extra activation function that will have $f(x,w)$ as an input and will calculate output y in a non-linear fashion, some of the most commonly used non-linear functions will be introduced in section 2.2. The structure of such an artificial neuron can be seen in figure 3. First the sum of the weighted inputs is calculated and then the so called transfer function or activation function will produce the corresponding output of the neuron.

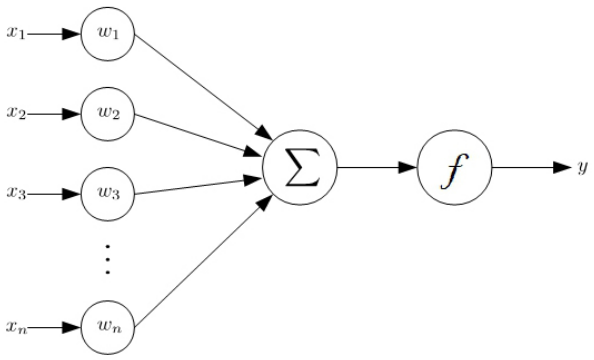


Figure 3: An artificial neuron simulates the basic functionality of biological neurons. It receives information from other neurons, it process the information and transmits the result to other neurons.

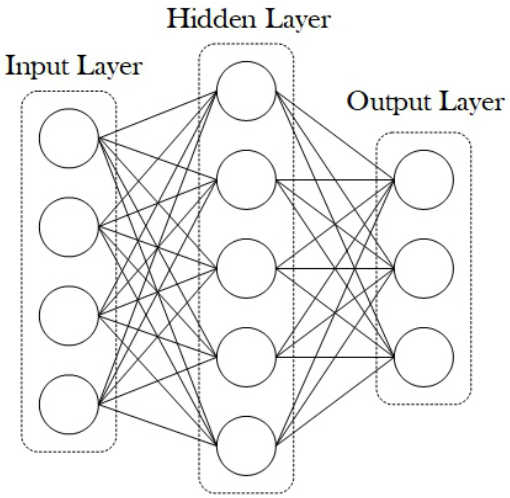


Figure 4: Example of an artificial neural network (ANN) and more specifically of a deep neural network as between the input and output layer is also a hidden one. An ANN is an interconnected group of nodes like the one depicted at figure 1. Information flows towards one direction (*input* $\rightarrow$ *output*) in a feed-forward network like this one.

2.1.1 Neural Network Topology

By using the artificial neurons as basic structure elements we can now arrange them so that multiple neurons are interconnected and form what is called a deep feedforward network, such a topology is depicted in figure 4. The neurons are grouped in layers where a neuron's inputs is connected to the outputs of all the neurons that the previous layer has, such an arrangement is called a densely or fully connected network.

A deep feedforward network may have 3 or more layers, the first layer is directly connected to the input and the last layer is the output of the network, the layers in between are called hidden layers. The basic property of this type of network is that the information flows from the input towards the output without any feedback loop. Furthermore, the neurons do not have any connection with other neurons of the same layer. Those properties are important when it comes to hardware design because they allow a great degree of parallelism as shown in chapter 4.

Neural network architectures can vary significantly and in fact they can also include feedback loops to the network in order to give the ability to memorize information from previous inputs. A previous input may be from any dimension of the dataset, for example in a video, the word 'previous' refers to the time-stamp whereas in natural language it may refer to the word or sentence before the one that is being processed. Although recurrent networks can achieve remarkable results in areas like natural language processing or video processing, they are out of the scope of this project due to the complexity that they introduce in the data process path.

2.1.2 Training of Neural Networks

The success of neural networks derives from the ability to gradually learn how to solve a given task by being trained. Training is the procedure of setting the correct values to every neuron's weight set so that the final output of the network approximates the desired response as close as possible.

Throughout this project the training methods were not investigated and by using a machine learning library the underlying training algorithms are hidden as the neural network development is done in a higher level of abstraction. Although there is no demand for a machine learning practitioner to be fully aware of the training procedure, some basic concepts need to be explain due to the fact that we still lack of standardized method to chose a network architecture for a given task.

Neural nets can be used for classification which is to classify a given input into one of a finite amount of categories that the network has been trained against. Another option is to use the network for regression, in this case the output can be within a range of continuous values. Figure 5 shows a simple example of a training dataset that can be used in order to extract the location of a shape from an image. Although the example is trivial and it is easy to extract the location with a traditional algorithmic approach, it can be seen that the same problem can be tackled with both classification and regression.

In general, we use classification when the output takes discrete values or we have a number of classes from which we want to pick one. For example, if we wanted a neural network to predict the number of cars in an image, we would use a classifier whose output neurons will correspond to the discrete values 1, 2, 3 etc. On the other hand, if the task is to predict the future value of a stock or commodity, the output would be a single neuron whose activation can be any value.

In the case of figure 5, a classifier would have 3 neurons at the output layer where the one with the highest value would indicate in which part of the image is the shape located. In contrast, by using regression the network can be trained to predict the position of the shape by indicating a specific pixel index with a single neuron at the output layer.

Another important aspect of a neural network is its capacity. Capacity is called the maximum complexity of the function that the network can fit. It is closely related with the topology and the size of the network, a network with 2 hidden layers and 20 neurons for example has less capacity that a network with 200 neurons and 4 hidden layers. This is a very rough explanation but sufficient to let us introduce the problems of over-fitting and under-fitting of a neural net.

Given a training dataset, a neural network will under-fit if the architecture and number of neurons that were chosen is simply not enough to learn how to solve the problem, namely how to map the training set inputs to the desired output. On the contrary, if the network's capacity is much higher than the capacity that the problem requires, the network will be over-trained and will subsequently fail to generalize and work on previously unseen data.

By using a training dataset that contains images and labels as shown in figure 5, under-fitting will be experienced as an inability of the training algorithm to set the correct weight values, thus the neural net will have low success rate (accuracy) even on the training set. On the other hand if the network is big enough, over-fitting will result in very high success rate while training but at the

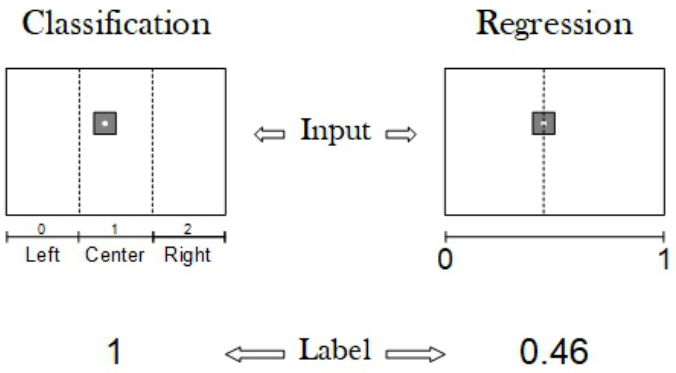


Figure 5: Example of a training dataset. This dataset could be used in order to detect the position of an object in an image. On the left, the dataset is structured in a way that allows us to use a classification network. By dividing the image into multiple segments we can train the network to predict in which of those segments is the object located, in the example we use three but the number of segments can be vary. The label in this case is an integer that points to the segment that the object is. On the right side the dataset is oriented into solving the same problem with regression. This time the label is a decimal value that can be directly translated into a specific pixel coordinate along the horizontal axis. Those two ways of formulating the same problem will have consequences on the architecture of the networks that could be used for those two examples. In the case of classification the neural network's output layer will consist of three neurons whose value will indicate the probability that the object lays into the corresponding segment. On the other hand, if regression is used then the output layer will consist of only one neuron whose output value will directly indicate the position of the object.

moment that the network operates on new images, different from the training set, it will fail to locate the shape.

To conclude, training is the process of tuning the parameters of a neural network which are usually carefully initialized so they do not introduce any bias. The purpose of the training is to create a neural network that will be able to generalize on a problem by only learning on a limited amount of examples. It has been proven that the bigger the size of the training set, the better the network generalizes on new data. However, training dataset generation is a time-consuming process as it usually requires humans to manually annotate the dataset. Usually a training set for computer vision will consist of millions of images that will come together with a vector with size equal to the number of images and will indicate the class of every image as a number. Some extensively used training sets are ImageNet, PASCAL and COCO[28, 4, 18].

A good source for further information regarding machine learning and more specifically deep feed-forward neural networks, is the Deep Learning book by Goodfellow, Bengio et al. [7]. This was used as an introduction to the subject for this project as well.

2.2 Convolutional Neural Networks

A fully connected neural network is powerful enough to tackle a wide range of problems but will require an enormous amount of parameters when it comes to computer vision problems. To illustrate this problem the MNIST database can be used as an example of a computer vision task that has to do with hand writing recognition. The database consist of 60000 training samples and 10000 testing samples where every sample is a grayscale image of size 28×28 pixels containing one handwritten digit. If a neural network with a first layer that contains only 40 neurons is used it will result to a high number of weights which is equal to $28 \times 28 \times 40 = 31360$, that is because every neuron will be connected to every pixel of the input image. The aforementioned amount of neurons is very small compared to the size of a neural network designed for computer vision. So, eventually the amount of memory needed to store the parameters will become the bottleneck of the system.

The number of parameters can be reduced if a weight sharing technique is applied. Convolutional neural networks are the most successful type of deep feedforward network in practice and they show remarkable results in dataset with strong spatial relation like images and sound recordings. They use small kernels (as defined in image processing field) in order to reduce the amount of parameters but the idea itself is not new. In traditional computer vision convolution of an image and a kernel is used in feature extraction where by applying a mask over the image it is possible to extract information about the edges for example.

In the past feature extraction was used as a first step to extract information about the content of an image and then pass it to an SVM in order to classify the content by using the extracted features. LeCun et al.[17] show how the hand writing digit recognition problem can be solved with a convolutional neural network, note that the study was published 20 years ago. Conv nets were never utilized widely only until recently, mainly due to some newly developed algorithms [6] that made training for big datasets feasible and because GPUs provide now the computation power that is needed for the training procedure.

Convolutional neural networks consist of a number of convolution layers and usually a smaller number of fully connected layers on top of them that produce the final output of the network. This is similar to the traditional approach with

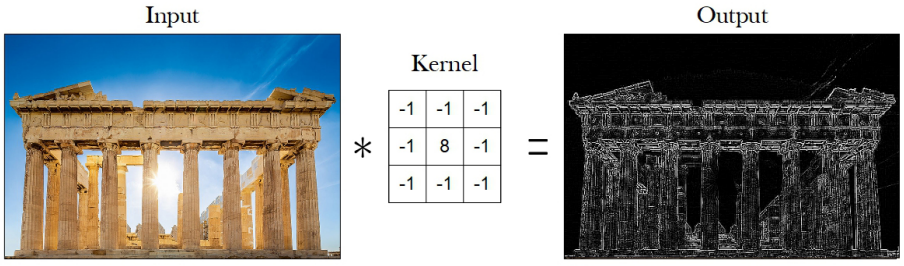


Figure 6: Convolution example of a color image with a 3×3 handcrafted kernel, also called masking within the field of image processing. In this example the coefficients are chosen for edge detection.

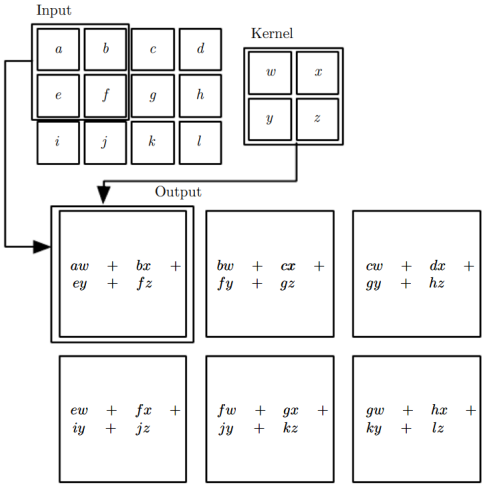


Figure 7: An example of 2D convolution. We perform 'valid' convolution as the output is restricted to only positions that the kernel lies entirely within the image. The number of pixels that the kernel is moving in order to calculate the next output value is called stride. In this example the stride of the convolution is one as the kernel moves one pixel every time. Illustration from [7]

the future extraction and an SVM on top but they differ because instead of hand crafted kernels, now the whole network is trainable from end-to-end and the all the parameters are subjected to the training algorithm.

Another function that is introduced in ConvNets is pooling. Pooling is when we sub-sample the result of a convolutional layer before it is forwarded to the input of the next layer. As a result it further reduces the amount of computation as it basically reduces the number of neurons without limiting its capacity. Pooling will be discussed in more detail in part 2.2.3.

2.2.1 The Convolution Operation

Convolution is a mathematical operation that can be applied on two functions and create a third one that will be a modified version of one of the functions. Under the scope of neural networks and image processing in general, convolution is the cross-correlation between an input multidimensional matrix and a smaller matrix (the kernel) of equal dimensions. The following equation describes the convolution of a two dimensional feature map I and the two dimensional kernel K. i and j are the width and height of the input and m,n are the width and height of the kernel.

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{1}$$

The output of the convolution is a two dimensional matrix called a feature map and the input can be a number of feature maps stacked as a 3D matrix or the input image for the first layer. Although the input image is a special case, by the convolution operation it is handled as a feature map because a color image is represented by a 3D matrix or in the case of a gray-scale image

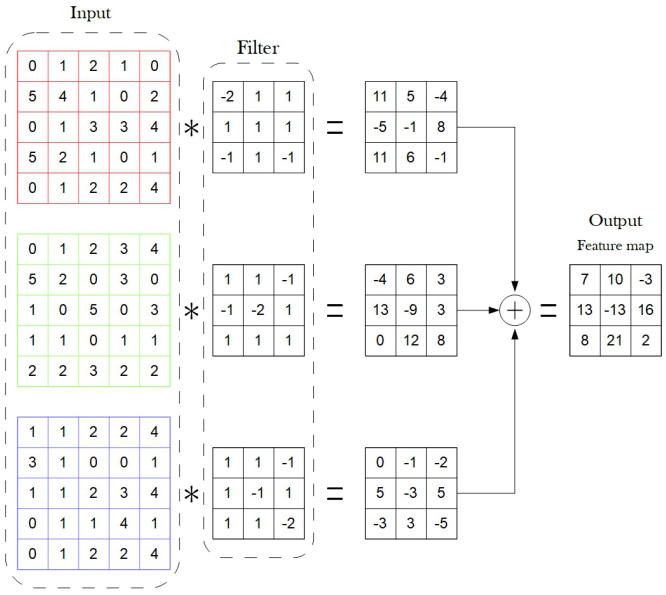


Figure 8: An example of 3D convolution with an RGB color image as input. As in figure 7 we perform a 'valid' convolution, thus the output is smaller than the input, in this case from an input of 5×5 pixels we get a 3×3 feature map. In general if 'valid' convolution is used then the output size is equal to $Input\,Size - Kernel\,Size + 1$ and this applies for both the width and height.

a 2D matrix which is similar to 3 or only one feature map. Figure 8 illustrates a convolution of a small input example that represents an RGB image with a simple filter with 3×3 kernels.

In the case of convolutional neural networks, each element of the resulting feature map can be thought as a separate neuron. Those neurons are the result of applying on the input the same kernel, thus we achieve weight sharing by using only a small number of weights that are shared among all the neurons that constitute a feature map.

2.2.2 Activation Functions

Although the main component of a convolutional neural network is the convolutional operation, there is still need for non-linear components in order to give the ability to the network to approximate complex functions. In this part some of the commonly used functions are listed. The use of these activation functions is not limited only to ConvNets but can also be used in traditional neural networks.

- The *Logistic function* is a commonly used S shaped function which belongs to the Sigmoid functions family. It has the property to limit the output in between two horizontal asymptotes as the input value approaches infinity. The function's equation is the following:

$$f(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1} \tag{2}$$

- *ReLU* is an abbreviation for rectifier linear unit, it turned out to be one of the most successful and widely used activation function. Basically it only lets positive values to propagate through the network and sets all negative values to zero. Thus, its equation is:

$$f(x) = x^+ = max(0, x) \tag{3}$$

- The *Softmax* function is a special case, its an activation function that is mostly used as the very last component of a classification neural network. The reason is that it has the ability to map an arbitrary vector into an equal sized vector that contains values between zero and one that have a sum of one. This is very useful as those values can be interpreted as a probability distribution over the possible classes. In the following equation K is the number of elements (classes) of the vector σ.

$$\sigma : \mathbb{R}^K \to (0,1)^K$$

$$\sigma(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \; for\, j = 1, \ldots, K \tag{4}$$

The aforementioned functions are some of the most frequently used but the list is not exhaustive as there exist many other functions with different properties and advantages. Because of the research that is still ongoing on the field of deep neural networks, new ideas and components are proposed by researchers constantly.

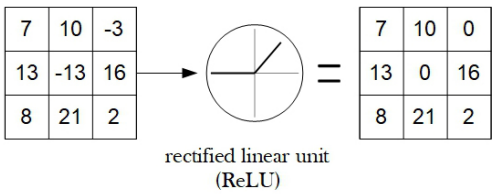


Figure 9: Example of the ReLU activation function. The input to the function is the example convolution's output of figure 8. The functionality is simple as all the positive values remain the same whereas all negative values are replaced by zero.

2.2.3 Pooling

Pooling has became an essential building block for ConvNets as it helps in minimizing the computational burden without loosing information on the process. A pooling layer operates over a feature map and creates a more compact representation by forwarding a single value out of an area of neighboring values. Although there are multiple variations of the pooling layer, figure 2.2.3

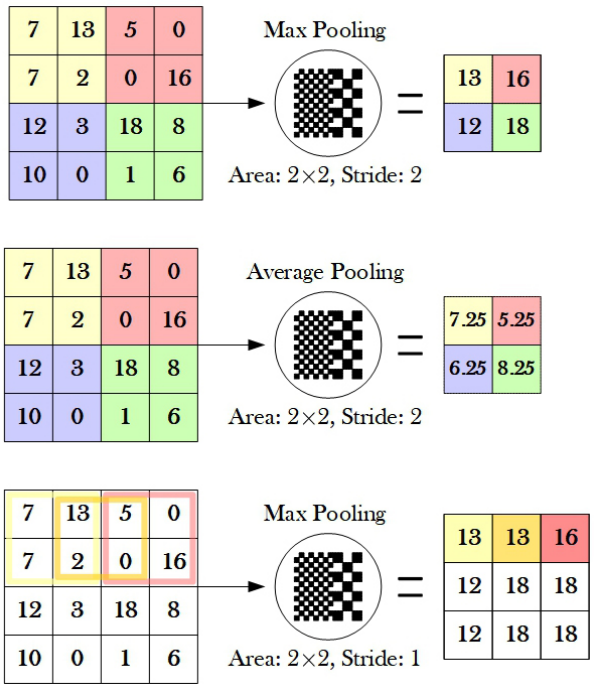


Figure 10: Some pooling examples. The two upper examples show the result of max and average pooling over an area of 2×2 and with stride 2, the area from which the resulting value is extracted and the value itself have the same background color. The last example at the bottom of the figure depicts a max pool over the same area as before but with stride 1. This time the output has bigger dimensions as the areas now overlap due to the smaller stride, this is also the reason that we don't use background colors but instead we use only a few boxes for clarity.

depicts *max* and *average* pooling which are the most used among convolutional ConvNets. As their name suggests, in the case of max pooling we extract the maximum value of an area whereas in the case of average pooling we calculate the average value of the area.

It is important to note that depending on the objective of the CNN it may be beneficial to choose average pooling over max pooling due to its property to preserve the location of a feature in a better way, thus giving better results when localization of an object is needed. On the contrary, average pooling will require more computation in order to calculate the average value of an area than max pooling which only extracts the maximum.

2.2.4 Convolutional Neural Networks for Classification

After presenting the basic building blocks of a convolutional neural network we can now investigate some example architectures in order to understand how those components interact in order to compose a robust classification or detection system. Figure 2.2.4 illustrates the usage of the convolution followed

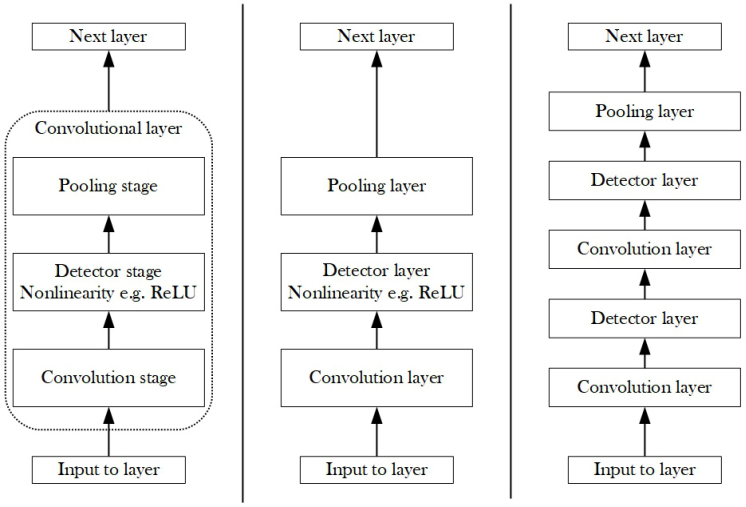


Figure 11: The components of a typical convolutional neural network (left and center) and an example of more a complex arrangement (right). In some cases a convolutional layer is viewed as a convolution followed by an activation function and sub-sampling (Left). More recently though, due to the number of complex architectures that have been proposed, this terminology is not used often. In the most common terminology every stage is regarded as a separate layer (center), in this way we can more easily describe network architectures like the one shown at the right part of this figure where pooling is not applied after every activation (detection layer). This illustration was inspired by figure 9.7 of the Deep Learning book[7].

by the non-linear activation and the pooling operation, the *convolution* $\rightarrow$ *activation* $\rightarrow$ *pooling* arrangement is usually called a convolutional layer. Another common arrangement is the stacking of multiple convolution-activation blocks before the pooling operation. Some of the state-of-art networks that will be presented in this part make use of this topology.

- *Alexnet[16]* was the first neural network to win the ImageNet competition back in 2012. Developed by Alex Krizhevsky et al. it was based on the first ever convolutional network we previously discussed, namely the LeNet, introduced by LeCun et al. at 1998. Although from an architectural point of view they are very similar it was the availability of GPUs for general computing that made the training of such a network feasible for a multi class classification problem by training against millions of annotated images.

- *VGG[31]* is a family of ConvNets developed by the Visual Geometry Group of the University of Oxford. Its most popular variations are the VGG-16 and VGG-19, where the number that follows indicates the depth of the network in layers. VGG is used by many other studies due to its simple architecture and the good results at the ImageNet challenge of 2013. Its main drawback is the vast amount of computation it requires, namely 16.9 GOPS for the VGG-19.

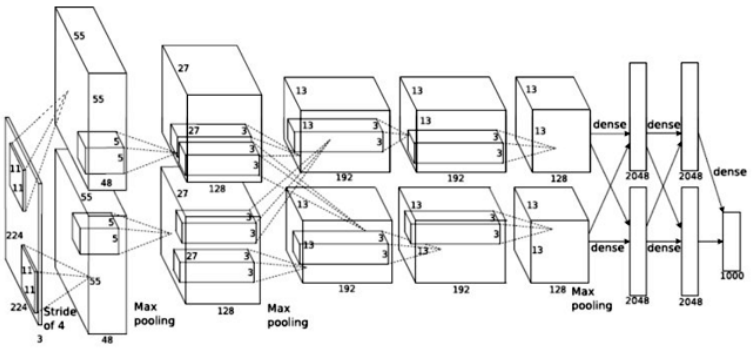


Figure 12: The Alexnet CNN[16]. It consists of 5 convolution and 3 fully connected layers. The network has two branches because it was trained on two separate GPUs, one branch on each GPU. With the use of newer libraries there is no reason to split a network into different parts. Note that the first convolution layer consists of kernels with dimensions of 11×11, the second of 5×5,and the rest of 3×3.

AlexNet and VGG are CNNs with relatively simple structure because the layers are stacked after each other and data flows in one path. In the following years after their initial success it became obvious that deeper networks are not necessarily more accurate so new techniques were introduced in order to increase the capacity of the networks but not limit their train-ability which was the main issue with very deep architectures. The problem is that during the back-propagation process the very first layers of a deep network will not be sufficiently trained (vanishing gradient problem) so there was need for developing new ways that will allow even deeper network to achieve state-of-the-art accuracy.

The solution is to make wider architectures, not only deeper and to allow layers to get as an input not only the output of the one previous layer but rather have as an input information from multiple previous layers. The networks that are listed bellow are noteworthy as they introduced some very interesting ideas. With those ideas it was possible to lower the classification and detection errors even further and reduce the overall size of the networks as well. This means that by using more sophisticated and complex architectures, the number of parameters of a network can be reduced without an accuracy penalty.

- *ResNet[9]* was a groundbreaking network as it introduced the idea of building networks that are based on more complex structures (network-in-network), these ideas reduced the error on the Imagenet classification challenge from the 25% of AlexNet in 2012 down to 5% in 2015, the year ResNet was introduced by He et al. The main advantage of the residual network is that it allows even deeper networks to overcome the learning problems that arise when the number of layers is large. This was made possible by using the residual module which is depicted in figure 2.2.4 , another achievement of this conv net is the reduced size in comparison to the VGG networks for example. Although the ResNet is 51 layers deep it need only 100MB of storage whereas the VGG networks need 530-570MB, depending on the variation. Furthermore, some variants require 3.6 GOPS,

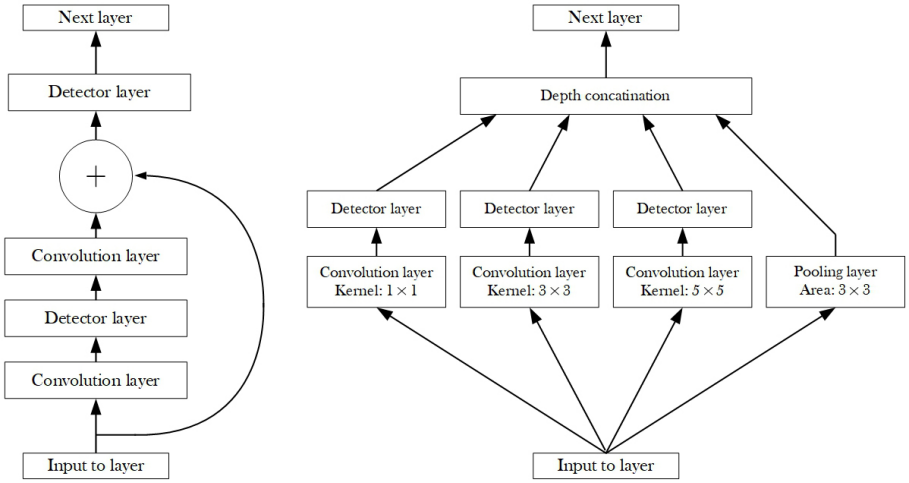


Figure 13: An example of a residual learning block (left) and of an inception module (right). Both architectures help the training process of deeper networks. A rough explanation of how this is achieved, is that the training algorithm has the ability to chose the path that conveys more useful features or even combine them with features of previous layers.

a number which is much smaller from the VGG ConvNets for example.

- *Inception V4[32]* is yet another interesting network from an architectural point of view. With this convolutional network Szegedy et al. introduce an other module that is used in order to produce features from various sized kernels (1×1, 3×3 and 5×5 convolutions) in every layer and then concatenating the result before it is fed to the next layer.

- *MobileNets[10]* are a family of neural networks (developed by Google's engineers) that targets mobile devices with limited resources. It has the ability to be configured according to the user's needs. The depth and the number of parameters of the network can be decreased if the target platform is not powerful enough to cope with the full network. What make MobileNets to require less computation than the previous networks, is the use of point wise convolutions (1×1 kernel) in order to create a more compact representation of the features that are being extracted with 3×3 convolutions. A version of the MobileNets was used for the object detector and proved to be very robust in the single-object detection task, this is further discussed in Chapter 3.

Although the aforementioned CNNs are trained in order to classify multiple objects, understanding the different particularities of the state-of-the-art convolutional networks becomes very important when the goal is to develop a robust and powerful object detector. The main reason is that by reusing those networks, a technique that is called Learning Transfer, we are allowed to use a smaller training set and still be able to achieve remarkable accuracy in any

environment. On the other hand, reusing such a network will require an accelerator capable of performing the required computation. In the following part we introduce some of the object detection techniques that have been based on CNNs.

2.3 State-of-the-art CNN Based Object Detectors

After the initial success of CNNs for classification it was natural to use them for the task of object detection as well. In classification challenges the goal is to predict which is the main objects of an image. Usually, the object covers a large area of the image and is in the center. On the other hand, detection is the task where we predict which objects lie within the image and furthermore specify their locations. This makes the CNNs trained for the Imagenet classification challenge not directly useful for detection.

The initial approach to bridge the gap, was to extract areas of the image with a sliding window of various scales and evaluate the result of the classifier. This approach requires an enormous amount of computation and does not offer real-time performance due to the multiple evaluations of the CNN. Another approach is to pre-process the extracted areas and list them based on the probability that they include an object. After the selection of the most prominent areas the CNN is evaluated only on those and not all the initially extracted areas. This method is minimizing the computational burden but not enough in order to make it suitable for real-time systems.

Convolutions have the property to preserve information about the location of the features that are being extracted so the next generation of object detectors make use of this property by training the network to directly predict bounding boxes. The following CNN based object detectors represent the two aforementioned methods and are worth our consideration for the object detector that has to be developed.

- *Region-based CNN* (R-CNN)[5] is a very well known object detector that was originally based on AlexNet for feature extraction. Instead of the sliding window method, it uses selective search in order to extract about 2000 regions that are then evaluated with the classifier. On top of the classifier, an SVM determines the existence of an object and its location. The first version was not fast enough so as to be a candidate for real-time systems. It also suffers from not being able to be trained from end-to-end in order to improve performance. Fast R-CNN and Faster R-CNN are newer versions that introduce improvements on those weak points and also use less regions for speed-up. The improvements include the use of a region proposal network (RPN) for region proposal and also the ability to train the detection pipeline ($RPN \rightarrow CNN \rightarrow SVM$) all together.

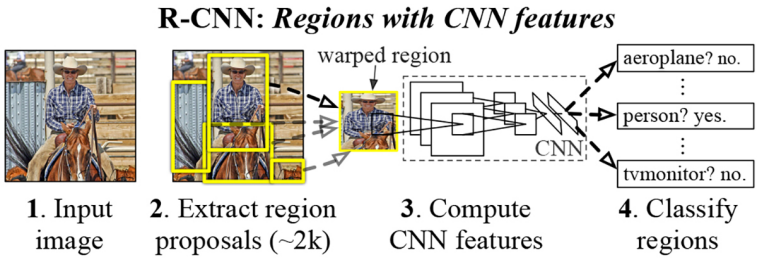


Figure 14: R-CNN object detector pipeline[5]. The first step is the extraction of regions that have the higher probability to contain an object, after the selection those regions are resized to fit the classification CNN and then evaluated. Finally, the results of the evaluations are combined in order to create bounding boxes around the detected objects.

- *YOLO[27]* is a object detector which treats detection as a regression problem in one single CNN that outputs bounding boxes and class probabilities. This is done by segmenting the image into multiple areas, where each area can contain a fixed amount of bounding boxes. For each box of each region, the output layer will predict a pair of coordinates (x, y) for the center of the box, its width and height (w, h) and a confidence value. Furthermore, for each area the neural network makes a class prediction. The last layer's output vector has a length of $A \times (B \times 5 + C)$, where A is the number of areas, B is the number of boxes per area and C is the number of classes that the CNN can classify. Note that B is multiplied by 5 as each box has 5 variables $(x, y, h, w, confidence)$. The final step is to combine the predicted boxes and classes of each area in order to create the final result . By evaluating the CNN only once there is a big benefit regarding the computation which is required, so YOLO is the most prominent candidate for real-time systems. This approach has been proved to deliver robust results, comparable with the region based detectors but its much faster. Lately, more and more object detectors are based on similar methods which extract the location of the objects directly through the convolutional layers.

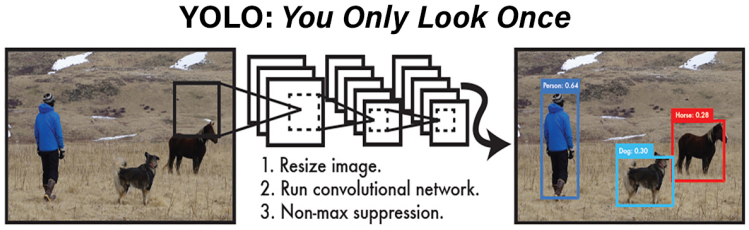


Figure 15: YOLO object detector pipeline*[27]*. As YOLO is a 'one shot' object detector, we only need to resize the input image and then evaluate the CNN. The last step is to use non-maxima suppression in order to filter the boxes that are not valid detections.

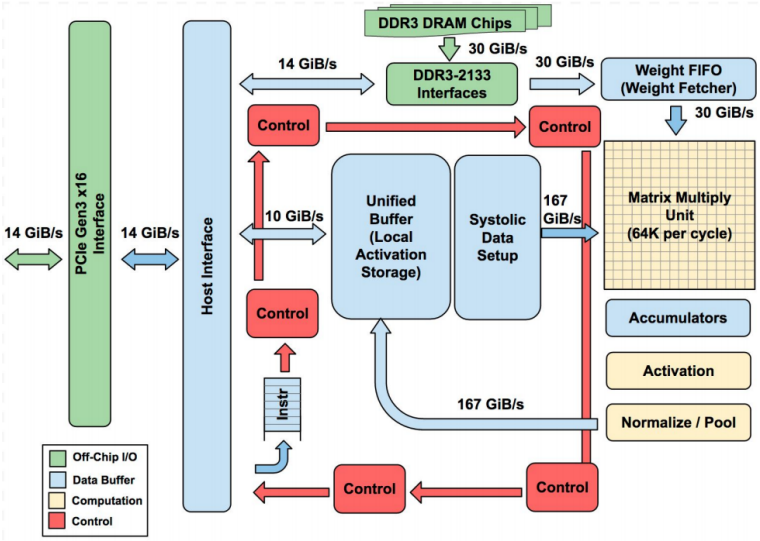


Figure 16: TPU block diagram [14]. Note that most of the components serve as data buffers due to the very high demand for local memory in neural network accelerators. The core of the system is the matrix multiply unit and everything is built around it in order to fetch the parameters and the data but also to write back the result.

2.4 Convolutional Neural Network Accelerators

The rapid development of convolutional neural networks in the recent years is also attributed to the introduction of GPUs into general purpose computing. Of course big training datasets and new training methods along side new architectures played a significant role as well, but in absence of sheer computing power those achievements would not be available to everybody.

In this part we discuss some of the hardware architectures that are designed to accelerate the training and the inference of neural nets. In general such specialized hardware can be faster and is more energy efficient than the GPUs, CPUs on the other hand are not the optimal platform in any way for neural networks, as they fail to exploit the parallelism that exists in neural network computation.

Although researchers published some very specialized hardware for ASICs and FPGAs, it is be noted that big tech companies nowadays are behind many of the recent achievements and also provide a lot of open sourced tools and training sets. The same companies will usually offer cloud computing platforms in order to make the usage of their software possible even in less capable devices like the Raspberry Pi 3. In the case of an computer vision application for example, the image would be acquired from the camera and then transmitted to the cloud platform, the computation will take place in the data-center and only the answer will be transmitted back to the embedded device.

Tensor Processing Unit (TPU)[14] is the ASIC designed by Google in order to allow its datacenters to perform the computation during inference of deep neural networks. Although the first version was not able to assist during the

training phase, the second version that was recently announced will be able to do both inference and training. In figure 16 illustrates the block diagram of the TPU, it can be seen that the basic block of the device is the matrix multiplication unit which performs the convolutions. A significant amount of area is dedicated to the connection of the computing blocks to the weights and data memory but most of the area is allocated for the internal buffers and the matrix multiply unit.

Microsoft on the other hand decided to deploy FPGA based solution in order to provide cloud DNN computing that is targeting only the inference of real-time applications. The main reason behind the choice of FPGAs is the flexibility that they offer for future improvements and the changes that can be done regarding the datatype of the accelerator.

3 CNN based Object Detection

This part documents the development of a suitable training dataset for the prey - predator application and the training procedure of the CNN together. We also show the results we got from the different classification architectures that were tested. Furthermore, it is shown that those architectures are suitable for basic object detection and this can be further improved if we combine our neural networks with a pre-trained CNN.

Due to the iterative development, all the components of the work were re-designed and evaluated multiple times, especially due to the close relation of the CNN functionality and the accelerator. Here, presented is only the final state of the development

3.1 Training Dataset Generation

An artificial neural network training may be unsupervised or supervised. The former means that we define a cost function and we let the back-propagation algorithm minimize the cost by working on the task without any previous knowledge or example. During supervised learning on the other hand, we also define a cost function but now we first provide examples of the desired output and we hope that the network will imitate the behavior on new inputs as well. For this thesis we train the CNN in a supervised fashion as this is the common case in literature for image classification and has been proven to deliver good results, whereas unsupervised learning is more challenging to apply.

In order to perform supervised training there is need for a big number of examples in order to tune the parameters for maximum performance on unseen data. Moeys et al. created a training dataset of 500.000 examples by recording 1.2 hours of video and manually annotating each frame to indicate the ground truth position. The video recordings were acquired directly from the image sensor of the robot which was remotely controlled for this purpose. In this thesis there was no constraint on the environment that the robots could operate so the goal was to be able to detect the robot in any environment. This rises the demand for an even bigger training dataset that will include many and also diverse examples. Additionally, at the beginning of the project the image source was not specified so the option of recording video from the robot itself was not available.

To overcome those issues so as to quickly start experimenting, it was decided to create an artificial dataset by using a small number of images from the robot in combination with a larger number of random background images. In order to have the GoPiGo in various poses that we can combine with the different backgrounds, it was first needed to extract only the robot like in figure 17 . Then we save it in an file format that can also include transparency information, in this case the PNG file format was selected.

After the extraction of 50 images, we mirror them in order to create a total number of 100 PNG files that contain only the GoPiGo robot. The next step was to download about 1500 random images from the Internet and combine them by overlaying the robot's pictures over them. The motivation behind the choice of random backgrounds is that our goal was to create a system that would work in any environment. So, by presenting to the network a large number of images that contain shapes and colors that can be found around us. We will

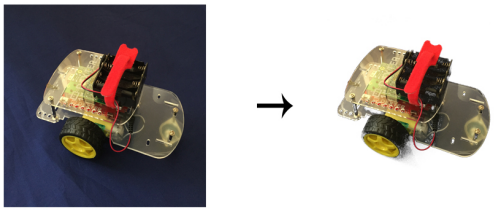


Figure 17: Extraction of the GoPiGo robot. It can be seen that the extracted picture of the robot is influenced by the environment due to the transparent material that it is made from and furthermore, by the lighting conditions as well.

help the network to not correlate the GoPiGo with features that can also be extracted from other objects as well. To make the dataset even more versatile, the robot's pictures are randomly resized and rotated in addition to altering the color and lighting properties of the generated image. Some of the images used are shown in figure 19 together with samples of the generated training images and the corresponding label. The generated images are in most cases unrealistic but that is not a disadvantage as we want to extract features that do not correlate with the context of a scene.

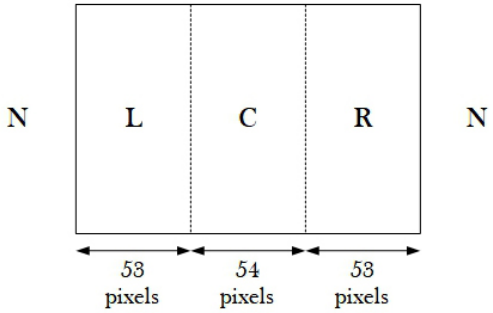


Figure 18: Labeling of the different regions that the robot may lie within. L, C, R are for the left, central and right part of the image whereas N indicates the absence of the robot. The dimensions of the images we used to train of our CNNs are 160×120 pixels.

The great advantage of this method is that as far as the script places the robot over the background, the pixel coordinates are already known. Thus, independently from how the problem is defined, the ground truth is available and can be directly used for the creation of the training dataset. The output product of the generating script are two matrices, a 4D matrix which contains a number of color images and a second 2D matrix which contains the ground truth for every image that the former matrix contains. In this specific case we treat the problem with classification so the ground truth value is an integer which indicates the output class for each image. Figure 18 shows how the image is divided into 3 regions, namely the Left, Right and Center. In order to describe

Figure 19: Example of images created by the training dataset generation algorithm. The ratio between the GoPiGo and background images is altered in this illustration. In reality, the number of GoPiGo photos is much smaller that the backgrounds we have used. The capital letters on the left and right side of the generated images are their corresponding labels. Finally, two examples with the same background are shown in order to emphasize on the different color and lighting settings.

the absence of the robot we need a fourth class, this means that the classifier will be trained to classify each frame into those four categories. Finally, for better results during the training phase it is beneficial to have an equal number for examples per class, so each category (Left, Center, Right, Non-visible) should be represented ideally by 25% of the images in the training dataset.

It was chosen to follow this method (LCRN) which was found in the work of Moeys et al. The reason is that the initial approach to the problem was to train a network from end-to-end, thus the output of the network would directly drive the robot. This is the simplest way to tackle the predator - prey problem but is not optimal due to the inability of this simple solution to preserve information from the past frames. However this method was proven to be a good starting point and the classification networks that were trained with this training dataset were also used in for localization as we discuss in part 3 of this Chapter.

3.2 Training a Simple CNN

The availability of the training dataset lets us move to the next objective of this part which is the training of a CNN on this classification problem. It was decided to perform the training procedure offline. This is to define and train the neural network on a workstation rather that the ZYBO board itself. Similar to the development of software for embedded platforms which is done on a workstation. The training procedure of a CNN is a very heavy workload due to the large number of CNN evaluations that need to be done, thus it is convenient and in some cases unavoidable to train the ConvNet on a workstation.

In this part, the terms *architecture*, *topology* and *structure* of a CNN, all refer to the specific way that the layers are arranged. The term *model* on the other hand, is used to describe both a CNN architecture and the trained weights that constitute its kernels and neurons in general.

3.2.1 Tools

Due to the even growing interest in machine learning recently, there is an abundance of tools that somebody can use in order to train and use ConvNets. As mentioned previously, the big tech companies like Microsoft, Amazon and Google were the first to utilize machine learning for commercial applications. This led them to develop tools that were firstly used internally but later were given to the public as well. Tensorflow[1] for example was published by the Google Brain team as an open source library in 2015 and is currently used by many machine learning practitioners. It provides APIs in many programming languages and also supports the use of NVIDIA's GPU's for parallel computing. Big companies do not directly profit by the tools they develop as they are mostly open sourced. As we discussed in part 4 of the previous chapter though, they offer the possibility to accelerate on their cloud computing platforms the training phase or even the inference in case of an embedded device and those services are not for free.

Except the big tech companies, tools are developed by academics as well. Theano for example is also a computation library for Python that was developed by researchers at the University of Montreal. Its development will not be continued though due to the strong competition by the aforementioned industrial players. Another commonly used framework is Caffe which is developed

and maintained by researchers of the Berkeley University.

For this thesis we make use of the Tensorflow library but not directly. The reason is that Tensorflow is regarded as a low-level library and requires the user to have some prior knowledge on the training algorithms. This was not the case for this project, so the Keras library was used on top of Tensorflow in order to simplify the training process.

Keras is an open source library for Python which has the advantage of being well documented and constantly maintained by its developers. Thus, it offers a friendly environment for inexperienced machine learning practitioners by hiding a lot of the complexity that Tensorflow needs in order to set up the training algorithm and tune its parameters. As Keras is simply an interface between the user and the computation libraries, it can be used with Tensorflow, MXNet, Microsoft's CNTK and Theano as well.

3.2.2 Hardware Design Considerations

After the initial trials on CNN training, it became apparent that we should define a small set of supported functions due to the multitude of options that are available. Because the final goal of the thesis was to accelerate the CNN computation on the resource limited ZYBO board, it would have been very ambitious to start the development of a fully flexible architecture which would support all the available functions and their variations. Those restrictions though, apply only to the convolutional part of the networks. This is because the computation of the fully connected layers at the end of the network can be handled by the ARM cores of the Zynq SoC. Thus, more complex functions can be performed by writing C code.

In order to define this sub-set of supported functions we take into consideration how commonly they are used in general and the difficulty of implementing them on an FPGA. In the case of the convolution which is the basic block of every CNN, the decision was to constrain the kernel size to only 3×3 with a stride of 1 pixel like the example in figure 8 . This convolution configuration is the most commonly used among the state-of-the-art ConvNets and in some cases like the VGG CNN, it is the only one.

This is in contrast to the Inception CNN that utilizes convolutions with various kernel sizes for example. As a consequence to this decision, many of the state-of-the-art CNN will not be able to be accelerated. However, on the other hand the design of the convolver becomes easier due to the absence of a sophisticated interconnection network between the multipliers that would have been required if we had to support multiple kernel sizes.

The pooling function was decided to only extract the maximum value from an area of 2×2 with a stride of 2 pixels, similar to the upper example of figure 2.2.3. The reason behind this decision is again the fact that this variation is the most commonly used and furthermore it usually results to feature maps with even dimensions which is convenient and requires simpler controller on the FPGA. Usually, the width and height of images acquired from cameras have a ratio of 4/3, for example 640×480 or 1024×768 are common resolutions for small size images. By using this variation of the pooling function the result after sub-sumpling a 640×480 feature map will have dimensions of 320×240 which are again even and eventually easier to handle.

Another design choice towards this direction, is to support zero padding on

Function	Restrictions
Convolution	Kernel size: 3×3, Stride: 1, Zero padding: optional
Activation	ReLU only
Pooling	Area size: 2×2, Stride: 2

Table 1: A summary of the functions that the accelerator can support.

the feature maps before the convolution stage. Because convolution will always reduce the size of the output as explained in figure 7 . It is possible to keep the output dimensions similar to the input's if we first add zeros around the input feature map. This type of convolution is called "same" usually. For example, if convolution is performed on a 640×480 feature map, it will result to an output with dimension of 638×478. By using zero padding we first increase the size of the input to 642×482 and then we perform the convolution which will bring back the dimensions to the original 640×480.

Finally, the activation allowed on the convolutional part of the CNN is only ReLU and this is due to the simplicity of implementing the hardware to perform this activation. ReLU lets all positive values pass and only sets all negative values to 0. In hardware the sign of a value is described in a single bit, where logical 1 indicates a negative value and 0 indicates that the value is positive. Thus, ReLU can be implemented by only driving a multiplexer with the sign bit which is very fast and cheap regarding the resources needed. Luckily, ReLU is the mostly used activation and delivers good performance during training as well.

3.2.3 The Architectures

Inspired by the work of Moeys et al. the architectures defined for our problem are very similar to the ones that they have tested for their application. By obeying to the restrictions as listed in table 1, we specify similar architectures for our CNN. In contrast to their work, those shallow architectures do not achieve high accuracy on our training dataset though.

An explanation may be that our training dataset is more complex due to the random backgrounds we use. Additionally, they blend the input image with information from a Dynamic Vision Sensor. Therefore, there is a substantial difference between the dataset they used and ours. However, eventually we were able to achieve higher accuracy by specifying deeper neural networks with more than two convolution layers.

The training dataset consists of 38000 samples and we use another 2000 samples for evaluation. The total number of 40000 images was the upper limit due to the RAM usage of the Keras library which for this number of samples is 10GB. This amount of RAM is needed because instead of using 8bit integer values for the pixels, we have to scale them in between 0 and 1 which requires the usage of floating point values of 32bits. This is due to the faster convergence that we achieve during the training procedure.

By utilizing an NVIDIA GeForce GTX 960M, each iteration over all the training samples (also called epoch) takes about 2 minutes, depending on the number of parameters that each model has. The optimization algorithm that

Model A	Model B	Model C	Model D	Model E	Model F	Model G
Conv 4	Conv 8	Conv 4	Conv 4	Conv 8	Conv 8	Conv 8*
Conv 4	Conv 8	Pool	Pool	Pool	Conv 8	Conv 8
Pool	Pool	Conv 8	Conv 8	Conv 16	Pool	Pool
Conv 8	Conv 16	Pool	Pool	Pool	Conv 16	Conv 16*
Conv 8	Conv 16	Conv 16	Conv 16	Conv 32	Conv 16	Conv 16
Pool	Pool	Pool	Pool	Pool	Pool	Pool
Conv 16	Conv 32	Conv 4	Conv 4	Conv 4	Conv 4	Conv 32
Pool	Pool	Pool	Pool	Pool	Conv 1	Conv 4
Conv 8	Conv 4	Full 64	Full 64	Full 64	Pool	Pool
Pool	Pool	Full 16	Full 16	Full 16	Full 32	Full 32
Full 128	Full 64	Full 4	Full 4	Full 4	Full 4	Full 4
Full 32	Full 16	-	-	-	-	-
Full 4	Full 4	-	-	-	-	-

Table 2: Topologies of the CNNs that were used for the accuracy comparison of figure 20. All the models are simple feed-forward structures without any branch or feed-back loop. The first layer is the one at the top of each column while the last layer is always the fully connected layer (Full 4) at the bottom. Thus, the flow is from the top to the bottom of each column. The asterisks (*) at model G convolution layers indicate the absence of an activation function directly after the convolution. In any other case of this table, the convolutional layers are always followed by a ReLU and the fully connected layers by sigmoid or softmax activation. Furthermore, the number next to the "Conv" indicator is how many filters are applied on the specific convolution layer. The rest of the parameters of each layers follow the rules as defined in table 1.

was used is 'adam'[15] together with 'categorical crossentropy' as the loss function. The result of the training procedure of some example models are shown in figure 20 while the structure of each model is described in table 2.

Although the accuracy that the models achieve during training is very high, the networks do not perform so well when evaluated on new images coming from a webcam or a mobile phone. The main issue is the big number of false positives. This is when the GoPiGo robot is not in the image but the network predicts that it is left (L), right (R) or in the center (C) instead of the correct non-visible (N) class. In this case we call predicted class the one that had the biggest probability, so it might be that in some cases the network is not strongly suggesting a wrong class but we treat this as a misclassification.

The bad generalization of the models from the beginning of the experiments led to the continuous improvement of the dataset generation algorithm. In order to create a more diverse dataset we randomly select values for parameters like the color, the lighting, the rotation and size of the robot. This gradually led to better generalization but it was observed that some characteristics of the robot like the red band that holds the batteries, led to many red objects be misclassified as being the GoPiGo robot. As a result the backgrounds were chosen to include

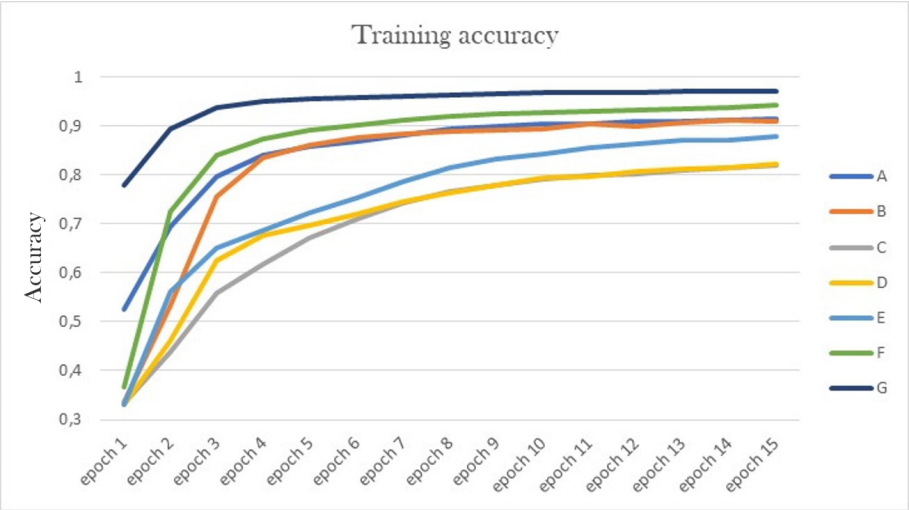


Figure 20: Accuracy comparison between various models. The trend we observe is that topologies with more convolutional layers perform better on the generated training dataset.

more red objects which led to a further improvement on generalizing but the final result is not satisfying in general.

Even after those improvements the ConvNets we trained behave in an unpredictable way when the robot is not present in the input image and a false positive can be triggered by objects that are not similar to the GoPiGo. The only positive outcome is that given that the robot is in the image, then it will almost always draw the attention of the network independently of what other objects are there. Later it is shown how this behavior helps us to build an object detector by using a pre-trained network so as to cope with false positives.

A final remark on the training procedure is that while versatility of the training dataset was rising, the training of the CNNs became more difficult. With the latest version of the generation algorithm, some models would get stuck during training and predict always the same class. By trying different combinations of optimization algorithms and loss functions, eventually it was possible to train the CNNs. It is to be noted though, that it was the switch from ReLU to the sigmoid activation function for the fully connected layers that made the training a lot more predictable and easy.

Figure 21 gives a more detailed description of 3 chosen topologies, namely models B, C and G. Note that model G which is the best performer on the training dataset has convolution layers that are immediately followed by another convolution layer. That is very beneficial for the performance as this model converges much faster that all the other topologies while it reaches a 97% of accuracy which is the highest. The reason may be that by stacking two convolution layers after each other we increase the receptive field of the second one. That means that we increase the area that a convolutional neuron of the second layer can 'see' through the first. Doing so in our case, compensates the inability to use larger than 3×3 kernels. If an activation function is in between the convolution layers, this advantage is reduced.

Finally, figures 22, 23 and 24 depict the feature maps that are being generated after each convolution layer of model C. The choice of model C derives from the fact that it has less convolution layers and the spacial information is better preserved. We show the results of evaluating the CNN with three images which have been generated with our algorithm.

It can be seen how the output feature maps start with simple edge detection and color filtering at the fist convolutional layer. But, the last convolution layer on the other hand shows a much higher level of abstraction where high values indicate the presence of the robot. In general, looking into the output of the hidden layers is very helpful in understanding what the convolutional neural networks 'sees' and improve the practices that one is using to make it better.

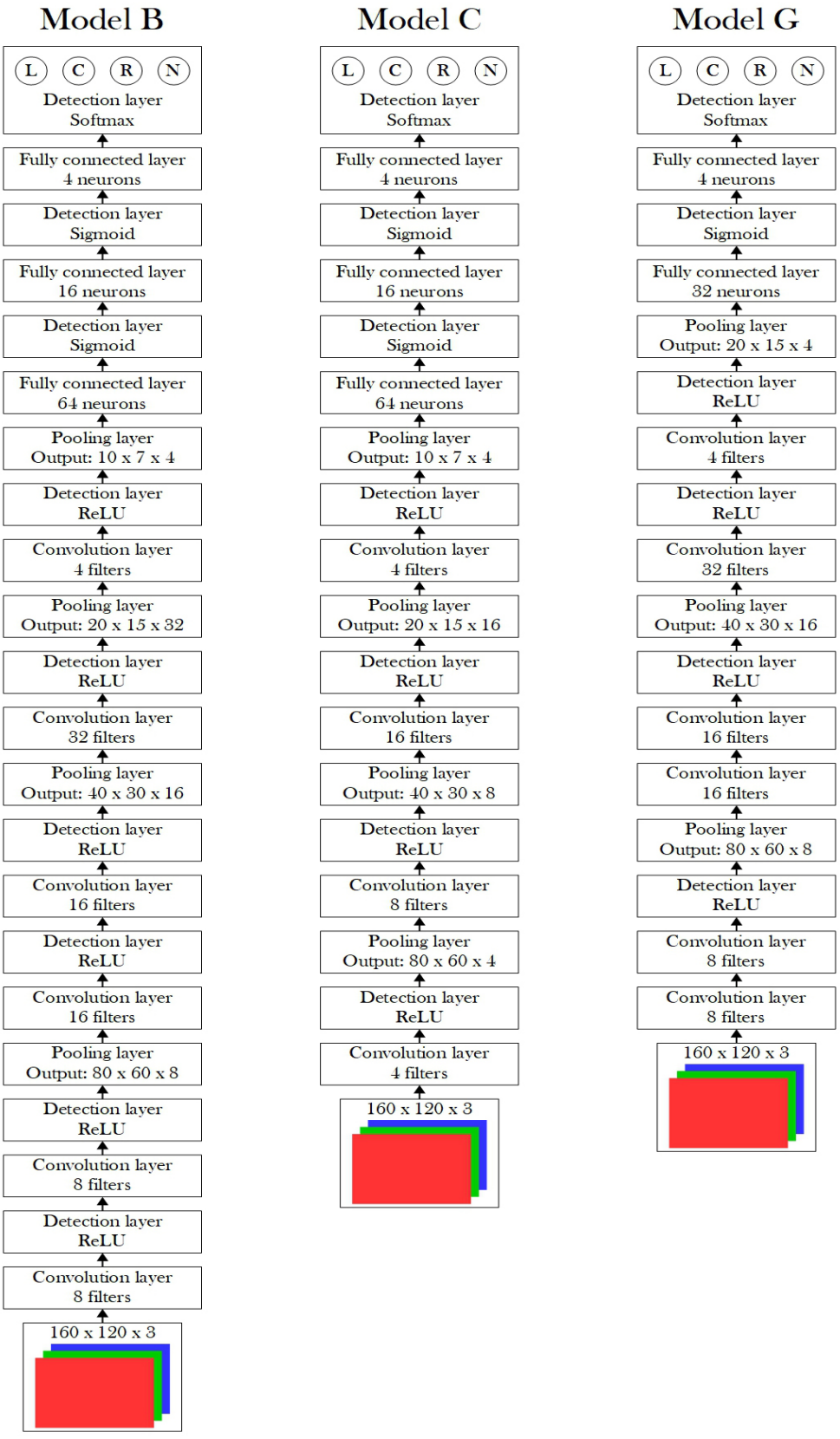


Figure 21: Some of the architectures that were tested, in more detail. Each of he three models depicted in this figure represents a different approach regarding the sequence of the layers. Only the dropout layers are omitted from the figure because they do not influence the computation during inference.

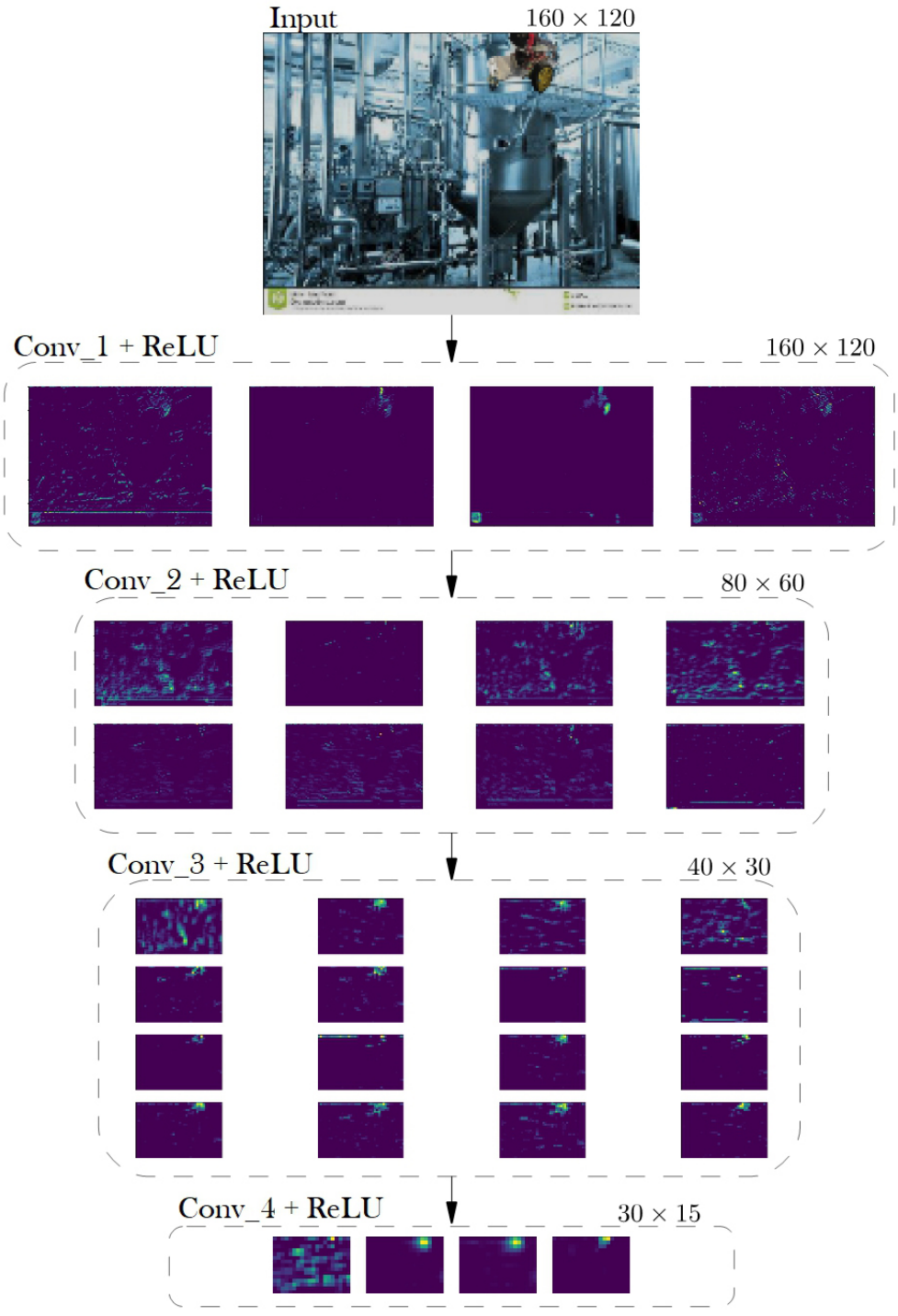


Figure 22: The feature maps generated by the intermediate convolution layers of model C. The model can easily predict the position of the GoPiGo robot in this example. This can be seen by the activations which are high only around the area that the robot lies. Note that each feature map's values have been normalized. As a result, the same color does not represent the same activation value in different feature maps.

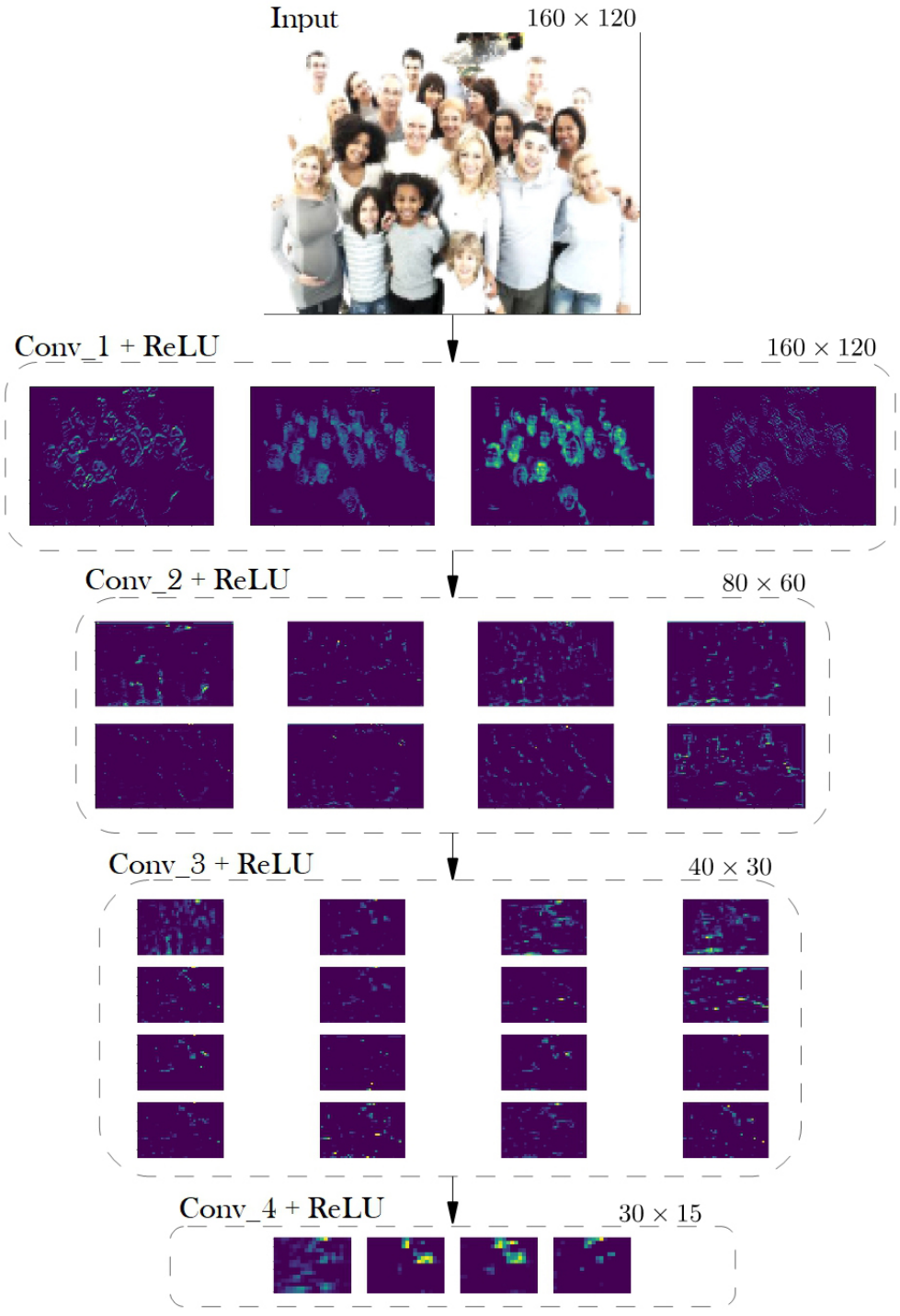


Figure 23: Another example of feature maps generated by the intermediate convolution layers of model C. This example is more difficult for the model as the robot is not fully present in the input image.

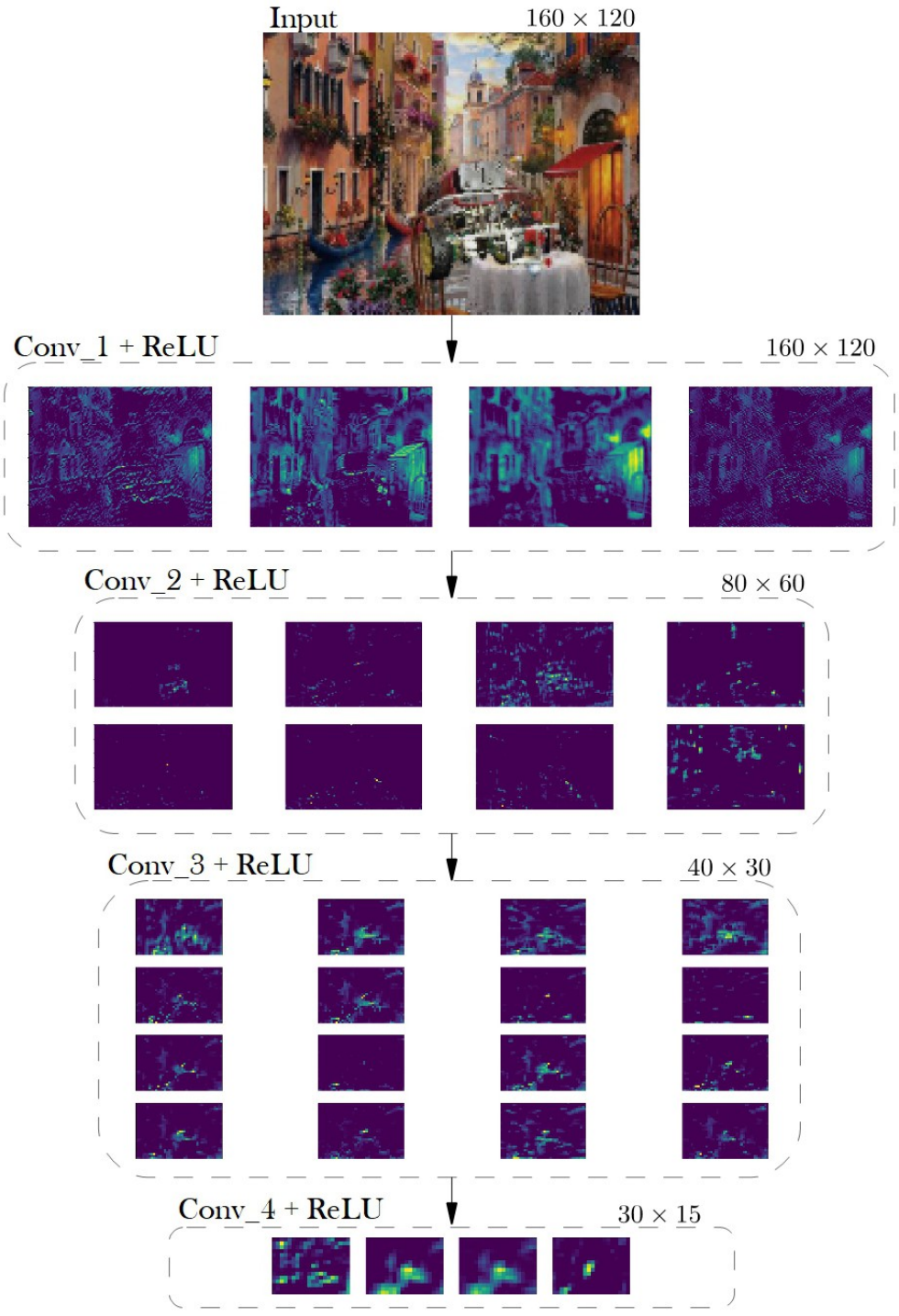


Figure 24: Yet another example of feature maps generated by the intermediate convolution layers of model C. This last example is also difficult for the model as the robot is blended in to the background and cannot be easily distinguished.

3.2.4 Training Dataset Evaluation

The experience of training different ConvNets on the generated datasets suggests that they are not suitable for small CNNs and result to bad generalization. Contrary to the large number of false positive predictions that we get if we run the ConvNet on real images, figure 25 shows that this is not the case with the generated ones. We see that for all the models which are depicted in figure 25 the classification error is dominated by misclassification between the positions of the robot. Surprisingly, the existence or not of the robot is predicted almost perfectly even on a newly generated dataset, especially for model G. This leads us to the conclusion that the models overfit on the generated dataset.

The later can be confirmed by the observation that the more we train the models, the worse they perform. Any model we train for a lot of epochs will eventually end up unable to detect anything on real images. Even if the robot is in the images the output is stuck into the non-visible (N) class. This indicates that we need a much larger number of different poses from the robot or a more sophisticated method to alter the few we already have.

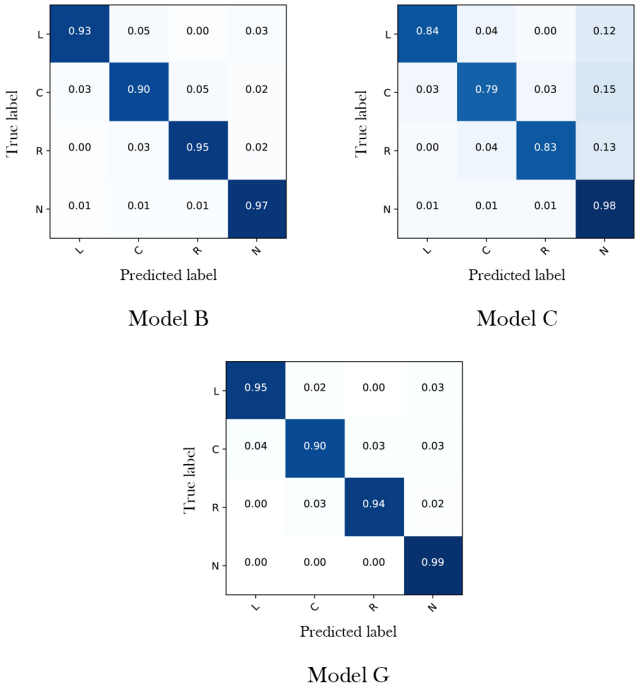


Figure 25: Confusion matrix for models B, C and G. The vertical axis shows the correct class while the percentages at each column show which class was actually predicted by the model. The first row of the matrix that corresponds to model B for example, shows that when the robot was left (L) in the image it was classified correctly 93% of the times. It was wrongly classified as being at the center (C) in 5% of the images and finally for the remaining 3% it was misclassified as non-visible (N). In this case the sum of the percentages is 101% but that's just a rounding error.

3.3 Object Localization

Up to this point it has been described how based on the work of Moeys et al. we were able to train relatively small CNNs to perform loose object detection by segmenting the input image into three parts. This method could be sufficient to create an end-to-end control like the system in figure 2. After the involvement of Matheus Terrivel[33] in the project though, we redefined the specification of the neural network and it was decided to create a more conventional system where the CNN would be part of an object detection pipeline that will localize the GoPiGo robot in greater detail.

In part 3 of the second Chapter, two of the most successful object detectors were briefly discussed. They represent two of the main methods that are based on convolutional neural networks for object detection. On one hand, R-CNN is region based and uses a CNN to classify multiple regions of the image. The main drawback of this approach is the number of CNN evaluations that is needed to draw the final conclusion and this has a serious effect on the computation that is needed and subsequently on the latency of the system. On the other hand, YOLO is much faster and can detect multiple objects with only one evaluation of the CNN. This is possible because the neural network is trained to directly predict the bounding boxes around the objects it detects. A detailed comparison between object detection methods is done in [11].

In our case the most important factors were the time requirements to develop such a system and also its complexity due to the limited resources of the ZYBO board. A sliding-window or region selection based approach like the R-CNN can not be considered due to the aforementioned limitations. Subsequently, the only viable solution was the development of an 'one-shot' detector like YOLO.

In general it is well known that convolutional neural networks have the ability to preserve spatial information throughout the convolution layers [24]. This has been studied, utilized in numerous projects and has led to many state-of-the-art single shot detection algorithms like SSD[19] and Overfeat[30]. The training of those object detection CNNs is not easy though as it requires a training dataset with annotated bounding boxes around the objects to be tracked and expertise on machine learning. In this stage our dataset generation algorithm was proven not very helpful so it was decided to not put more effort and invest time towards altering the generation algorithm to also produce bounding boxes or to develop any new CNN architecture.

Instead we choose a different approach after the observation that the last convolutional layer of the ConvNets we trained give enough information to localize the GoPiGo in a very simple way. This is done by summing all the feature maps of the last convolution layer and then just finding the highest value within the 2D matrix which is the result of the summation. This matrix can be interpreted as a heat map where higher values strongly indicate the presence of the robot.

Figures 27 and 28 show the sum of the last convolution layer of model C (left column) when evaluated with the three images of figures 22, 23, 24 and furthermore on real images taken with a mobile phone. We observe that if we scale the sum of the feature maps and overlay them on top of the input image (center), the high activation areas overlap with the GoPiGo in all of them. There are other areas that also trigger high activations but the highest values always overlap with the robot.

The advantage of our method is that we extract the location of the robot with very simple and quick algorithm. Additionally, we avoid any further development of the CNN or the dataset generation algorithm. We use a simple classifier both for detection and localization. The experiments that were performed show that this localization technique is robust. Furthermore it can be also used for bounding box prediction if we apply a blob detection algorithm in order to specify the whole region of high activations around it. Instead of only extract the coordinates of the highest value.

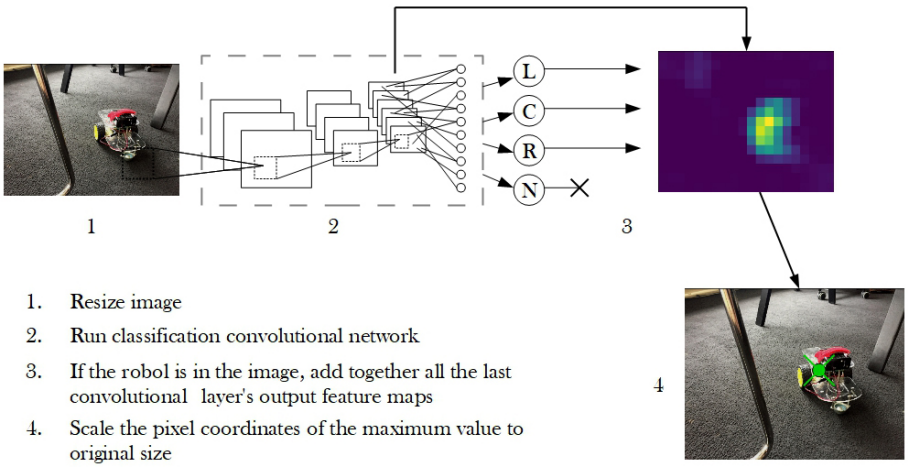


Figure 26: The object detection pipeline.

The detection pipeline can be seen in figure 26. First we evaluate the classification CNN on the input image and only if the result is positive we proceed to summing the feature maps and extracting the pixel coordinates of the highest value. A positive result here means when the classifier predicts that the robot is in the left (L), central (C) or right (R) part of the image. This information can be used to further reduce the computation by summing and searching only the appropriate part of the image. For example, when the classifier indicates that the GoPiGo lays within the left part of image we only need to sum the left half part of the feature maps and then search only there. This results to half the computation compared to the case that we add the feature maps on their full extend.

Being based upon the same classification CNNs we previously discussed, our object detector as a whole still suffers from the large number of false positives. The localization process on the other hand shows good potential in this case of single object detection. In the following part we show how we can greatly improve the classification performance by utilizing a pre-trained CNN for improving the classification. This results to much better performance of the object detection pipeline as we overcome the false positive issues.

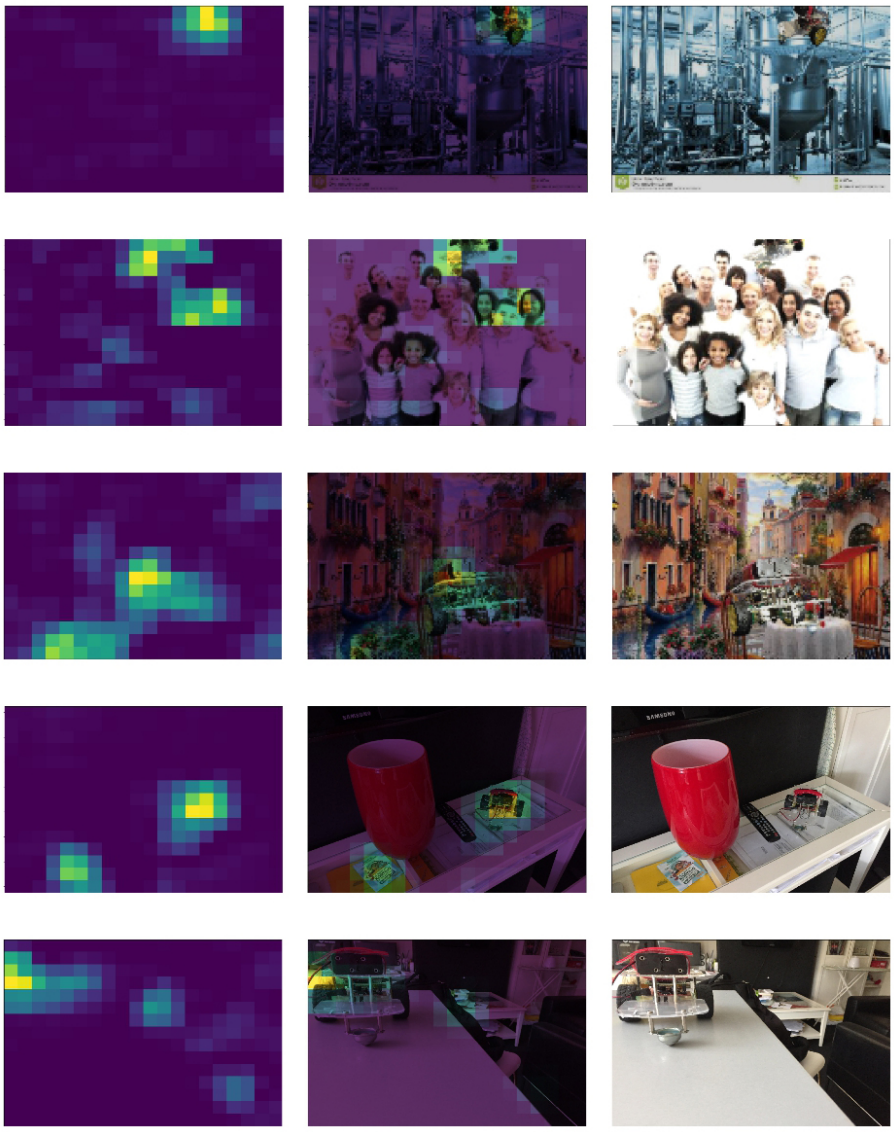


Figure 27: Localization examples. The three images at the top are the same that have been used for figures 22, 23 and 24 while the last two are real photos. Note how the network produces the highest value at the region which overlaps with the GoPiGo. Those images, except the first one, are difficult examples but with our 'highest value' method, the real position could be extracted in all of them.

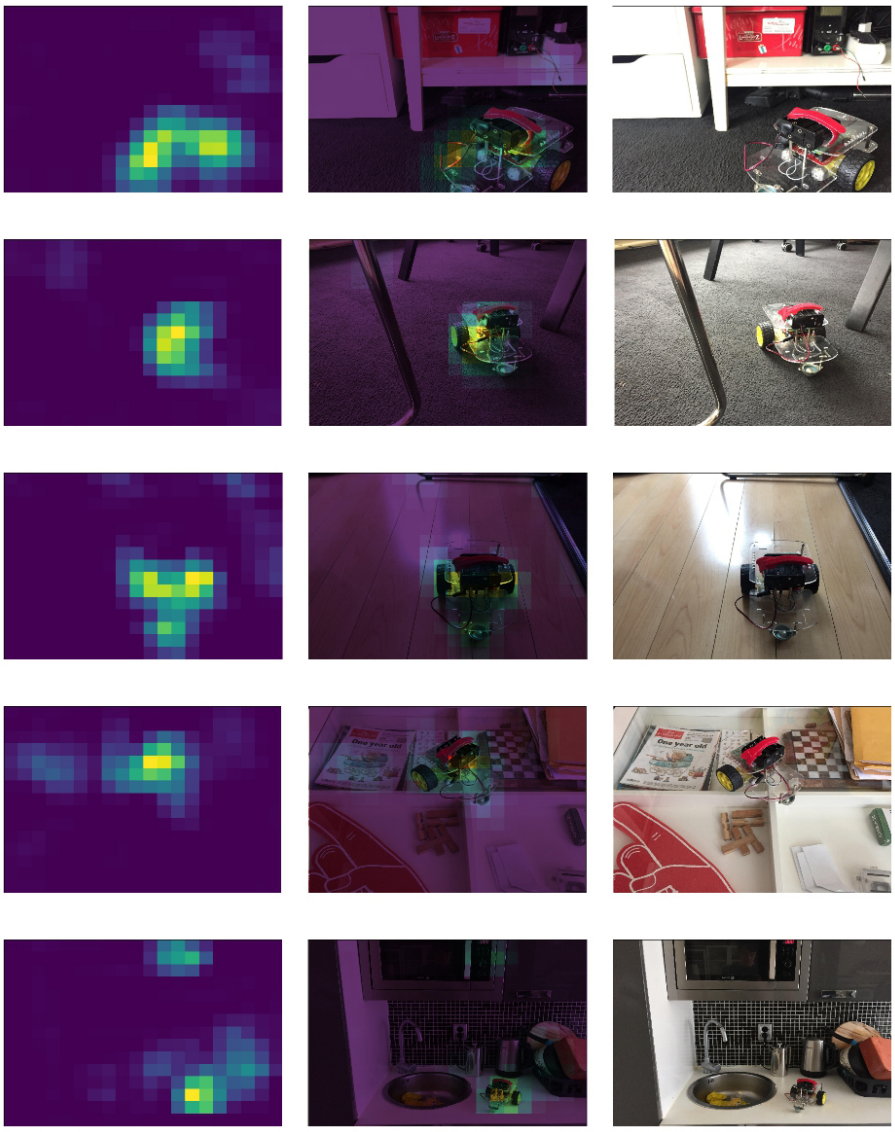


Figure 28: More localization examples. In these examples of real photos its easier to distinguish the robot from the environment.

3.4 Transfer Learning

Within the field of machine learning, transfer learning is a term that describes the use of knowledge that was gained for solving a given task, into another task different from the first one. In our case, knowledge is saved as a network topology together with a set of parameters, namely a model. The task for which the knowledge was developed is the Imagenet classification challenge. Finally, the new task we want to solve is the GoPiGo loose detection with the LCRN classifier we discussed so far.

In fact, what we do is that we replace the handcrafted feature extraction from the traditional image processing approach, with a CNN that has learned parameters. On top of that we add some extra layers, similar to the traditional SVM on top of the feature extractor. The resulting CNN can been seed in Figure 29. Note that the ILSVRC is mainly focused on classification, thus we have to remove the top fully connected layers and only use the convolutional part of the CNN.

When using an already trained CNN it is known that its possible to achieve robust performance with a relatively small number of examples. We can work around the small number of example by augmenting them. The Keras library we use, has build-in functionality for image augmentation. Thus it can natively support various transformations like rotation, zooming, lighting and color alteration etc. The only limitation we face is that our classification problem (LCRN) needs the GoPiGo to be in a certain position. Consequently we can not use most of the build-in functionality because it will change the position of the robot. But, our dataset generating algorithm does exactly the same thing and also provides the correct position so it can be used without any change.

The pre-trained CNN we've chosen to utilize is one of the MobileNets family. The main reason is its small computation and memory requirements, this can be seen also in [11] where MobileNets are used in different object tracking pipelines and compared to other available CNNs. At this stage we rely on the Keras library for support of the pre-trained ConvNets as it is time consuming and challenging to port the architecture and the weights from another framework or library. Thankfully, most of the state-of-the-art CNNs are supported officially or can be found open-sourced from Keras' users.

In order to begin with the training procedure we only need to chose an appropriate value for the α parameter of the MobileNets. It controls the size of the network, so by setting $\alpha = 0.25$ we get the smallest possible configuration of the CNN, whereas $\alpha = 1$ is fully utilizing the network. What the *Alpha* parameter controls, is the number of filters applied at each convolution layer. For example when $\alpha = 0.25$ the output of the network are 256 feature maps, when $\alpha = 1$ the number of feature maps at the output is 1024. Subsequently, the number of parameters is also influenced. From about 300.000 for $\alpha = 0.25$, up to 3.200.000 for $\alpha = 1$. For our classifier we chose the minimum configuration as we only want to track a single object, namely the GoPiGo robot.

The procedure of training the last layers that we've added on top of the mobilenet, takes a bit longer this time but its much more predictable and progressive. We use 38.000 images for training and 2.000 images for validation like we did with the compact CNNs we trained. The time that each epoch requires now is about 10 minutes but we achieve very high accuracy with only 4 epochs. The accuracy on the training dataset is 98% but more important is the fact that

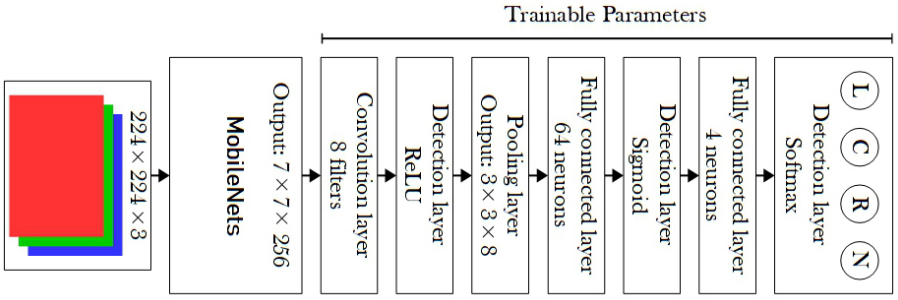


Figure 29: MobileNets-based convolutional neural network. Because the MobileNet serves as a feature extractor, we 'freeze' its parameters and let the optimization algorithm tune only the layers we added on top.

the performance on real images this time is very robust. We test the classifier with streaming video from a webcam and we observe that the lighting conditions or other objects that previously degraded the performance of our custom CNNs, now do not influence the performance of the MobileNets based classifier. Of course, the performance drops in absence of adequate lighting.

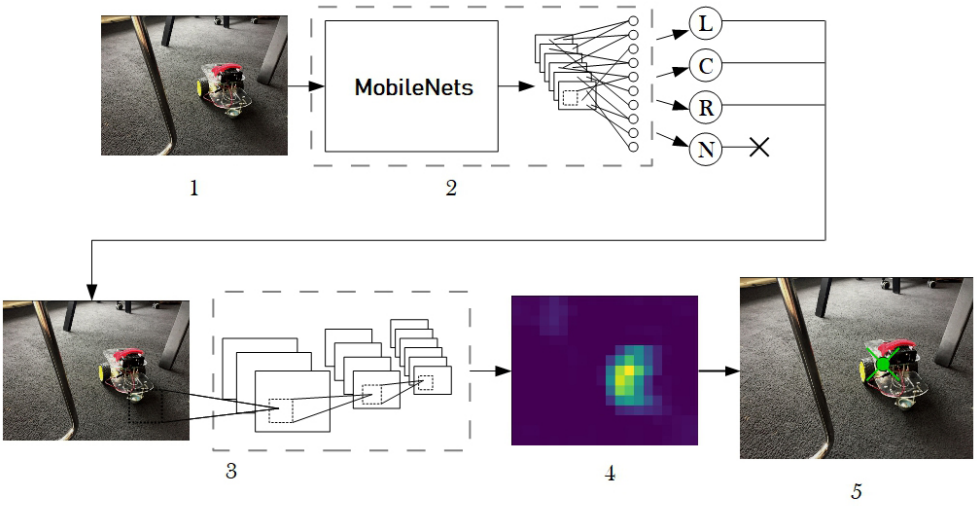
Finally, the last think we've tested is to train the same MobileNets based classifier but with a different configuration from the LCRN we did so far. The decision was based on the observation that the LCRN classifier produces most of the erroneous results when the robot is in between the segments we defined. This is when the robot for example, is not exactly on the left or central part of the image but rather in both. In order to avoid this effect we train a binary classifier with only two outputs that indicate the presence or not of the GoPiGo in the image.

To achieve this, we only alter the training dataset generation slightly and we train the layers that we add on top of the MobileNets which acts as feature extractor. Similarly to the previous configuration, the CNN achieves very high accuracy on the training dataset and this hold also on previously unseen images. The performance of the binary classifier is very robust and we only observe some false positives from a very small number of frames. This problem in the case that the robot is moving, will not be difficult to overcome if we use a Kalman filter for example to filter out those errors that last for a couple of frames.

3.4.1 Using Mobilenets for Robust GoPiGo Detection

The GoPiGo detection pipeline we previously presented, suffers from the poor generalization of the classification neural network. On the other hand the localization method delivers adequate performance even in its most basic form. Having solved the classification problems, with the aid of a pre-trained CNN we can now improve the overall performance of the detection pipeline.

This version of the GoPiGo detector can be seen in figure 30. Now we have to evaluate the MobileNets-based CNN first and only if the prediction is positive we go on with evaluating our compact CNN. Because the classification is based on a different network, it is only needed to perform the computation of the convolutional layers and not compute the fully connected layers of our compact CNN. This is because the fully connected layers produce the output



1. Resize image
2. Run classification MobileNets-based convolutional network
3. If the robol lies within the image, run only the convolutional part of the our compact CNN
4. Add together all the last convolutional layer's output feature maps
5. Scale the pixel coordinates of the maximum value back to the original image size

Figure 30: MobileNets-based object detection pipeline.

of the classifier but are not needed in order to extract the location with our method.

3.5 Evaluation

The Improved object detection algorithm was evaluated in the work of Matheus Terrivel. A GoPro Hero4 which has the ability to stream the video over Wi-Fi, is the camera that was used. The computation of the CNN, the object detection and object tracking algorithms took place on a workstation. Regarding the single-object detection, our algorithm delivered robust performance.

However, the drawback of the implementation was the delay that was introduced by the wireless connection to the camera. The resulting latency was half second and degraded the overall performance of the predator robot. This fact rendered the development of an accelerator important, not only for the computation of the CNN but for the fast acquisition of the video stream as well. The development of the accelerator and the complete system is documented in the following chapter. Furthermore the prey/predator robots are depicted in figure 31 and a sample of the object detection algorithm's output is shown in figure 32.

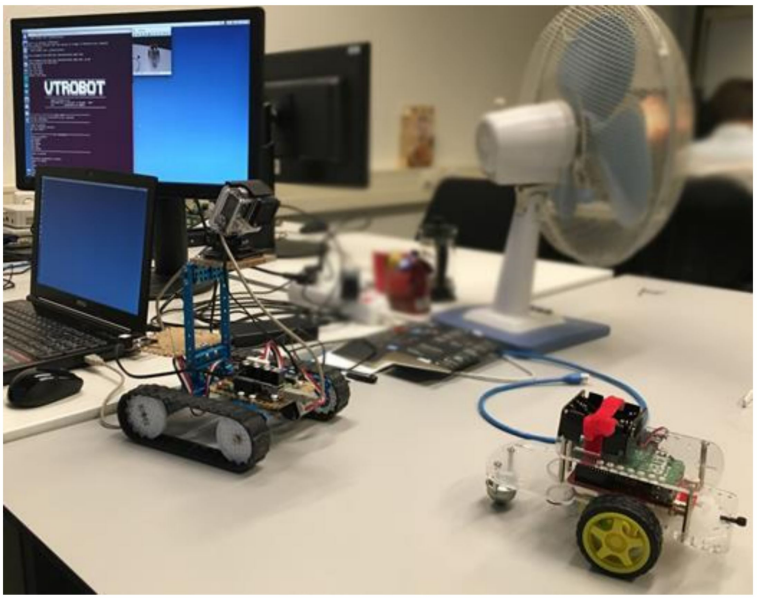


Figure 31: Predator and prey robots together with the object tracking algorithm running at the background [33].

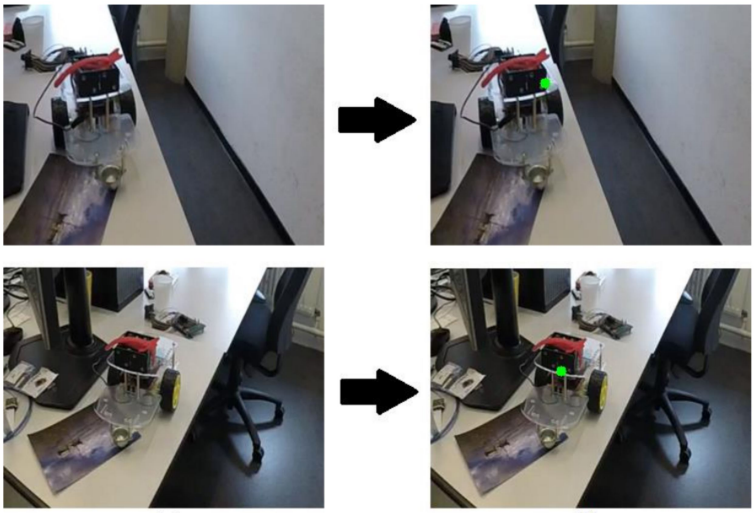


Figure 32: Samples of the CNN-based object detector's output. The input frames are on the left and the output on the right. The actual output of the algorithm are only the pixel coordinates of the green dot, not the whole frame [33].

4 FPGA Accelerator Design and Implementation

This chapter includes information about the accelerator and the object detection system that was build. First we develop software in order to get an insight at the underling computation. Then we can formalize the specifications of the hardware in order to perform in real-time. In this case real-time performance is defined as the ability to process 30 Frames Per Second (FPS). This is because most of the image sensors at the moment deliver at least this frame rate, or even higher depending on the resolution. Moreover, this frame-rate is adequate in order to provide enough information to the robot controller.

Finally, the specifications are translated into hardware and integrated into the Zynq SoC of the ZYBO board. This part also relates with the acquisition of the video stream. But, in this case due to time limitation, we make use of an example project which is provided by Digilent, the ZYBO board's manufacturer.

4.1 The ZYBO FPGA Board

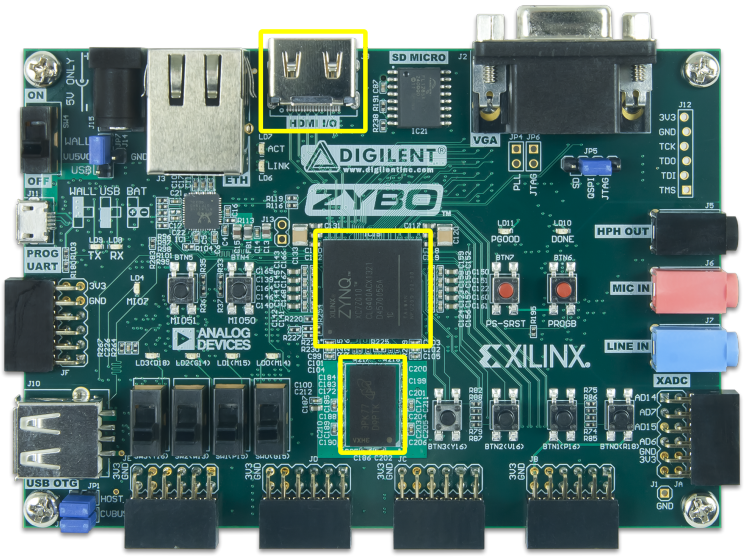


Figure 33: The ZYBO Board. Highlighted are the HDMI connector, the Zynq SoC and one of the DDR3 SDRAM ICs. Those are the components that are utilized by the object detection system.

From the beginning of this project the ZYBO board was chosen to facilitate the object tracking system. The advantages of this choice are the small psychical dimensions, the connectivity and finally the amount of computing power which is packed inside the Zynq Z7010 SoC. Additionally, the board has low power requirements and throughout this project it was only powered by a regular 2,5W USB port.

Another useful characteristic of the board is the support of Embedded Linux operating systems. Although, in this work an operating system was not used and the software runs on 'baremetal'. However, its relatively easy to develop

the drivers which are needed in order to let the system work under a Linux environment in the future.

In figure 33 we highlight the components that were utilized for the object tracking system. Those include the Zynq SoC, the external DDR3 SDRAM ICs and the HDMI port. It is worth noting that while this project was ongoing, Digilent made available a second version of the ZYBO board which includes an image sensor connector and the larger Zynq Z7020 SoC. With its new specification the ZYBO board would be even more suitable for the object tracking system. That is because as we discuss later the small size of the FPGA fabric which lies inside the Z7010 SoC was the limiting factor for our system to achieve better performance.

4.2 Requirements

Up to that point the computation was performed by the Keras library via Tensorflow on a GPU. So, it was decided to develop some software and get an insight on the computation that is required. Using a small library written in C++ gave us the ability to test the performance on the ARM cores of the Zynq SoC and use the timing as a reference. In addition, it was also possible to profile the CNN execution and validate that as in most cases, the convolution layers constitute about 90% of the required computation and time. Based on this fact, it was decided to accelerate only the convolution layers and compute the fully connected on the ARM cores together with the localization algorithm.

At this stage it was also possible to have an estimation of the performance which is needed in order to accelerate the compact CNNs we've trained. Usually, the computation requirement of a ConvNet is described with Multiply Accumulate (MAC) operations. The reason is the high number of MACs that are performed at the convolution layers. As convolution is the most computationally expensive part of CNNs, the number of MACs that are required gives an indication of the size of a network as well. The pooling and activation functions require less computation in any case, so sometimes the number of calculations on those parts is not taken in to account.

Every convolution layer takes a set of feature maps as input and produces another set of feature maps at the output. A set of feature maps is a 3D tensor where 2 dimensions are the width and height of the feature maps and the 3rd dimension is the depth, namely the number of feature maps. Let F_i be the depth of the input set, F_o the depth of the output, W_o and H_o are the width and height of the output and finally, W_k and H_k are the width and height of the kernel. Then we can compute the number of MACs that a convolution layer requires with the following simple equation:

$$\#MACs = F_i \times W_k \times H_k \times W_o \times H_o \times F_o \tag{5}$$

Equations 5 holds under the assumption that stride is one and the input feature maps are padded. The convolution layers of models that we have trained have those properties. Therefore, it is possible to use the equation and calculate the MACs required for model C for example, this is done in table 3. The structure of the model is shown in table 2 and figure 21.

The total amount of MACs that are performed is slightly more than five millions. Thus, for 30FPS we have to perform around 150 million MACs per

Conv Layer	F_i	W_k	H_k	W_o	H_o	F_o	MACs
1	3	3	3	160	120	4	2.073.600
2	4	3	3	80	60	8	1.382.400
3	8	3	3	40	30	16	1.382.400
4	16	3	3	20	15	4	172.800
						Total MACs:	5.011.200

Table 3: Computation of Model C's convolution part

Conv Layer	F_i	W_k	H_k	W_o	H_o	F_o	MACs
1	3	3	3	160	120	8	4.147.200
2	8	3	3	160	120	8	11.059.200
3	8	3	3	80	60	16	5.529.600
4	16	3	3	80	60	16	11.059.200
5	16	3	3	40	30	32	5.529.600
6	32	3	3	20	15	4	345.600
						Total MACs:	37.670.400

Table 4: Computation of Model B's convolution part

second. Model C is a very small CNN though, and the computation of CNNs can increase rapidly by adding more layers, or by applying more filters per layer. This can be seen in table 4 where the same calculations are done for model B. In this case the total amount of MACs is close to 38 millions and to achieve the 30FPS goal, we have to perform slightly more than 1.13 billion MACs.

Note that the first convolution layer operates on the input image itself. But, we can regard the color channels (Red, Green, Blue) of an image as feature maps and do the calculation in the same way by setting $F_i = 3$. Also, note that because our neural networks are simple feed-forward structures, the output feature map set of the first layer is the input to the second layer and so on. Therefore, the following holds: $F_i\left(Layer_n\right) = F_o\left(Layer_{n-1}\right)$

4.3 Hardware Architecture

CNN computation is to a very large degree inherently parallel. In fact, the only data dependency of a feed-forward network is in between its layers. Motamedi et al.[22] define four sources of parallelism. Inter Layer Parallelism refers to the different layers that we cannot compute in parallel for a single input. However, its possible to define a pipeline where every stage is working on a different input. The number of stages then, should be equal to the number of convolution layers. This method will not only introduce complexity to the design but also requires larger area and resources in general. Furthermore, it doesn't guarantee improved performance, so it is not preferred. Its much easier from a system's perspective to exploit the parallelism that lies inside each layer.

Inter Output Parallelism refers to the ability to calculate each feature map independently from the others of a given layer. As a feature map is the result of the convolution between the input feature map set and a filter, it is possible to apply multiple filters on the same input simultaneously.

Inter kernel parallelism lies inside the computation of a single feature map. A feature map is the result of multiple 2D convolutions as shown in figure 8 of the introduction. In that case the input had 3 feature maps and the result was the sum of the 2D convolutions between the 3 feature maps and three kernels. Thus, it is possible to calculate multiple 2D convolutions in parallel.

Finally, the last source of parallelism is Intra Kernel Parallelism which is possible due to the fact that 2D convolution is essentially a number of multiplications and additions. Those multiplications do not depend on each other, so with a pool of multipliers and adders we are able to perform them in parallel.

All the aforementioned sources of parallelism explain why the computation of ConvNets is greatly accelerated by using vector or SIMD processors like GPUs. Naturally, the first candidate for our accelerator would be a vector processor as well. Its main advantage would be the flexibility of accelerating different architectures by simply creating the appropriate instructions with a compiler. That would have allow us to relax the restrictions we defined for our functions and are shown in table 1. For example, with a vector processor it is possible to calculate any kernel size as the computation can be spread across multiple clock cycles.

The disadvantage of such a solution, is that requires the development of both the vector processor and its compiler. Subsequently, due to large amount of time already spent on the development of the object tracking pipeline, it was

1	2	3	3	3	2	1
2	4	6	6	6	4	2
3	6	9	9	9	6	3
3	6	9	9	9	6	3
3	6	9	9	9	6	3
2	4	6	6	6	4	2
1	2	3	3	3	2	1

Figure 34: Pixel reuse rate for a 7×7 input feature map. In case of bigger feature maps, the higher reuse areas become larger. The kernel size is assumed to be 3×3.

decided to implement a streaming processor. The later is relatively easier due to the nature of the Zynq SoC as well.

The main interconnection component of the Zynq is the AMBA bus, which is well connected with the FPGA fabric via master and slave AXI4 ports. This makes it easy to move data between the board's DDR3 SDRAM memory, the CPU and the accelerator. Moreover, the number of available components in the Vivado suite that support the AXI4-Stream protocol helps to further reduce the development time.

Additionally, CNN computation exhibits some characteristics that make it very suitable for streaming processing. First of all, the convolution layers are compute intense. That is because in a case of a 3×3 kernel each value of an input feature map is contributing in 9 pixels of a single output feature map. This number is increased even more if we consider the amount of filters per layer.

For each layer, the number of operations per global memory reference is equal to $9 \times F_o$, where F_o is the number of output feature maps. So, if a convolution layer applies 16 filters and subsequently produces 16 feature maps, then the ratio of operations per global memory reference is $1 : 16 \times 9 = 1 : 144$. Those calculations hold for values that are not at the edges of the input feature map though. The pixels that are at the four corners of the input feature map contribute only once for example. Figure 34 shows how many times a value of a small feature map will be reused for a single 2D convolution.

Furthermore, the reuse of each value happens in a short amount of time. The temporal locality can be exploited by designing a streaming architecture with local buffers. This will result to far less global memory access, which in the case of an external DDR SDRAM IC is also energy inefficient and non-deterministic in terms of access time.

To conclude, the aforementioned characteristics of CNN computation make a streaming hardware architecture very suitable in order to accelerate the convolution layers in an energy and resource efficient manner. We have to sacrifice some flexibility but on the other hand, from a system's perspective the development is easier. In the following parts we show each of the basic components

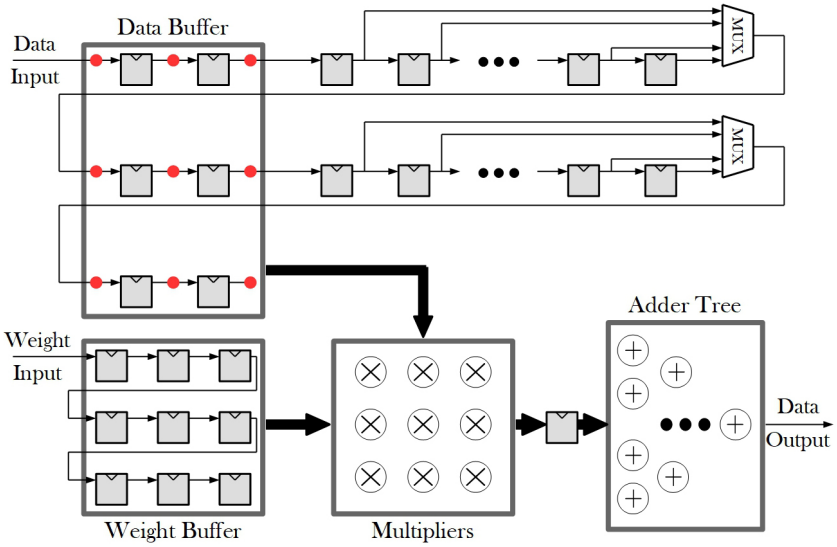


Figure 35: The sliding window architecture (kernel). For clarity, the red dots indicate which signals from the data buffer are connected to the multipliers. The weights on the other hand are all registered, so the multipliers are connected to the corresponding register.

separately and then the complete system, we also evaluate its performance and show some possible improvements

4.3.1 The Kernel

Convolution is the main operation of a CNN. As discussed in Chapter 2, it is already being used in image processing for various other reasons as well. The most common architecture found in literature in order to perform 2D convolution is the sliding window architecture which is depicted in figure 35. With this method, instead of sliding the kernel over the image, the image is streamed through the kernel. By using two row buffers we only need to fetch each value once and then reuse it whenever needed.

There is only one modification that has been done to the sliding window architecture. The reason is that the size of the feature maps varies, thus the row buffer depth has to change accordingly. This is done by introducing a multiplexer that can drive the output of the row buffer with the value which is saved in any of its registers. Subsequently, the delay which is introduced can be adjustable to the length of the input feature maps.

For example, if the length of a feature map is 5 pixels, the length of the row buffer should be two. For a feature map with length of 160 pixels, the row buffer should introduce a delay of 157. In general, the length of the row buffers is equal to $W_i - W_k$. Where W_i is the width of the input feature map and W_k is the width of the kernel. Finally, the number of row buffers also depends on the kernel's height but this is fixed to 3 in this case, thus only two row buffers are needed.

Algorithm 1 is listing a simplified convolution function in C. The code itself

Algorithm 1 The convolution function. This snippet can calculate a convolution layer given a 4D tensor with the filters that are to be applied on the 3D input feature map set.

```
1  tensor3D Convolution(tensor3D in, tensor4D filters)
2  {
3    for(fo=0; fo<features_out; fo++)
4      for(fi=0; fi<features_in; fi++)
5        for(x=0; x<output_width; x++)
6          for(y=0; y<output_height; y++)
7            for(i=0; i<kernel_width; i++)
8              for(j=0; j<kernel_height; j++)
9                out[fo][x][y] +=
10                 in[fi][x+i][y+j]*filters[fo][fi][j][i];
11   return out;
12 }
```

gives an indication of how massively parallel CNN computation can be. By using the structure of figure 35, on every clock cycle we compute the two most inner *for* loops (lines 7 & 8). Those two *for* loops iterate through the kernel's width and height and accumulate 9 weighted values. This fact helps us to define how fast the circuit should be clocked in order to meet the requirements.

With a clock frequency of 100MHz, the theoretical maximum of our accelerator will be 900 million MACs per second. That is satisfactory because it will allow the acceleration of all the models of table 2. For example, the convolutional part of model C would be calculated almost 180 times per second which is far more from the 30FPS goal. For model B though, the number is not so high. The theoretical maximum FPS would be 24 in this case, which is lower from our target. However, because the CNN training phase was not conclusive due to the dataset related problems, this performance was regarded as adequate. Thus, the frequency goal was set to 100MHz.

4.3.2 The Convolver

The sliding window architecture that was described so far is the core element of the accelerator but is not useful without some additional components. First of all, this structure is capable of calculating a 2D convolution each time. As shown in figure 8 though, we sum the result of multiple 2D convolutions in order to compute the 3D convolution. Subsequently, there is need for a temporary memory in order to accumulate the intermediate results.

All the locations of this memory are set to 0 before the calculation of each output feature map. As the input feature maps start being streamed trough the kernel one by one, each output of the kernel is accumulated into the corresponding location in the memory. The final output is ready only when the last feature map is being streamed through the convolver.

At this stage the memory contains the accumulated results of all the previous convolutions. By adding them to the values coming from the kernel we produce the valid output. The output value will then go through the activation circuit. The kernel together with the temporary memory and the ReLU activation are parts of the convolver, which is the component depicted in figure 36. The ReLU activation circuit can be seen in figure 37.

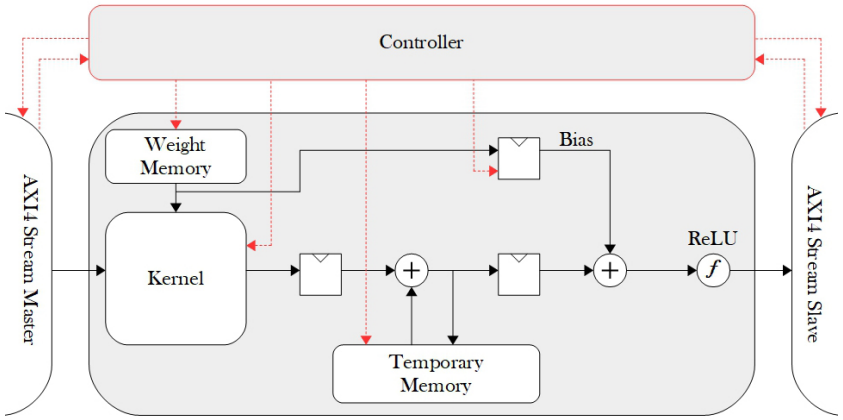


Figure 36: The convolver module. To avoid confusion, the bias which is added to every output was not discussed so far. Bias is a single value per filter and all those values are saved together with the weights in the dedicated BRAM.

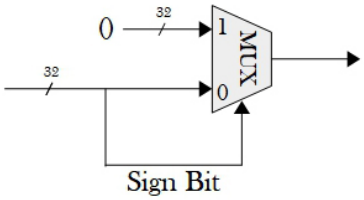


Figure 37: ReLU activation hardware.

Another part of the convolver is the weight storage. In general, the weights of a CNN require a large memory to save them. It can be seen that for our convolver, the weight storage is integrated. This is possible because our compact CNNs have a small number of parameters, usually less than 30 thousands. For that reason and in order to speed up the development, it was decided to use some of the available block RAM that exists inside the Zynq's FPGA. This would not be possible if a big CNN was to be accelerated due to the small amount of available block RAM. To tackle this limitation with bigger CNNs we could utilize some form of external memory. For example, the off-chip DDR3 SDRAM memory of the ZYBO board could be used together with a DMA module to transfer the weights inside the FPGA fabric.

The way that the weights are loaded by using the BRAM is by just increasing a counter's value which in turn addresses the BRAM port. This means that the weights are sorted based on when they are to be used. The ones that are used in the first filter of the first layer are located close to address 0. Subsequently the weights of the last filter of the last layer are located in the highest address of the BRAM that contains a valid value. Due to the kernel size the weights for a 2D convolution are 9. This is convenient as we can load the weights in 9 clock cycles, during the time that the row buffers of the kernel are being filled up.

One more important aspect of the convolver is that its interface is based on the AXI4-Stream protocol. As result of doing so, this design can be easily

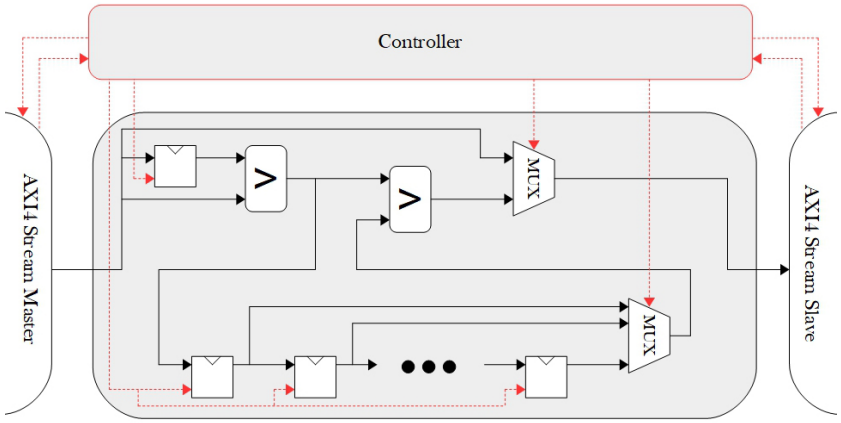


Figure 38: Max pulling module. The components with the greater-than sign, have as output the larger value between their inputs. Thus, they consist of a comparator and a multiplexer.

connected with other modules provided by Xilinx in the Vivado suite. The convolver has a slave and a master AXI4-Stream port. The control signals of those ports are connected to the controller that is indicating when the output is valid and whether the convolver is ready to process incoming data. The controller consists of a Mealy finite-state machine that creates the control signals based upon the two ports status and the instruction that is executed at the moment. These instructions contain information about the number of rows, their width, the number of incoming and outgoing feature maps etc.

4.3.3 Max Pulling Module

Next to the convolver we place the max pulling module. This design is also connected via the AXI4-Stream protocol so it may be used in other applications as well. Its functionality is to perform sub-sampling to the output of the convolver. It does so by forwarding only the maximum value out of a square area of the feature maps that stream through the module. As we previously defined, this area has dimensions of 2×2 pixels.

An important aspect of this module is that it can be deactivated. This means that upon selection, it can be transparent to the rest of the AXI4-Sream chain. Thus, it allows the stream to flow through the module without doing any process or applying any control on the ports. This functionality was added in order to allow convolutions to not be followed by sub-sampling all the time. As data will always stream from the convlolver to the memory through the max-pulling module, there was need for a bypass. A part of the bypass mechanism can be seen in figure 38 which depicts the max pulling module. This is the input which can directly drive the output through the multiplexer which is closer to the control unit.

The control unit of this module is similar to the convolver's. Thus, a Mealy machine is controlling the datapath and the AXI stream ports. The rest of the components that constitute this module are simple. Two comparators, a register and a row buffer with variable length like before, are the only things needed.

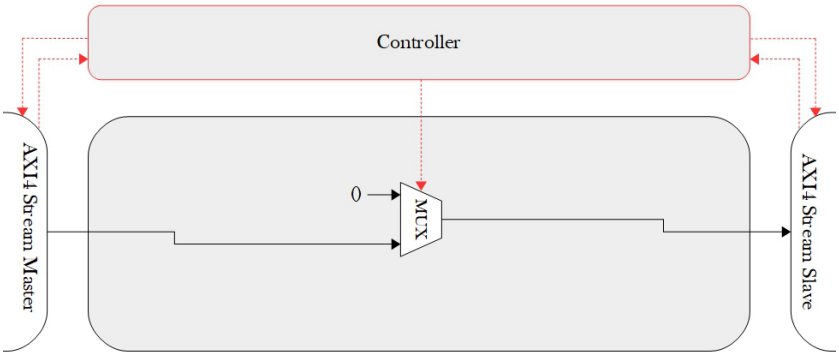


Figure 39: Zero padding module.

Each row of the incoming feature maps is divided into pairs who's max values are saved into the row buffer. This takes two clock cycles. On the fist cycle the pixel that arrives is saved into the register, then on the next clock cycle we compare the incoming pixel with the saved one. At the time that an even row is being streamed, again we compute the maximum value value of a pair. But, this time we also compare the maximum value of the pair with the corresponding max value from the previous row which was saved in the shift register. The highest value among those 3 is then the valid output of the module.

Finally, in case that the width or height of the incoming feature maps is an odd number, then the last column and/or row is discarded. That is done by ignoring the corresponding values when they appear at the input.

4.3.4 Zero Padding Module

Finally, the last part that was designed and implemented is the zero padding module. This is responsible for injecting zeros in the input feature maps of a layer. In contrast to the max pulling module which is after the convolver, the zero padding module is placed in front. A similarity with the max pulling module is that it can be transparent as well. So, if a model does not use zero padding before the convolutions, then it can be disabled.

By using this module we get an advantage though, due to the streaming method we use. When a feature map needs to be zero padded, firstly a full row of zeros should be injected into the stream. That provides an amount of time to the DMA engine to load some values into the local FIFO buffer while the zero padding module is transmitting zeros to the convolver. This amount of time is just a few clock cycles. The exact number of clock cycles depends on the width of the incoming feature map and in our case is maximum 160 pixels, so 160 clock cycles. As a result, the convolver is not waiting for the data to arrive from the DDR SDRAM and can immediately start filling the row buffers with zeros, thus increasing its efficiency.

The structure of this component is very simple. As it can be seen in figure 39, the only piece of hardware in the datapath is a single multiplexer. The controller, depending on its current stage will select to let the stream pass or inject a zeros. The control unit as in the previous modules, is a Mealy state machine.

4.3.5 Accelerator's Overview

After the short discussion on each of the modules that were designed, we can move on and see how everything fits together. At this stage we have the 3 main AXI4-Stream compliant modules, namely the *ZeroPadding* $\rightarrow$ *Convolver* $\rightarrow$ *MaxPulling* chain. The datapaths of the modules are independent from each other but they share part of the controller. Although, every module has its own controller, the central control unit lies in the convolver. As a consequence, the convolver may be used as a single component but the other two cannot function if they don't receive external commands.

The convolver is also functioning based upon a sequence of instructions, one for each layer. Differently from the other modules, it has a small memory controller and is connected to a compact Block Ram memory from where it fetches the instructions one by one. It then distributes the appropriate settings to the zero padding and max pooling modules that depend to it.

We have chosen to save the instructions in a BRAM because its easily accessible from the FPGA fabric. Additionally, a dual-ported BRAM can also be connected to the central AMBA bus via the AXI4 slave ports. This allows the ARM cores to have access to the BRAM as well. In this way we can save the appropriate instruction to the BRAM by parsing the CNNs structure on the CPUs and then send a trigger signal to the convolver to start processing.

Alternatively, we could use a FIFO buffer or some local registers to save the instructions. However, the BRAM usage offers some advantages over the other possibilities. First of all, its control is easy to implement and it can be connected to the AMBA bus by using already available components from the Vivado suite. This was vital to save some development time. Moreover, because the CNN computation has to be performed many times, probably in a perpetual loop, a FIFO could not fit our application as it needs to be refilled every time.

Another form of persistent memory could be a number of local registers which can also be accessed by the ARM cores via the AXI4 protocol. This solution suffers from the fact that the instructions are lengthy and the synthesized registers would occupy resources which are important for a resource limited platform. Those facts render the BRAM solution as a good option because the available BRAM inside the FPGA fabric is enough and we can easily allocate some for the instruction memory. The size of each instruction is 32 bytes, so a CNN with 20 convolution layers requires only 640 bytes. Finally, BRAMs have a fixed access time which simplifies the control logic.

In figure 40, the complete CNN accelerator is illustrated. The only component that is not discussed yet is the connection to the DDR3 SDRAM memory chip. Despite the fact that in the Vivado suite we can directly make use of a DMA module, it has one drawback. This is that the DMA modules need to be instructed from either the ARM cores or a softcore inside the FPGA. In any way that could cost resources and add complexity to the design.

The solution was to only use part of the offered DMA modules. This is possible as in Vivado we can instantiate the basic block of the DMA modules which is called the 'Datamover'. This block offers a simple way to transfer data from memory mapped devices (SDRAM IC) to AXI4-Stream compliant (CNN accelerator) modules and vice-versa. In order to initialize a transaction it only needs a base address and the range that we want to have at the input stream. The same applies for the output stream, we only need to specify the base address

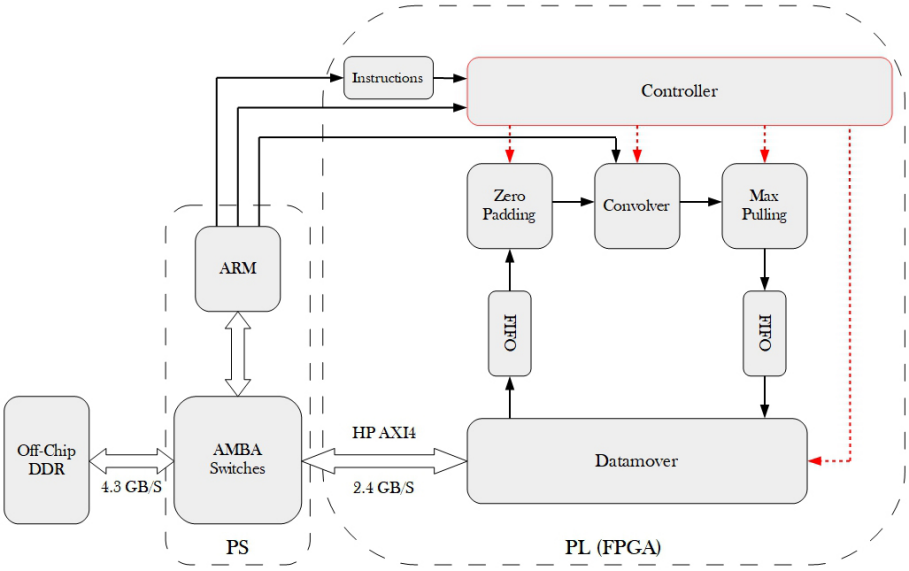


Figure 40: Overall CNN accelerator. PS indicates the predefined hardware of the Zynq SoC whereas PL is the logic which is synthesized inside the FPGA fabric. The DDR3 SDRAM memory is an external IC.

of the place that we want the data to be stored, and also the number of bytes that we are to transmit.

The use of the Datamover module allows us to let the convolver's controller issue the simple commands whenever a data transaction is needed. By so doing, the accelerator can be independent of the ARM cores and calculate the full convolutional part without any stop. That speeds up the computation and also leaves room for the rest of the software to be executed on the CPUs uninterrupted.

In fact, the optimal scenario would be to not use the external memory at all. Unfortunately, this is not possible due to the limited amount of BRAM in the Zynq 7010 SoC, namely 262,5KB[1]. The problem is partially created by the arithmetic that was chosen which is 32bit fixed point. If a convolution layer applies 4 filters and produces 4 feature maps with dimensions of 160×120 pixels, then we need 230,4KB of storage. In reality this is a small number of filters so the storage needs are even higher, thus the usage of only BRAM for all of our storage needs is not possible. Furthermore, the convolver needs its own local temporary memory with a minimum size of 76,8KB ($160 \times 120 \times 4$ bytes) which has to by accessible in a single clock cycle. So, this temporary storage is also provided with the use of BRAM leaving even less for the input and output feature maps.

Generally, ConvNets are known to be very tolerant to quantization, so we could use a smaller fixed point representation. However, the focus of this work was more on the system level development and less on the possible improvements

[1]The available BRAM on other Zynq SoCs is more. For example the Z7020 has 612,5KB and the Z7045 has 2,4MB. For those devices the complete storage of the accelerator could be in BRAMs.

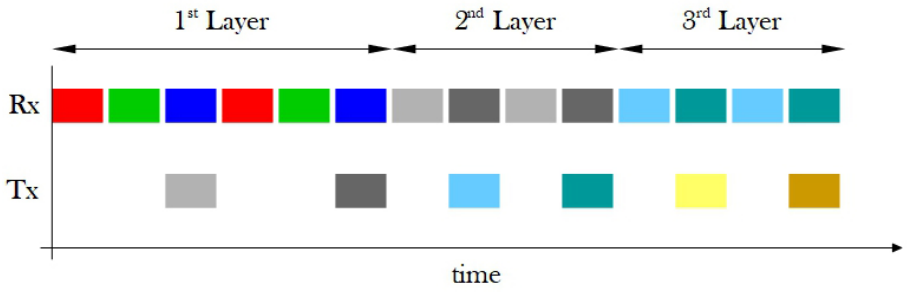


Figure 41: Data traffic between the accelerator and the DDR3 SDRAM memory. Each of the blocks represents a feature map, except the first layer where the input is the color image itself. In contrast to the common way of saving an color image where we have an array of pixels where the RGB values of each one are interleaved (pixel-1-red, pixel-1-green, pixel-1-blue, pixel-2-red, pixel-2-green, pixel-2-blue ...). Here we have to separate the colors and stream them as independent 2D matrices (pixel-1-red, pixel-2-red ...| pixel-1-green, pixel-2-green ...| pixel-1-blue, pixel-2-blue ...) Moreover, the feature maps are colored in order to show the relation between the output of a layer and the input of the next one.

that can make the accelerator even faster. As a matter of fact though, the use of smaller arithmetic like Q8.8 would allow multiple improvements on the design. That is due to the width of the HP AXI port which is 64 bits. With such a word length it is possible to fetch multiple pixel values per clock and subsequently perform more computation.

In its current form, the accelerator is utilizing only half of the HP AXI port's width, namely 32 bits. Consequently, with our 32bit fixed point arithmetic (Q16.16) we fetch one pixel value per clock cycle, if there is one available. In case of having more pixel values per clock, we could compute multiple pixels of the output feature map. This can be done by instantiating multiple multiply-accumulate blocks within the kernel which is depicted in figure 35 . More details on how we can further parallelize the kernel can be found in [29] where a FPGA acceleration template for stencil codes is provided.

After presenting some of the technical details we can have a look into the accelerator's operation. In figure 41 the data exchange between the DDR SDRAM and the accelerator is depicted. In this case an example of processing 3 convolution layers is shown. Each of the 3 layers applies 2 filters on the input feature map set and subsequently produces 2 feature maps. The input to the first layer is a color image, so we use red, green and blue to indicate the 3 color channels.

In order to produce a single output feature map, we have to stream the whole input feature map set. This is seen in figure 41 where each of the outgoing bursts is happening during the last incoming feature map. Therefore, for the first layer we stream the 3 color channels twice and we produce the two output feature maps. Those feature maps are then streamed twice back to the convolver in order to produce the next two feature maps and so on. The final output of this model's convolutional part are the two feature maps that are transmitted during the computation of the third layer. They are the yellow and brown block which are not streamed back to the accelerator. They will only be processed by

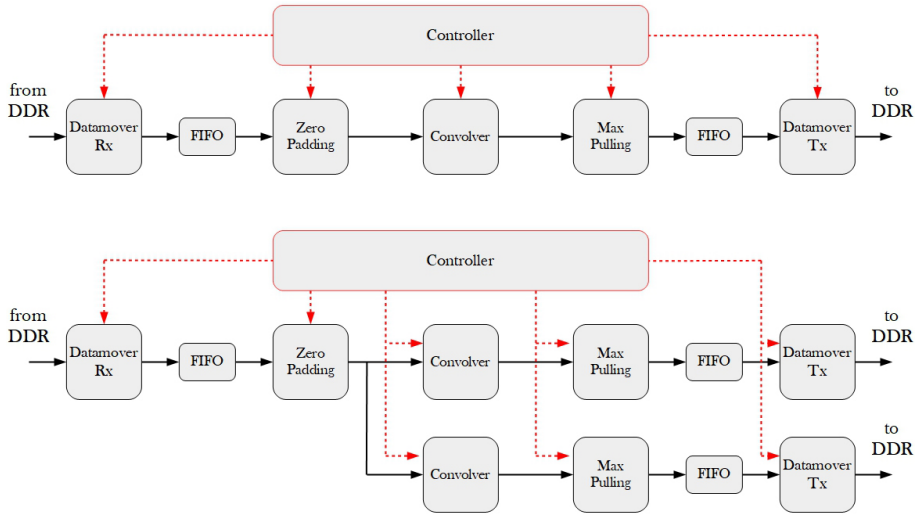


Figure 42: Single(top) and multi(bottom) processing element implementation of the accelerator. Given enough available resources, it is possible to greatly improve performance by instantiating multiple convolvers inside the accelerator.

the ARM cores in order to compute the fully connected layers and produce the final output of the CNN.

The need of re-streaming the input can be explained by algorithm 1. There it can be seen that in order to produce an output feature map, we have to apply the 3D filter on all the input feature maps. Due to our streaming processing architecture this means that the input has to be streamed as many times as the number of output feature maps. This is a pitfall of our architecture because it increases the traffic on the central bus, thus it raises the need for high memory bandwidth. Possible solution to this, is the use of a tiling mechanism that would store locally part of the input every time. Another one is to have available enough BRAM so as to store the whole input and output feature maps.

One more way to reduce but not eliminate the need of re-streaming the input, is to calculate multiple filters every time. This is possible if multiple instances of the convolver are synthesized in the FPGA. By doing so, we exploit the inter-output parallelism and improve the overall performance of the accelerator. On the other hand, this requires more FPGA resources and at least in the case of the Zynq Z7010, we show later that this isn't possible.

Figure 42 shows how a multi-core implementation would look like in contrast to the single convolver scenario of this work. It can be seen that the input circuitry remains the same, namely we would only need one DMA module for streaming the input and one zero padding module to inject zeros into the stream. Differently, for each of the convolvers we would need a max pulling and a DMA module to handle its output. Finally, figure 43 shows how the timing diagram of figure 41 will change in case of a system with two convolvers.

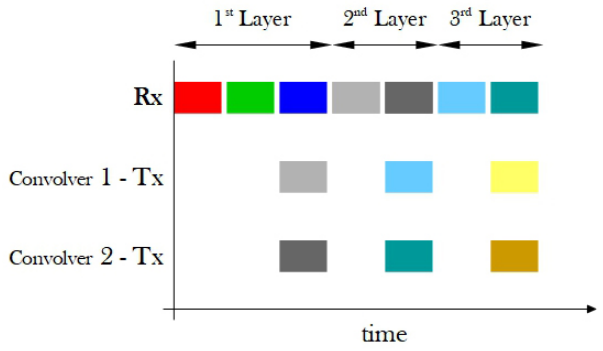


Figure 43: Data traffic between a multi-convolver accelerator and the DDR SDRAM memory. Based on the example of figure 41. In this case, because the accelerator includes 2 convolvers we only need to stream the input once. Each of the convolvers is computing a different feature map so in contrast to figure 41 the complete computation takes half of the time that a single convolver would require.

4.4 Loading a Keras Model into the Accelerator

As mentioned earlier, the computation of the CNNs is shared among the accelerator and the ARM cores. But still, we need a way to extract the structure and the weights of a model and load them into our system. We have chosen to transfer the model to the ZYBO board in two steps. This was done because the instructions of the accelerator are very long and complex. So, they cannot be easily read and modified by hand. The intention of this work was to also make the accelerator as flexible as possible.

Thus, the first step is to encode the Keras model from Python object into two simple text files, one for the structure and one for the weights. We then let the software on the ZYBO board to parse one and create the complex instructions of the accelerator. Thereafter it copies the weights from the other into the appropriate storage. This will allow the user to manually write a description and even set the weights if needed.

A model in Keras is an object with multiple variables. The majority though, are not useful for us in order to accelerate only the inference. The way that we parse a Keras model is focused around convolution. This is due to the structure of the accelerator. It was previously discussed how its components are placed one after the other. Although we can activate or deactivate the max pooling and zero padding modules, the convolution on the other hand, will always take place and will always be followed by ReLU activation.

Subsequently, when our algorithm parses a Keras model it focuses on finding only the convolution layers. Then it checks if their input is zero padded and whether they are followed by max pooling or not. This information is encoded in a simple way by just producing a text file that contains information about the convolution layers. Each row of the text file stores all the setting that are needed to describe a convolution layer. Those settings include the width and height of the input feature maps, the number of input and output feature maps and finally, an ON/OFF indication for zero padding and max pulling.

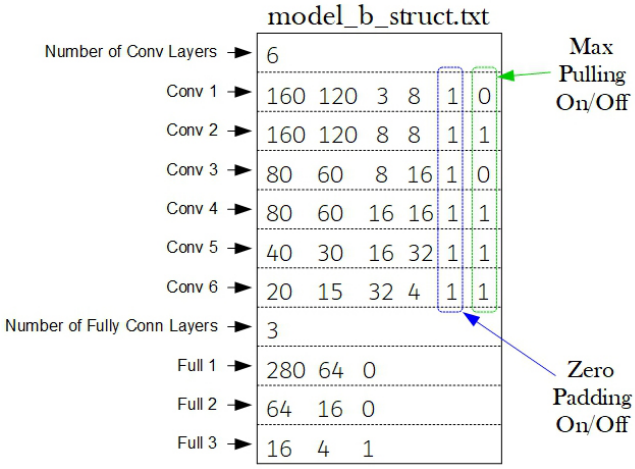


Figure 44: Structure description text file for model B.

A sample of a structure description text file can be seen in figure 44. The file contains also information about the fully connected layers as well. This information is used by the software which runs on the ARM cores. It basically instructs the software on how compute the rest of the neural network which follows after the convolutional part. In this case the settings include the number of input and output neurons together with a number that indicates which activation is to be applied at each layer's output.

The second step in order to set the accelerator, is to handle the text files at the ZYBO board. The structure description is combined with information about the feature map buffers which are located in the DDR3 SDRAM memory and the amount of bytes that are exchanged. This complexity arises from the fact that the convolver computes a full pass of the convolutional part autonomously. Thus the instructions should combine the settings that will enable the controller to control the data stream, together with the commands that the Datamover module needs.

During the initialization of the convolver the program allocates the feature map buffers in the SDRAM memory. In order to save space, we use a ping-pong configuration for the buffers. This is when we use two buffers that serve a different role during each layer's computation. In case that for a given layer, buffer A holds the input feature maps and buffer B is used to store the output, then for the next layer, B will be the input and A the output buffer. In this way we can save space by using only two buffers instead of one buffer per convolution layer. This configuration is possible due to the sequential nature of our simple CNNs. As one layer uses the output of only the previous one, we can overwrite data that will not be further used.

The memory allocation is followed by the compilation of the instructions. The CPU now can compile the instructions together with the known base addresses and the offsets. When ready, the program will write them into the Instructions BRAM which is connected also to the accelerator's controller. The same will happen for the weights as well. They don't require any further process though, so they can be directly copied from the text file to the weight memory

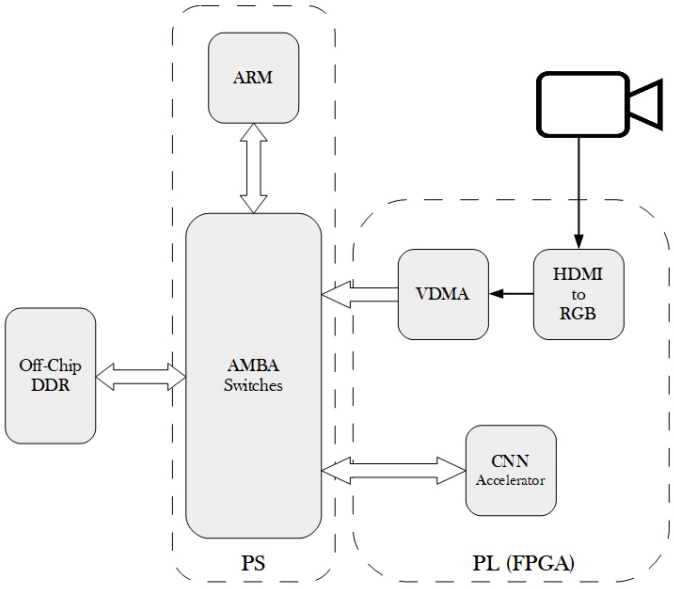


Figure 45: The complete object tracking system. Similarly to figure 40, Ps includes the predefined hardware of the Zynq SoC and PL indicates its FPGA fabric.

that is depicted in figure 36 . Finally, we set the number of convolution layers directly in a controller's register via an AXI4 lite interface. This is needed in order to make the controller aware of the number of instructions it has to execute.

4.5 Overall System Structure

In the previous part some information about the initialization of the accelerator was provided. In fact, this is more a part of our compiler because during initialization the instructions are formed. Therefore, in this part we discuss how everything fits together in order to perform the CNN based object tracking.

The aspect of the system which has not been discussed yet is how the images are acquired. This is done via a GoPro Hero4 action camera which is connected to the HDMI input of the board. The signals of this connector are available inside the FPGA fabric of the Zynq SoC, so there is need for some special hardware to get the frames that are coming from the camera. The image acquisition hardware consists of a frame decoder and a VDMA module. The frame decoder is responsible for generating pure RGB values for every pixel of the image. Those RGB values are then saved into the external SDRAM memory via the VDMA module which is instructed from the ARM core during the initialization. This system was developed by Digilent which is the manufacturer of the ZYBO board, thus we do not provide any further detail on this part of our work.

A system's overview is depicted in figure 45 . The Zynq SoC gives us the advantage of parallelizing the image acquisition and the CNN acceleration. This

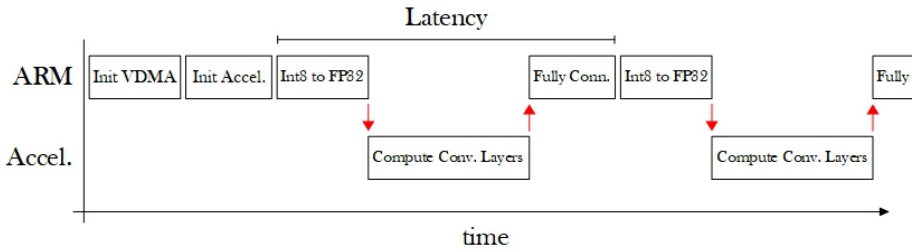


Figure 46: Schedule of the object detection system.

is due to the fact that the camera's VDMA and the developed accelerator are using their own high performance AXI4 port and are totally independent from each other. The only downside as will see later, is that the hardware to acquire the frames is utilizing a considerable amount of resources in a small SoC like the one we are using.

An issue that arises from the arithmetic representation we use is that the 8 bit integer values that the camera provides, have to be transformed to the 32 bit fixed point values that the accelerator is designed for. Furthermore, the integer values for each color intensity are within the 0 - 255 range. But, the ConvNets we trained work with color intensities within the 0 - 1 range. To tackle this discrepancy in a fast way, the RGB values are just shifted 8 positions to the left. This introduces a small error but does not require any computation at all. Speed was of greater importance for this conversion as it has to be performed for every frame that the accelerator will process, thus this amount of time is added to the total latency of the system.

Figure 46 provides insight into the scheduling of the system. The first step is to initialize the VDMA so as images start to stream from the camera into the SDRAM memory. After its initialization this process is totally autonomous and does not require any intervention from the CPUs. We just grab the frames from the DDR memory whenever we need to process one.

Subsequently, after the initialization of the VDMA and the accelerator, the CPU will immediately be able to convert a frame's values from integer to floating points and trigger the accelerator. The accelerator and the image acquisition hardware use different buffers inside the same DDR. Thus, the CPU is not only changing the range and arithmetic but also copies the image into the accelerator's input buffer. After this conversion the accelerator is ready to process the frame.

By just setting a register of the accelerator, the ARM cores can trigger the execution of the CNN's convolution part. This is indicated in figure 46 by the red arrow that points downwards. When the computation is done it is time for the accelerator to trigger the CPU to grab another frame, compute the remaining fully connected layers and also extract the location of the GoPiGo robot. This is implemented by using the Generic Interrupt Controller (GIC) of the Zynq SoC.

By programming a simple interrupt routine we let the system run as fast as possible. The interrupt routine is responsible for looping over the same procedure. This is to compute the fully connected layers when the accelerator is done, then load the next frame into the input buffer and trigger the accelerator

to start. In the meanwhile the CPU can run other algorithms.

In general, the software that was developed is tailored for our specific application. At its current state the accelerator can be used only for simple feed-forward networks similar to those we have trained. However, it is only a mater of developing the appropriate drivers and software in order to allow the acceleration of more complex neural networks. Even complex networks like the ones shown in the introduction. That's because the convolver which includes the sliding window kernel, can be utilized in any application that includes convolution as part of the process.

The only real limitation of this accelerator is the fixed kernel size. The problem comes not only with networks that contain larger kernels which cannot be accelerated at all. But, we also face a problem with smaller kernels. For example the MobileNets CNN family makes use of 1×1 kernels. This is possible to be accelerated, however it will reduce the efficiency as we would need to set the weights of 8 multipliers to 0 and use only the central. Subsequently the majority of the multipliers will be inactive, reducing in this way the performance maximum to only 100 million MACs.

4.6 Evaluation and Results

Unfortunately, the object tracking system is not working properly. It was already known that the images which are produced by the camera we used, have slightly different colors from the ones that the training dataset has. Furthermore, we were able to re-train our networks and overcome this problem when the object tracking algorithm was evaluated in the work of Matheus Therivel. However, now we face also a discrepancy due to our resizing method. That is due to the fact that the images we obtain from the camera are larger than the CNN's input, so they have to be resized. The resizing method then, has to be similar with the one that which is used on the workstation. This takes time and moreover in an embedded system we get a time penalty for using more sophisticated methods.

All the aforementioned issues strongly suggest that the correct order of deploying CNN-based object tracking is not the one we followed. During this project, the ConvNets were trained based on a generated dataset and this is a problem when it comes to porting those networks in a system with different image characteristics. The proper way to train the CNNs would be to directly acquire the training imagery from the setup with the exact components that are to be used after deployment.

In any case, the algorithm and hardware that were developed are suitable for the task at hand. As described in the following parts of this chapter, we were able to deliver enough performance with a relatively simple system that is both energy efficient, scalable and flexible. But, because the object detector cannot operate correctly on live stream from the camera, it was necessary to test the system with static images from our training dataset.

This test was mainly needed in order to validate that the quantization due to the fixed point arithmetic is not introducing a large error. Indeed, this arithmetic representation provides enough detail to let the system evaluate the CNN without any noticeable difference from the corresponding implementation in Python. This fact reassures us that the only problem is the difference between the training dataset and the images that the setup's camera produces.

Site Type	Used	Available	Utilization %
Slice	4399	4000	99,98
LUT as Logic	12318	17600	69,99
LUT as Memory	2023	6000	33,72
Block RAM Tile	41	60	68,33
DSP48E1	36	80	45

Table 5: Total Resource Utilization.

Component	LUT Logic	LUT Memory	BRAM Tiles	DSP48E1
AXI4 interconn	5395	1024	0	0
VDMA	2064	168	5,5	0
Datamover	876	153	1,5	0
Convolver	730	512	33	36
Max Pulling	158	128	0	0
Zero Padding	70	0	0	0

Table 6: Resource Utilization of the Major Components. The AXI4 interconnection figures are the sum of all the AXI4 interconnections that are instantiated inside the FPGA. Those include an interconnection for the AXI4 Lite slave port and 2 AXI4 interconnections for the high performance master ports.

4.6.1 Resource Utilization

The ZYBO board's version that was used utilizes one of the smallest Zynq SoCs. Therefore the hardware was designed with small resource utilization being of equal importance with its performance. The restrictions that we applied on the functions that constitute our CNNs helped a lot towards this direction. That can be seen in table 6 were the resource utilization of each component is listed separately. The FPGA fabric inside the Z7010 SoC is so small that the majority of resources is used for data transfer, namely the VDMA module, the Datamover and the AXI4 interconnections.

Subsequently, the accelerator barely fits in the FPGA even with only a single convolver. This resulted to under-utilization of the available DSP modules which are available and could allow us to fit at least one more convolver. The only possibility to make more resources available in the current SoC, is to design dedicated memory controllers instead of using the DMA modules that Xilinx provides. That is because those modules are designed to fit many usage scenarios, thus introduce a large overhead.

Finally, the figures in table 5 and 6 would be greatly improved in case of a smaller arithmetic representation. As discussed earlier, this would have a significant positive effect not only on the memory requirements but in resource utilization and on the combiatorial path lengths as well.

4.6.2 Performance

Based on the calculations in the second part of this chapter the frequency goal was set to 100Mhz. In order to achieve this clock rate, the accelerator was designed as a four-stage pipeline. The three registers that are placed between the input and output are all located inside the convolver and can be seen in figures 35 and 36. Consequently, the zero padding and max pulling modules are purely combinatorial circuits.

On the first clock cycle, zero padding and the parallel multiplications take place. The second cycle includes only the adder tree and the third cycle accumulates the resulting value to the local temporary memory. Finally, on the last clock cycle the activation and max pulling is performed. With this arrangement the four paths have a similar time delay with the critical path being the adder tree, with only a small difference from the rest.

Theoretically, by achieving the 100Mhz target, the accelerator would be capable of performing 900 million MACs per second. In reality though, this performance cannot be reached as it requires an efficiency of 100%. This efficiency figure is simply the percentage of clock cycles that we have a valid output. The first reason that results to wasted clock cycles is the communication to and from the external DDR3 memory. This communication is not deterministic in term of time access and from the moment that a data transaction is requested it takes some time for the values to appear at the input of the accelerator.

The second reason for the wasted clock cycles, is that the sliding window hardware as depicted in figure 35 requires a number of clock cycles to fill up its row buffers. This number of clock is proportional to the width and height of the incoming feature maps. Although the absolute number is higher for larger input, the ratio between the wasted clock cycles and the ones that we produce a valid output is better. For example if the input feature map has a size of 5×5 and we get 25 values at the input, then the output will be 9 valid values, thus the efficiency is only 36%. However, for incoming feature maps with a size of 160×120 the efficiency is 97%.

By testing the accelerator with a CNN that requires 25.663.400 MACs, it was found that its convolution part is calculated in 37 milliseconds. Thus, in this case the real performance is close to 700 million MACs per second which indicates an efficiency of 78%. However, this number holds only for this example. In general, different models would have different performance depending on their topology. It is possible though to predict the execution time of a given model. This rough estimation is possible if we assume that the connection to the memory is never delayed and then just find the sum of the number of values that is expected to be streamed at the input channel for all the layers.

Finally, the total latency is the sum of the resizing and arithmetic conversion of the input frame, the calculation of the convolutional part on the accelerator and the computation of the fully connected layers on the ARM CPU (figure 46). This is 4, 37 and 0.16 milliseconds respectively, thus the total latency for the model that was tested is 41.16 ms on average. Subsequently the throughput is about 24 FPS which is an adequate performance for a control loop.

The most important figure in this case is the latency. As it was shown in the work of Matheus Therivel, it is crucial to have the information as fast as possible at the controller. With even this basic configuration of the accelerator we are able to bring latency from half a second, down to the 40 milliseconds

range.

Furthermore, it is also possible to directly compare the performance of a single ARM CPU to our accelerator. As mentioned, the fully connected part of the CNN is computed in 160 microseconds. However, the number of multiply-accumulates in this case is only 11360. This means that in a microsecond a single ARM core performs 71 MACs. So, it theoretically needs 361,5 ms to perform the 25,6 million MACs of the convolutional part. That is a $\times 9$ better performance for the accelerator, compared to very fast software implementation with maximum optimization settings and without an operating system. The only point for the CPU is that the calculations are done with 32bit floats instead of fixed point.

The library that was developed in an object oriented manner with C++ and evaluated on the ZYBO with Linux, was even slower. In that case the latency for a similar model was more than 7 seconds. So we can conclude that even a single-convolver and unoptimized accelerator, is quicker than any software we tested on the 650MHz ARM Cortex-A9 CPUs.

5 Discussion, Conclusion and Future Work

The research question that was defined in the introduction can be divided into two parts. The first one referring to the possibility of training a CNN in order to use it into an single-object-detection application. Whereas the second part argues the feasibility of a CNN accelerator that will be able to accelerate the trained ConvNets and also meet the real-time constrain on the ZYBO board.

We also defined a number of objectives for the purpose of answering the research question. The objectives that are related to the first part of the research question are objectives number 1, 2 and 3 as listed in part 2 of the introduction. The first objective was to create a suitable training dataset which is a time consuming task. Instead, we propose a method to create artificial training datasets for object-detection that uses only a limited number of original images of the object. Those images are combined with random backgrounds and adjustments of the lighting and color properties. The reason is to create a large number of training material that has a plenty of possible poses, sizes and locations of the object to be detected.

The second objective of the first part was the study and evaluation of existing CNN architectures. This objective was partially fulfilled by studying the work of Moeys et al. as they demonstrate a suitable architecture for a very similar task. We formalize the localization problem in a rough way by following their example. Thus, the output of the CNN is a 4 value vector that shows the probability of the object being at the left, center and right part of the image or not present at all. This classification method, was not able to deliver robust performance as it was influenced by different lighting conditions, camera specific properties and similar objects. This behavior is attributed to the training dataset, so it may be that by compiling and using manually annotated training material, the architecture will deliver sufficient performance.

A possible answer to what causes the CNN to perform well on the generated images but not in general, are the patterns that are introduced by the generation algorithm and that can be exploited by the CNN. Another reason was proven to be the small number of images that include the robot. By using a CNN trained on the artificial training dataset we can have good performance only under a constraint environment. For example, if the robots were only used inside a specific room it is possible to generate a training dataset that will cover all the possible conditions that may occur.

After the failure to deliver good overall performance with a compact CNN, the artificial training dataset was used to train a bigger one. This larger CNN uses as a feature extractor another ConvNet which is trained with the Imagenet dataset. This decision was motivated from the fact that neural networks which are trained on huge datasets contain very useful filters that help them to predict between thousands of object classes. By using the feature maps generated from those filters we can easily achieve robust performance in various tasks, different from the original classification challenge. Of course, this comes with a penalty on the amount of computation and memory demands as well. This is almost $\times 3$ more computation and $\times 10$ more parameters compared to the compact CNNs. The pre-trained ConvNet that served as a feature extractor for the large CNN is a variant of the MobileNets. This family of ConvNets was chosen because it is specifically developed for energy and resource limited devices.

The resulting CNN is performing very well and can correctly detect the

robot in any environment; given that there is adequate lighting. Furthermore, a similar CNN was trained as a binary classifier that predicts only if the robot is in the image or not. Again the performance is excellent and there is only a small number of frames that the CNN's prediction is a false positive. Actually, the CNN is able to detect the object even in poses that it has not previously "seen".

The later exposes the problems of our small CNN training procedure. The GoPiGo robot has a distinctive shape and color combination. So, although we trained the network with only a few original poses, the expectation was that generalization could be easier. Instead, our CNNs tend to recognize only the few poses that they were exposed to during training. MobileNets on the other hand, will produce very similar responses on every pose. The layers we add on top can then successfully exploit the patterns on the output feature maps of the MobileNet and offer robust performance.

The third objective was to integrate the trained CNN into an object detection system. Usually, a CNN trained for classification has to be used multiple times over the same input in order to find the location of an object by trying different areas and scales every time. Undoubtedly, this method is not optimal because we have to repeat the computation of the CNN and that is a very heavy workload even for powerful workstations, let alone embedded devices. More recently, CNNs are trained to exploit the spacial information which is preserved throughout the convolutional layers and directly predict the bounding boxes around the detected objects with only one pass of the image through the CNN.

In this project, we base localization on the same property. The very last convolution layers of the compact network we trained convey enough information in order to extract the pixel coordinates of the object. Our method delivers a rough estimation of the coordinates by only taking into account the position of the highest value in the last convolutional layer's output. The reason that makes this possible is that a neural network trained to classify the existence or not of a single object and loosely localize it, will naturally convey information about its location as well.

This method can be used only with the compact CNN though. the reason is that in the MobileNets-based CNN, the spatial information is mostly lost throughout the many convolution layers. Additionally, the output we get from MobileNets has width and height of only 7 pixels. This resolution is very small in order to perform localization after so many convolutions. So, to deliver robust object detection we combine the large CNN with a smaller one. The large MobileNets-based CNN performs the classification and only if the result indicates the presence of the GoPiGo, then we also evaluate the compact CNN which can help us to localize the robot.

The resulting system has the drawback of not predicting the size of the object. It has the potential though to let us predict bounding boxes if a more sophisticated algorithm for blob detection is used on the output of the last convolutional layer. To conclude, we may use a CNN trained as a classifier to also extract the location of the object. The method used, has very low computational complexity and is easy to implement on any platform. In our case though, the training dataset wasn't very useful so we utilized a pre-trained neural network to achieve robust classification.

After the discussion on the 3 aforementioned objectives that relate to the development of an object detection system that is based on a CNN. We now can move to the objectives that constitute the integration of the detection system into the ZYBO board and its acceleration in order to meet the real-time constrain. It is to be noted that some decisions that relate to the development of the hardware had to be taken early while working on the algorithmic part so the objectives are not listed in strict chronological order.

The fourth objective for example, includes decisions that had to be made early. The objective itself was to define a subset of functions that can be accelerated easier than the rest and also have a small footprint on the FPGAs resources. Due to the abundance of architectures, functions and variations of each function, it was crucial to limit the number of functions and their variations in order to be able to develop a lightweight accelerator. This is in contrast to the accelerators that can be found in literature but because of limited time and experience it was impossible to design a fully flexible and robust architecture. The decision was to only support convolutions that use kernels of 3×3, stride of 1 pixel and ReLU as the activation function. The max pulling operation is restricted to an area of 2×2 pixels with stride of 2 pixels and finally it was decided support zero padding on the input feature maps of the convolutions.

Those restrictions do not have any significant effect on the training of the compact CNN and the chosen function variations are commonly used in general, for example the VGG architecture uses exactly the same operations. The only restriction that is potentially affecting the performance of the CNN is the fixed kernel size, this is because especially on the first layers of a CNN it can be beneficial to use kernels larger than 3×3. It is possible though to overcome this issue by stacking more than one convolutions before sub-sampling, thus increasing the receptive field of the last convolutional layer before the pooling operation.

The fifth objective was the development of software that performs the computation of the trained CNN and then the selection of the most computationally expensive part for replacement with an accelerator. Although we were previously able to develop the object detection algorithm in Python by utilizing the Keras library, there was no easy way to use the same scripts on the ZYBO board. Additionally, Python is not known for being fast. So in general, it wasn't the optimal solution for implementing a neural network with real-time constraint. To overcome this problem a program was written in C++ and was used on the ZYBO board that was running Linux. The profiling of this program showed that 90% of the time was spent on the convolutional layers of the CNN. Therefore, it was chosen to develop only a convolver that also performs sub-sampling in order to accelerate the execution of the time consuming convolutional layers. The computation of the fully connected layers was chosen to be performed on the ARM cores.

The last objective was the development of the accelerator itself and its integration into the Zynq SoC. In order to accelerate the convolution we develop a streaming architecture that performs 3D convolutions by calculating the 2D convolution of every feature map with the appropriate kernel and then accumulates the result into a temporal memory. The final result is written back to the main memory during the last feature map's convolution.

We place the convolver which acts as the main block between the zero padding and the max pooling modules and we connect the incoming and outgoing data streams to the SDRAM memory of the board by using the high performance AXI4 port. The acceleration of the CNN's convolutional part is a result of calculating the activations, the max pooling and the convolutions in parallel. The convolution itself is accelerated by a processing element with 9 multipliers and 8 adders. The accelerator is clocked at 100MHz, its arithmetic is Q15.16 and can perform about 1.7 GOPS. This performance is enough to calculate the reference CNN in 30ms, much faster than the software we previously tested on the ZYBO board and needed 7 seconds.

The main drawback of the accelerator is that for every filter, the input feature maps have to be streamed again from the SDRAM memory. Thus, if a convolutional layer consists of 10 filters we have to stream the input 10 times as well. The solution to this is to only use only local memory inside the FPGA fabric. Unfortunately the amount which is needed is not usually available at the low-end FPGAs like the one of the Zynq Z010 SoC. In order to overcome the low availability of local memory a tiling mechanism could also let us store locally part of the input every time. In our case though, due to the low performance of the accelerator, the throughput of a single AXI4 high performance port is more than the convolver's demand. So, memory access is not the bottleneck of the current system.

The strong points of the accelerator are its flexibility regarding the size of the feature maps and the depth of the network. Additionally, this design is scalable due to the the fact that the convolver can calculate one filter every time. Thereby, the instantiation of multiple convolvers on the FPGA allows us to calculate multiple filters in parallel. That is possible because each filter's calculation does not depend on other filters of the same layer. Doing so, will also result to less times that the input will have to be re-streamed from the DDR3 SDRAM memory. On the other hand though, it will require higher memory throughput as the output values will be calculated faster.

The last part of the integration is the process of the video stream coming from the camera which is attached on the HDMI input. As the FPGA's memory resources are not enough to store a whole frame, the stream is directed to the external SDRAM memory from where it is streamed again back to the FPGA. Although this is not optimal the latency remains in acceptable levels. The main problem with the camera's stream is that the frames which are captured and then fed to the convolver have different properties from the images of the training dataset.

One of the inconsistencies originates from the sub-sampling we perform on the images. Because the CNN is configured to work with images that have dimensions of 160×120 pixels, we need to scale the frames which originally are 640 pixels wide and have a height of 480. Depending on the scaling method, the resulting image has different smoothness. Other reasons for the inconsistent images are the ultra wide lens and the different color response of the camera which was used.

Unfortunately, those inconsistencies render the object detector incapable of delivering any meaningful output. The whole system cannot provide any information over the content of the streaming images, although it can perform in real-time.

5.1 Conclusion

Overall, the system delivers enough performance to meet the real-time constraint and eliminate the latency of the image acquisition process but it fails to act as a robust object detector. We were able to accelerate the compact CNN but it cannot deliver robust performance for object detection, mainly due to absence of an adequate training dataset.

We also used a pre-trained CNN for feature extraction and this time we were able to achieve very good performance but the accelerator is not capable to accelerate this CNN. Furthermore, the workflow we developed is suitable not only for the specific task of this thesis but may be used to detect any sort of object if the training dataset generator is provided with a small number of images that include the object to be tracked.

After discussing the objectives that were defined we can now answer the research question of this thesis.

Can a CNN be trained in order to act as a single-object detector, accelerated to meet the real-time constrain and integrated into an object tracking system on the ZYBO board ?

Independently of the overall performance, this thesis shows that it is totally possible to use the ZYBO board in order to develop a fast CNN based object tracking system. The Zynq SoC is a very flexible platform and can deliver enough performance to potentially accelerate even some of the state-of-the-art CNNs due to the co-existence of FPGA fabric which is coupled with the fast ARM CPU cores. Furthermore, the tools that are available for development of machine learning applications provide the means to create compact CNN based systems with robust performance for object detection. Some experience on the subject is required though.

Surprisingly the most important aspect of machine learning application development is the training dataset. We show that instead of compiling a training dataset it is possible to generate one, but it can only help when learning transfer is applied and this increases the demand for more powerful hardware. The main problem with the generated training dataset is that its not really versatile and the neural network overfits and exploits color and lighting characteristics. On the contrary, when we use the pre-trained CNN as feature extractor we don't have the same issue as the features we get are a lot and mostly color agnostic.

In general, we observe that there is a trade-off between the two disciplines, namely hardware design and machine learning. On one hand a powerful accelerator will ease the development of the object-detection algorithm and reduce the dependency on a good training dataset. But on the other hand, the ability to train compact CNNs that deliver robust performance will reduce the need for computing power.

Finally, the collaboration with Matheus Terrivel who developed an object tracking algorithm[33] based on a Kalman filter showed that we can process the information of the object detection system and trach the robot. The main issue with the object tracking application was that the image sensor introduces a delay of half second when the image is streamed over Wi-Fi. That was the reason that degraded the overall performance of the robotic setup. This work shows that latency can be greatly reduced. Thus, an object tracking system can rely on the object detector we developed for fast image processing.

5.2 Future Work

Although every aspect of the system that was developed can be greatly improved, it is necessary to compile a large training dataset with images captured from the camera that is connected to the board. This will also provide the ability to evaluate the compact CNN, something that was not possible with the generated dataset due to the inherited patterns that it contains and the difference to the images that are captured from the setup's camera.

If an adequate training dataset is proven sufficient to train a small CNN in order to deliver robust performance at the classification problem, then it will be worth to investigate possible improvements of the localization algorithm. For now, the method that is used cannot provide information about the size of the object. However, this can be done if a more sophisticated algorithm is used in order to detect the blob which represents the object in the final convolution layer's output. Furthermore, it will be possible to detect multiple objects as well, if the algorithm is able to detect more than one blobs. Finally, we could also evaluate the possibility of extracting the object's location directly from the CNN if we force it to predict bounding boxes instead of the classification technique we used.

Promising are also methods that have been developed in order to lessen the computational burden of CNNs. Applying such techniques will let us reduce the number of operations that the state-of-the-art neural networks require. thus, allowing us to use such a network for feature extraction without the need of a powerful accelerator. Usually this can be achieved by applying quantization on the weights and the data itself or by pruning the CNN [20].

The first approach will result to more parallelism as the resources which will be not needed with a compact arithmetic representation can be used to introduce more processing elements. Thus, increasing the number of operations per clock cycle on the FPGA. The second approach will reduce the size of the network itself. By investigating which filters contribute more to the final output of the network. We can then get rid of those that do not, with a small performance penalty.

The aforementioned improvements all relate to the development of the CNN based object-detector but the accelerator can be improved as well. Firstly, the clock frequency can be increased by introducing more stages to the pipeline which now has 4. This change together with a more compact fixed point representation can result to a great increase of the performance. Furthermore, compact arithmetic will also allow us to fetch more values from the bus simultaneously so we could also calculate more that one outputs per clock cycle. An indication of the performance that we can be expecting from the Zynq SoC can be found in [26] where a very similar accelerator was developed.

Probably the most crucial improvement though is the increase of the kernel size which is restricted by the number of multipliers that we use. But, this will require a sophisticated interconnection between the line buffers of the convolver and its multipliers.

In general, the convolver from a system perspective offers a lot of flexibility and by further developing the driver we can also accelerate more complex architectures like the MobileNets or even introduce tiling by developing the appropriate buffer memory on the FPGA.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, Jan 2017.

[3] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. Shidiannao: Shifting vision processing closer to the sensor. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 92–104, June 2015.

[4] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88(2):303–338, June 2010.

[5] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Region-based convolutional networks for accurate object detection and segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(1):142–158, Jan 2016.

[6] Xavier Glorot and Y Bengio. Understanding the difficulty of training deep feedforward neural networks. 9:249–256, 01 2010.

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, Cambridge, MA, 2017.

[8] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang. Angel-eye: A complete design flow for mapping cnn onto embedded fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1):35–47, Jan 2018.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[10] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[11] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama, and K. Murphy. Speed/accuracy trade-offs for modern convolutional object detectors. In *2017 IEEE Conference*

on Computer Vision and Pattern Recognition (CVPR), pages 3296–3297, July 2017.

[12] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 4107–4115. Curran Associates, Inc., 2016.

[13] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016.

[14] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, and Jonathan Ross. In-datacenter performance analysis of a tensor processing unit. 2017.

[15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[17] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[18] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.

[19] 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.

[20] J. H. Luo, J. Wu, and W. Lin. Thinet: A filter level pruning method for deep neural network compression. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 5068–5076, Oct 2017.

[21] D. P. Moeys, F. Corradi, E. Kerr, P. Vance, G. Das, D. Neil, D. Kerr, and T. Delbruck. Steering a predator robot using a mixed frame/event-driven convolutional neural network. In *2016 Second International Conference on Event-based Control, Communication, and Signal Processing (EBCCSP)*, pages 1–8, June 2016.

[22] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi. Design space exploration of fpga-based deep convolutional neural networks. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 575–580, Jan 2016.

[23] Urs Muller, Jan Ben, Eric Cosatto, Beat Flepp, and Yann L. Cun. Off-road obstacle avoidance through end-to-end learning. In Y. Weiss, B. Schölkopf, and J. C. Platt, editors, *Advances in Neural Information Processing Systems 18*, pages 739–746. MIT Press, 2006.

[24] Maxime Oquab, Léon Bottou, Ivan Laptev, and Josef Sivic. Is object localization for free? – Weakly-supervised learning with convolutional neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition*, Boston, United States, June 2015.

[25] D. A. Pomerleau. Efficient training of artificial neural networks for autonomous navigation. *Neural Computation*, 3(1):88–97, March 1991.

[26] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 26–35, New York, NY, USA, 2016. ACM.

[27] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788, June 2016.

[28] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[29] M. Schmidt, M. Reichenbach, and D. Fey. A generic vhdl template for 2d stencil code applications on fpgas. In *2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, pages 180–187, April 2012.

[30] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *CoRR*, abs/1312.6229, 2013.

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

[32] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261, 2016.

[33] M Terrivel. Computationally efficient vision-based robot control, December 2017.