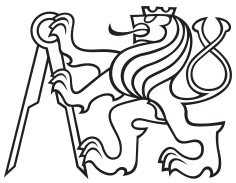
Bachelor Project



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Measurement

Object recognition using an FPGA on an embedded platform

Tibor Rózsa

Supervisor: Ing. Jan Kovář
Field of study: Cybernetics and Robotics
Subfield: Sensors and Instrumentation
May 2018

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Rózsa Tibor** Personal ID number: **452919**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Measurement**

Study program: **Cybernetics and Robotics**

Branch of study: **Sensors and Instrumentation**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Object Recognition Using an FPGA on an Embedded Platform

Bachelor's thesis title in Czech:

Rozpoznávání objektů v obrazu viditelné kamery pomocí FPGA

Guidelines:

1) Get familiar with algorithms for software-based object recognition and the limitations and portability of given algorithms to FPGAs.
2) Get familiar with the software Quartus, the limitations of the FPGA CycloneV 5CGXFC4C7U19C8, and the data protocols and interface requirements of the cameras Tau2 and Tamron.
3) Prepare and test the LVDS-deserialisation module for the input data(4-lane, 519.75MHz) from the camera Tamron. Verify the deserialisation and transfer reliability by previewing the picture from the camera, and by pixelwise comparison. The input data from the Tamron camera are of the protocol BT.1120, YCbCr.
4) Implement object recognition(human silhouette, cars, car logo/brand) in the picture of the camera Tamron. Implement the algorithm and training in software and then use the algorithm/training data to implement the classifier in the FPGA.
5) Discuss the achieved results.

Bibliography / sources:

[1] Computer Vision: Algorithms and Applications Richard Szeliski
[2] Digital Video Processing for Engineers Suhel Dhanani Michael Parker
[3] FPGA Implementations of Neural Networks Amos R. Omondi Jagath C. Rajapakse
[4] EURASIP Journal on Applied Signal Processing 2005:7, 1047?1061
[5] arXiv:1603.05279v4 [cs.CV] 2 Aug 2016

Name and workplace of bachelor's thesis supervisor:

Ing. Jan Kovář, Workswell s.r.o., Libocká 653/51b, 16100 Praha 6

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **07.12.2017** Deadline for bachelor thesis submission: _____

Assignment valid until:
by the end of summer semester 2018/2019

Ing. Jan Kovář Supervisor's signature	Head of department's signature	prof. Ing. Pavel Ripka, CSc. Dean's signature

Acknowledgements

I would like to thank Jan Kovář and Workswell s.r.o. for providing me the sufficient resources to complete my work and the opportunity for my thesis to have real-world impact; Jakub Jirsa, Jan Ječábek, Pavel Píša and Roman Bartošinsky for their advice and help when dealing with the platform; Jiří Matas for providing me the necessary background in machine learning as part of the course Rozpoznávání a strojové učení; and finally my family and friends for their everlasting support.

Declaration

I declare that I wrote the presented thesis on my own and that I cited all the used information sources in compliance with the methodical instructions about the ethical principles for writing an academic thesis.

In Prague, 25. May 2018

Abstract

Since the overwhelming success of the AlexNet deep neural network architecture in the year 2012, deep learning has become the go-to for many image-processing applications. However, until recently they have been limited to the high-performance CPU/GPU world, for the number of parameters made it impossible to store and use larger neural networks in embedded devices. With the advent of binarized neural networks and XNOR-convolutions this limitation has been lifted, and it is now possible to implement high-precision, low latency, real-time classifiers and other image processing solutions on embedded devices. Our application explores such an implementation on a reconfigurable hardware platform, FPGA, to offload the comutational complexity of the task from the main system processor of the embedded system.

Keywords: FPGA, binary neural network, machine learning, object detection

Supervisor: Ing. Jan Kovář

Abstrakt

Po enormním úspěchu architektury hluboké sítě AlexNet v roce 2012, se stalo hluboké učení spolehlivým řešením pro aplikace zpracování obrazů. Nicméně do nedávné minulosti byly tyto aplikace limitovány na vysoce výkonný CPU/GPU, počet parametrů totiž znemožňoval skladování a používání větších neurálních sítí ve věstavených zařízeních. Objevením binárních neurálních sítí a XNOR-konvolucí se tyto limitace odstránili a umožnili použití klasifikátorů s vysokou přesností, malým zpožděním, v reálným čase ve věstavených zařízeních. Naše aplikace implementuje klasifikátor tohoto typu na překonfigurovatelné hardware platformě, FPGA, aby odlehčil hlavní procesor věstaveného sytému od komplexity úkolů.

Klíčová slova: FPGA, binární neurální sítě, strojové učení, detekce objektů

Contents

Figures

Tables

Chapter 1

Introduction and premise

Lately, machine learning has seen an unprecedented uprise in almost all areas of image processing applications, mainly due to the enormous success of deep neural networks for these uses(see section Achievements and use cases). However, because of the computational complexity of these applications, most neural networks have either such long latency that they cannot be used for practical applications or require high power and performance GPUs to achieve real-time performance. Neither make them viable solutions for embedded applications.

Our application proposes the implementation of an object detection system on a low power, low cost FPGA to offload the computational burden from the CPU on our embedded platform. As our analysis of the currently used best object detection algorithms in chapter Limitations, precision and portability of object-recognition algorithms reveals, the best compromise between performance and precision for embedded applications are binary or ternary neural networks; our application explores the implementation of a binary network, XnorNet for maximal compression.

The purpose of this work is to present the basis for a real-time classifier implemented in the fabric of the FPGA for detecting objects in the picture of a connected 30Hz fullHD camera. The Proposed system architecture serves exactly this purpose: it enables multiscale object detection, sliding window classification and classification/localization post-processing, outside of the classifier itself. This work only deals with the implementation of the camera-FPGA interface and the implementation of the classifier; the implementation of the full image processing system is left for Future work.

Part I

Theoretical background

Chapter 2

FPGAs

2.1 Overview

Since the advent of digital logic, there has always been a need for reconfigurable hardware devices. First with manually switching between connections in a transistor array, then by using Programmable ROMs, and Programmable Logic Devices, they offered an option for rapid development and prototyping of hardware.

Nowadays, Field Programmable Gate Arrays provide the best of both worlds for high-performance applications in reconfigurable computing, combining pre-made hard IP and RAM blocks and reconfigurable hardware. They fill the niche not only for prototyping and development, but for full-fledged products, where the production count is reasonably low.

Application Specific Integrated Circuits are of course almost always a much cheaper option for larger quantities, as the logic does not have to be implemented in generalized lookup tables and routing(generally 90% of FPGA chip area!), but can be optimized for minimal chip area and number of metalization layers.

Another advantage of reconfigurable logic is that it is really safe in terms of bug severity and correctability. If a company chooses not to hard-code their hardware in the final product, so the code in the FPGA cannot be externally

changed, they can course-correct with applying patches, and thus reduce fatal flaws. They can also implement brand new features and enhance the used IP long after the product is rolled out.

The reason why most companies prefer the hard-code approach is of the fear of reverse-engineering, as the code loaded in the EEPROM memory of the FPGA contains the entire design. However, there are options to encrypt the design and make anti-tamper measures, that are of widespread use. Industry leaders Altera[HQ09] and Xilinx[McN12] offer a wide variety of options to ensure the safety of hardware IPs. Generally the strongest threat to IP safety is the simple complacency of design teams and companies – design security is not taken as an important factor and is thus neglected. Legal protection of code may cover these issues, but it is overwhelmingly expensive compared to taking simple security measures at design time.

2.2 Brief History

The evolution[cit] of programmable logic devices started with using PROM memory. These could be programmed either by the user(field-programmable) or at a factory using masks. PROMs are still used in the industry. In FPGAs Electronic Erasable PROMs are often used for loading code, as they can be erased and reprogrammed many times.

The next step towards FPGAs was the appearance of Programmable Logic Devices. These implement AND and OR gates in a gate array chip. They can also be both factory programmable and user programmable.

In 1985 with the invention of the FPGA Xilinx created a device that not only had programmable gates, but also programmable interconnections between gates, extending the functionality of PLDs. Later increasingly efficient compilers, first by Steve Casselman extended the functionality of FPGAs to be completely programmable by software.

FPGAs were finally brought into spotlight by Adrian Thompson in 1997, who, using a genetic algorithm managed to create a sound recognition device of extraordinary quality. The device was not only better at recognition than the competitors, but used a mere 20% of the logic resources of the best existing human design.

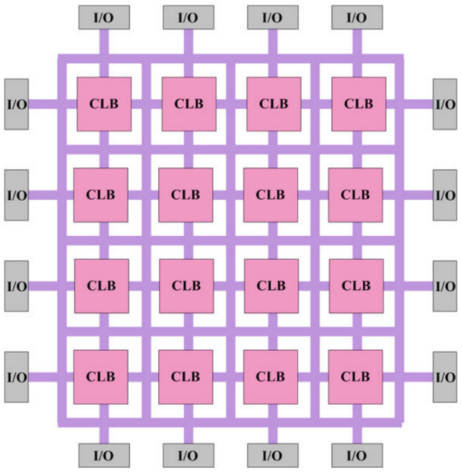


Figure 2.1: Overview of FPGA architecture

Since then, FPGAs have spread to all types of industries, lately outperforming GPUs in advanced image processing[NVS+17].

2.3 FPGA Structure

FPGA structure can roughly be divided into four different types of structures[UF12]:

1. Programmable logic blocks(generally called Confgurable Logic Blocks), which implement logic funtions
2. Pre-implemented Hard IP blocks without changeable structure
3. Programmable routing that implements said functions
4. I/O blocks used for off-chip connections

2.3.1 Configurable Logic Blocks

They are the basic building blocks of logic inside the FPGA. They can range in size and complexity: there is no theoretical ideal size of these blocks. FPGA manufacturers have to make a compromise between the required interconnect and unused logic area in the block, and select the size and content of these

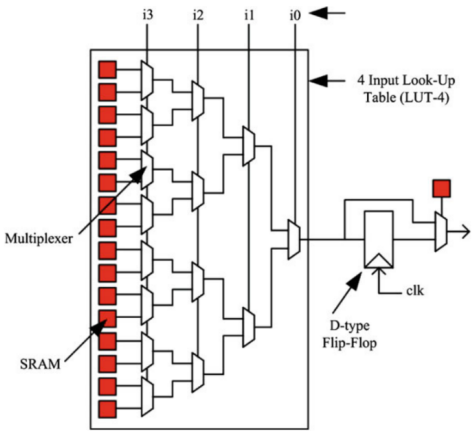


Figure 2.2: Example of a Basic Logic Element using a LookUp Table

blocks accordingly. As a tangible example we can take a transistor versus a processor as a building block: the former requires an awful lot of interconnect routing to create complex structures and is thus ineffective; the latter almost doesn‘t even require interconnect, but there is an unreasonably large area of unused logic in most applications.

One requirement CLBs have to satisfy is functional completeness[End01]. In essence, a set of Boolean operators(logical functions such as AND, OR, etc.) is functionally complete, if any logical function/truth table can be expressed by their finite combination. Ideally, CLBs should implement a logical operator that is functionally complete by itself, so only one type of CLB is needed to construct any logical function.

Examples for CLBs used in the industry are blocks made of NAND gates(the smallest functionally complete Boolean operator). More commonly used by commercial vendors like Xilinx and Altera are blocks based on lookup tables(LUTs), which are basically small pieces of memory. LUTs are generally used, because they provide a compromise between the too fine-grained gate-level designs and the too coarse-grained logic blocks. CLBs can use a single or multiple basic logic elements(BLEs). Modern FPGAs usually use 4-10 locally connected BLEs in a cluster.

2.3.2 Hard IP Blocks

Many FPGAs, in addition to BLEs implement specialised blocks such as memory,adders, multipliers or I/O blocks for specific protocols(PCIe, UniPHY

memory). These are very efficient in their function, as they are basically ASICs implemented in the fabric of the FPGA, but may end up as a large waste of on-chip area if left unused.

2.3.3 Programmable routing

FPGA routing resources generally consist of a combination of wires and programmable switches. These switches route the I/O of CLBs to each other. These routing resources must be very flexible, but also need to be efficient. Generally, design routing requirements have a local nature, with only a few signals needing to cross larger distances.

FPGA routing architectures exhibit either a hierarchical or an island-style nature. Hierarchical architectures take advantage of the locality of designs, and as such distribute local, semiglobal and global routing; island-like architectures have uniform, horizontal and vertical array-like routing consisting of wiring, programmable switches and connection boxes. Logic blocks connect to connection boxes, and are then routed using switches through wiring. Usually wiring of multiple lengths is provided to ensure that the router software is able to meet the timing requirements of the design.

Hierarchical FPGAs usually group together clusters of CLBs, connect them locally similarly to the island-style architecture, and then connect the local group to an interconnect on a higher level of hierarchy, thus getting the benefits of both architectures. However, the coarseness of the hierarchical layers also has to be carefully selected, using similar considerations as when selecting the size of CLBs.

2.4 Implementation of Neural Networks – Comparison with CPUs, GPUs and ASICs

In a fairly recent paper [NVS+17], FPGAs have been compared to CPUs and GPUs in efficiency. As the training and usage of convolutional neural networks is highly parallelizable, this is a reasonable comparison - FPGAs and GPUs tend to flourish over CPUs in such an environment.

The paper explores the usage of Stratix 10 FPGAs for ImageNet-type chal-

lenges and uses figures on the 2017 winners, ResNet, and historical winners such as AlexNet, VGG-Net and GoogLeNet to emphasize its point. In the experiments provided, the new Stratix 10 FPGAs use their large on-chip memory, many DSPs and high frequency support to their advantage, outperforming the industry leader Titan X Pascal GPU up to 5.4x in operations/s using the aforementioned latest winner architectures. They also have a much larger performace/power figure than what the GPUs can offer, and so are far more viable for embedded applications.

Although in our application we use a much weaker FPGA, a Cyclone V, there are general trends and figures we can take away from this paper:

1. Deeper neural networks tend to have higher accuracy, but come with increasing computational cost
2. Neural networks tend to suffer small losses in precision when decreasing weight bitcount
3. Efficient pruning can significantly decrease computation costs

The most important takeaway from these are the effects of decreasing bitcount. Ternary (only using weights 0,1,-1) networks tend to suffer a mere 1% of accuracy loss and can be implemented much more efficiently in FPGAs; binary(-1,1) networks can replace convolutions with a xnor followed by bitcount, which are much simpler operations. Even new generation GPUs are restricted to a minimum of 8 bit operations without wasting resources - thus FPGAs are much more efficient for the implementation of these networks.

As for every task an FPGA can do, we can design an ASIC to do the job much faster and cleaner for machine learning tasks, too. There already are neural network accelerator ASICs used in the market, such as the Google TPU accelerator. However, as the field evolves with insane speed and time to market is generally much longer for ASICs, they have to be generalized to be robust enough for changing needs. This takes away from the promise of specialized hardware, makes them much more like GPUs and makes reconfigurable hardware a much more robust choice for such a fast moving field. After all, nothing prevents FPGA developers to implement a similar-size, much larger accuracy network(e.g. the current ImageNet winner) in an image processing product already sold to customers and apply it as a patch or purchasable bonus without changing the hardware in any manner.

All in all, FPGAs are proven to be a viable solution for large-scale machine learning problems. As such, if we scale the problem down, there is a promise

that we can also scale down the hardware footprint required for implementation. This allows us to implement efficient machine learning accelerators on low-power, low-cost FPGA chips and can thus relieve system processors and GPUs of much stress.

Chapter 3

Object detection algorithms in machine learning

3.1 Overview

Although lately we have seen an emergence in wholistic object detection methods[RDGF15], object detection generally consists of two parts: object localization and object classification. For long, object localization was either performed brute-force by using sliding windows and evaluating each sub-image and the emphasis of research remained in speeding up and improving classification.

Object classification has its roots in the traditional methods of machine learning. These can be divided to two subcategories, based on their fundamental approach to classification: statistical and empirical methods. Due to lack of generalization and low precision machine learning has for long had limited success in real-life image classification problems. It generally got to the spotlight in 2001 by the enormous success of the Viola-Jones object detection framework, detecting faces with an unprecedented accuracy.

Since then, multiple approaches have been used for object detection. Support Vector Machines are a prime example of these; however, since the outstanding success of the AlexNet neural network in the 2012 ImageNet image classification task, the industry is dominated by the use of neural nets trained by reinforcement learning.

Although wildly effective, neural networks do not pose a silver bullet to object classification. There has been a large amount of research done in the field of fooling neural networks and finding their weaknesses, mainly by the pioneers of the field: [SZS+13],[SVS17]. These papers show methods of generating adversarial examples in knowledge of the structure of the neural net to create high-confidence false classifications, by altering the input image in a way that is seamless to human observers. Thus, one area of neural network usage that is not yet reachable is security and safety-critical applications.

3.2 Localizaton strategies

For localization, there are different strategies one can use. A common strategy for simpler methods was to use brute force - select a few suitable scales and slide windows of those scales through the image, classifying each of them along the way. Although neatly parallelizable, this is a highly ineffective method, as in almost any image there is likely to be overwhelmingly more negative regions/windows than positive ones, where there actually is an object of interest. Thus, there exist a variety of methods for reducing the number of areas to be classified.

One of the methods is to use a cascade of classifiers[SWvdH10] - here the classification is broken down to stages. Each stage by itself poses as a low-precision classifier, but each stage acts as an increasingly more accurate filter for obviously wrong examples. As such, only the hardest cases get to the last few stages, and an immense amount of computation is saved.

Another method for reducing computation time is by fast localization algorithms preceding the actual classification. One of these algorithms,R-CNN uses a neural network for fast localization [RHGS15]. There are also existing traditional image processing algorithms, such as Scale-Invariant Feature Transform or Speeded Up Robust Features. One additional approach is to use image segmentation, breaking the image into parts and then using a neural network for object detection on those segments [ESTA13].

Overall, the localization problem is far from being solved. The state-of-the-art solutions are still below human-level performance in accuracy, which cannot be said about the detection problem.

3.3 Statistical methods

Statistical methods used in machine learning environments operate either with known or estimated probability distributions of input variables. The machine learning problem essentially boils down to correctly estimating the parameters or hyperparameters of such a distribution and thus creating a classifier that, dependent on the input chooses the class with the highhest probability density for that given input. The learning works by minimizing structural risk[VH17a] - which is essentially the sum of weighted decision costs on given inputs multiplied by the probability of those inputs. A simple example for decision costs is having a cost of 1 for a wrong classification and a cost of 0 for a correct one - thus, high probability wrong classifications increase cost more than less probable ones and right classifications have no effect on it.

This approach has several drawbacks: the probability distribution function may be unknown, nonexistent or extremely hard to estimate. For unknown probability distributions, probability estimation methods such as Parzen windows[JM16] can be used; the other two cases are essentially unsolvable for statistical methods.

Even in this case, statistical methods are good solutions for a variety of simpler problems. They excel in their low on-site computation cost, as their solutions can be precomputed - then, decisions can be derived by simply checking into what class interval the input variables fall.

3.4 Empirical methods

3.4.1 Overview

Empirical methods tend to have a different approach to the problem - they work on the principle of minimizing the error on the training set[VH17b]. As such, not knowing the required degree of generalization is a big problem for them and they tend to overfit by nature. Thus, the learning algorithm has to have a method for maintaining a required level of generalization.

There are several empirical methods that have been used with large success

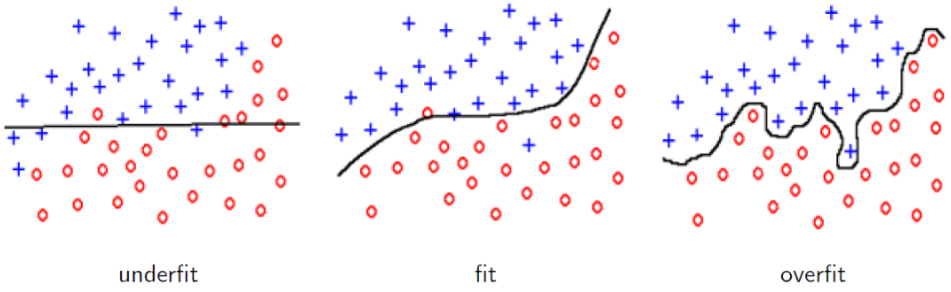


Figure 3.1: Underfitting, correct fitting and overfitting classification functions

on previously unsolvable problems: AdaBoost with cascade classifiers for face recognition; Support Vector Machines and Neural Networks for a variety of image classification problems. The following sections provide a rough description of these algorithms with their strenghts and weaknesses emphasized - we close out on neural networks, as they are the state-of-the-art solution for a multitude of image classification problems.

3.4.2 Adaboost

Adaboost[Jo17] is an algorithm for designing a strong classifier, $H(x)$ from weak classifiers $h_t(x), (t = 1 \dots T)$, selected from a classifier set. The strong classifier is constructed as a linear combination of the weak classifiers:

$$H(x) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

Both $H(x)$ and $h_t(x)$ output -1 or 1 depending on the class.

The Adaboost algorithm deals with both the selection of the weak classifiers and with finding the weights α_t, which are proportional to the classification error of the weak classifier. It works on the premise that the succeding classifiers should be more sensitive to previously falsely classified data. By that it can get a hold of classifying the harder examples, close to the classification boundary. It does this assigning a weight to each data point, and applying this weight in the error calculation when selecting the best classifier for the set. As such, the later classifiers may not have the objectively smallest errors, but classify problematic data points correctly.

The algorithm consists of the following steps:

1. Find the classifier with the lowest weighted error on the dataset:

$$\epsilon_t = \sum_{i=1}^{L} D_t(i)(h(x_i)! = y_i)$$

2. If this error is larger or equal to 0.5, terminate
3. Compute the weight of the selected classifier: $\alpha_t = log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$
4. Update the data weights:

$$D_{t+1}(i) = \frac{D_t(i)e^{-\alpha_t y_i h_t(x_i)}}{\sum_{j=1}^{L} D_t(j)e^{-\alpha_t y_j h_t(x_j)}}$$

The strength of the Adaboost algorithm is that the choice of the classifier set is arbitrary. The model gets increasingly precise with each added classifier with an error smaller than 50%. This makes it applicable for a variety of problems, although many times it is not clear what set of classifiers should be used for the problem, and the choice mainly comes down to personal preference.

3.4.3 Support Vector Machines

Support vector machines [JM17] are a type of linear classifier, and thus have a low computational cost. Unfortunately, there are a few major problems when trying to use linear classifiers:

1. Even if the classifier classifies the training data correctly, we cannot be sure it is generalizes well
2. The data may have a somewhat linear boundary, but may not be linearly separable
3. The classes are likely to have a nonlinear classification boundaries

The premise of support vector machines is mainly to tackle the first problem, but they provide solutions for the other two as well. They introduce the notion of margin - which is the distance of the closest data points to the classification boundary. Intuitively, we would like this margin to be as large as possible, so that the two classes are distinctly separated. Thus, SVMs

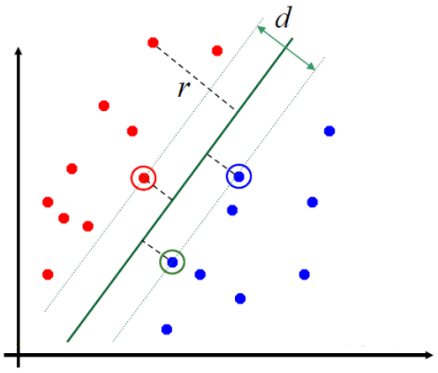


Figure 3.2: The margin d between two classes

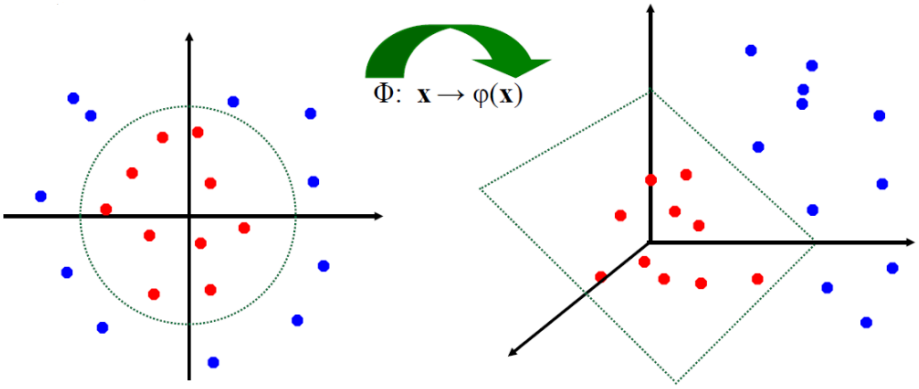


Figure 3.3: Dimension lifting

simultaneously maximize the margin and minimize the error on the training set. For linearly separable data, the training error should be zero at the end of training. For linearly nonseparable data, slack variables can be added to the equation we optimize, which can allow misclassification of data points with some penalty.

For data where the boundary is likely to be entirely nonlinear, we can use dimension lifting - nonlinearly transforming the training data to a space where it is linearly separable. This trick is applicable to other machine learning algorithms as well, however, with SVMs it is extremely powerful. The first reason is that this enables the computationally efficient linear classifier to be used on problems where the boundary is not linear. The second reason is that the mathematical formulation of SVMs allows the transformation functions that do not have an explicit declaration or have infinite dimensionality(e.g. Gaussian functions).

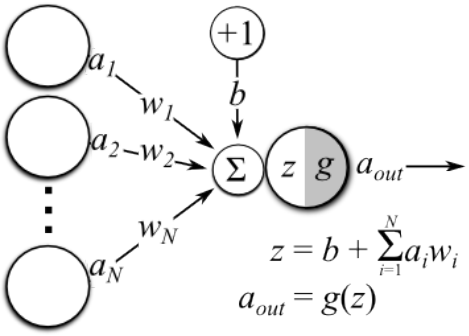


Figure 3.4: A neuron with inputs $a_1..a_N$ and an activation function $g(z)$

3.4.4 Neural Networks

Structure

Neural networks are a computational model inspired by the way biological neural networks process information. Their fundamental building blocks, the neurons consist of two main parts:

1. A linear combination of the inputs, using the learned weights and bias of the neuron
2. A (generally) nonlinear "activation" function

As we can see on Figure 3.4, the inputs to the neuron get multiplied with the learned weights, summed, and then a learned bias gets added. The resulting value is then entered into the activation function, $g(z)$. This activation function is generally nonlinear. If it weren't, the network would be reducible to a single layer by linear combination of weights and as such wouldn't be able to tackle problems with nonlinear classification boundaries(e.g. the xnor function). It also needs to be differentiable to allow training by backpropagation; more about this in the Training section.

To be able to approximate increasingly complex functions, neurons are often stacked into layers. These layers are generally of three types, as shown on Figure 3.5: the input layer, hidden layers and the output layer. The input layer consists merely of the data to be processed and the output layer is the result of processing, be it classification, segmentation, etc. The intermediate hidden layers provide the computations necessary to perform the function

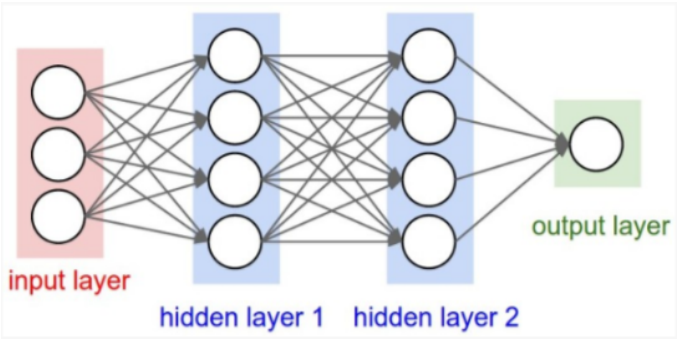


Figure 3.5: A network with the input, hidden, and output layers

of the neural network. Each the inputs of each neuron in a hidden layer are either the input layer, or the activations of the neurons in the previous layer.

As the field of deep learning(deep means more than one hidden layers) progressed, additional types of intermediate/hidden layers appeared that either sped up processing or improved accuracy. From now on, we will refer to layers, where each neuron is connected to each neuron in the previous layer as Fully Connected layers. The most prevalent types of additional layers used are:

1. Convolutional layers
2. Pooling layers
3. Normalization layers

Types of layers

Convolutional layers [sta] are essentially fully connected layers, where all the weights of non-local inputs are zero. For example, if the input layer is a 2D image and it connects to a hidden layer with the same dimensions, each neuron is only affected by the weighted values in a small window around the given pixel. This window can be of arbitrary size, but generally the sizes of 3x3,5x5,7x7,11x11 are most popular. The choice of odd dimensions ensures that the middle pixel is surrounded by equal number of pixels from each side, favoring no particular direction.

As can be expected, convolutional layers reduce computational complexity by an extreme margin. Additionally, when stacking multiple convolutional

layers on top of each other, the neurons in the deeper layers are affected by a much larger window of neurons. Also, for image processing, convolutional layers make intuitive sense. Small features such as edges and curves are detected in the first few layers; then these features are combined in the later layers to represent more complex features. For edge detection, we only need a few pixels around the edge, so we can save redundant computations. To make global assumptions about an image, classifying networks generally end with a fully connected layer - thus the local features are added together with weights dependent on their locality and thus we can make global assumptions about the contents of the image.

Pooling layers reduce the dimensionality of the image between convolutional layers. This serves two purposes: reducing the number of computations, and extending the influence of convolutional filters in the deeper layers of the network. Pooling can be done on multiple scales and by multiple operations. Generally, the 2x2 pooling is preferred, as pooling larger areas leads to an unreasonably high loss of features. Pooling operations can be either average or max-pooling, although generally max-pooling is preferred. For many applications, pooling is replaced by using a higher stride on convolutional windows. This means moving convolutional windows by more than 1 pixel at a time.

Normalization layers[IS15] are mostly useful during training. They reduce really low or really high activations - consequently, we can use higher learning rates. They also reduce overfitting, as they have slight regularisation effects.

Training

Neural networks are trained by gradient descent. The weights and biases of the network are initialized randomly or pseudorandomly. Then a training example is propagated through the network, and the output from the final layer is compared to the ground truth(labels in case of classification) - a gradient is calculated that represents how much each weight in the previous layer has to change to minimize error between the ground truth and the output. This calculation is done for all layers, propagating backwards - hence the name backpropagation. This is done either for a single training example, or a randomized subset(batch) of the training set, or the whole training set. For the multiple example case, the gradients are averaged, multiplied by the learning rate and added to the network weights. The whole process is repeated until the network reaches a reasonable accuracy.

Figure 3.6: Image segmentation - each class painted with its unique color

Updating a weights for every training example(stochastic gradient descent) makes learning unstable, as it introduces noise into the gradient descent. Ideally we would like to backwards-propagate the whole training set before updating the weights. However, this would take a really large amount of time. Thus, a compromise is chosen between learning speed and accuracy: batch learning. Here, a small randomized subset of the training set is selected, and weight update is done after the backprop of this subset. If this batch is sufficiently large and random, it can represent the training set quite accurately; thus the noise added to gradient descent is small, but we save an enormous amount of computation time.

3.5 Practical use of Neural Networks

3.5.1 Achievements and use cases

Neural networks have been lately popularized by their efficiency at various image classification and detection tasks. One of the major achievements can be contributed to the neural network named AlexNet, which won the ImageNet image classification and localization contest in 2012, leaving its competition behind by an astounding 10% precision margin. Before this breakthrough, the scene was dominated by SVMs using histograms of oriented gradients. Nowadays, all we can see are deep neural networks. The ImageNet problem is a particularly hard one - classifying 1000 types of objects and localizing them on a picture is no trivial task. This makes the success of AlexNet and succeeding networks even more profound and has led them be used for all sorts of image and sound processing tasks.

Since 2012, neural networks with various architectures have gotten incrementally better at the ImageNet task. Human level performance(95% precision) has been exceeded in 2015; since then, the top-1 accuracy of classification has risen to 96-97%. This is astounding compared to the 84% precision

of AlexNet - the classification error has been reduced to almost 1/5th of its former value.

As previously mentioned, they excel at solving other types of problems as well. The main criteria(one of several) for their correct function is the presence of a large, compared to other datasets enormous dataset. As such, many of these are related to image or audio processing, for instance image segmentation [WLZ+17] (as seen on Figure 3.6), image enhancement [IKT+17] or image and audio compression. Other important uses are text translation and text-to-speech, which have also lately achieved close-to-human performance[SPW+17]. Image generation from text or other sources is also an important application - neural networks are used to accelerate computations of physics-heavy graphics simulations, such as scattering in clouds[KMM+17] or viscoelastic fluid simulations[BGFAO17]. They also have been extensively used in search engines and article/video recommendations on various sites.

3.5.2 Limitations and proposed solutions

The beforementioned achievements make neural networks seem like a silver bullet to all problems. True, they are really powerful on a multitude of problems, but for others, they are either outperformed or even if not, have such a low success rate that they are not worth implementing.

One of the major drawbacks of neural networks is their computational complexity. Larger architectures, often used in image processing applications generally require hundreds of megabytes to simply store them, let alone be in use. They also require a large number of computations for each forward pass(==classification), so they are difficult to implement in single-CPU embedded devices without using a dedicated GPU. Lately quite a significant portion of research has emerged in this topic. One approach is pruning the network to require less connections without accuracy loss[HZYN18], [HMD15],thus significantly reducing the number of forward-pass computations. Another approach is to reduce the bit width of weights and thus computations, to reduce the memory requirements of a neural network and in some cases speed up computations. The usual bitwidth for neural network computations is 32 bits. However, recent research has shown that ternary(2-bit: -1,0,1) networks can be trained for ImageNet-type problems have only 1% accuracy loss. The extreme is the usage of binary neural networks(-1,1), that have lately reached only 5% accuracy loss from the full precision networks[THW17a], while remaining 32 times smaller in size.

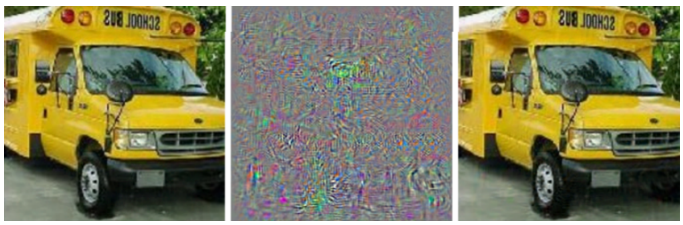


Figure 3.7: Adversarial example added by adding specialised noise -the image on the right is classified as an ostrich

Another drawback has been noted by Szegedy et. al. in 2013 [SZS+13]. If the architecture of our network is known, we can generate noise, that when added to the picture makes no significant visual difference, however, it generates high activations for a given class regardlesss of the original image. This means it fools the network - as seen on Figure 3.7. Lately, Su et.al. [SVS17] have explored how far they can go with this - they have proven, that it is enough to change a single pixel in an image for the misclassification to occur. This property of neural networks has two main dangers. One of them is the reliability to use such classifiers in a real-world, noisy environment; the other is using them for security. As for now, until the problem of adversarial examples is resolved, it is not preferred to use neural networks for high security or reliablility applications. We can train the network by enriching the training set with these adversarial examples, as done in [GSS15], as such, it can reject adversarial examples to a certain degree.

Although there has been much research in the field, there still isn't a mathematical theory on why neural networks exactly work, and how exactly they fail when they do. This, and the rather blackbox-like nature of the classifier(we do not implicitly know the types of features a neural network selects, only the output) can be a serious drawback when trying to decipher how neural networks make decisions. As they are increasingly used in a variety of industries, including security, finances and crime prevention, transparency must be key. Otherwise a wrong decision made by a neural network can have fatal consequences, as there is no way to check whether the classifier was biased for that particular decision.

For this reason, new techniques such as feature visualization[OMS17], activation visualisation(as seen on Figure 3.9) and other techinques such as distilling the neural network into a decision tree [FH17] are used to make the inner workings of neural networks more transparent. The most straightforward of these is visualizing activations - we can see, whether our filters get excited about the features we care about, or the background. This is an useful check, however, it can be misleading if we forget about false positives - this is where feature visualization comes into play. This boils down to generating images by optimisation that maximally activate the neurons in a filter. We have to

Figure 3.8: Feature visualization by optimisation - dataset examples on the left, generated image on the right

Figure 3.9: Visualizing activations - the input image(left) and the activations of a trained convolutional filter(right)

be careful not to get stuck in local optima, but we can get a rough idea about what that filter is looking for (Figure 3.8).

One of the prerequisites of using neural networks is the abundance of training data. As the training dataset grows and gets more diverse, the network gets more precise and is less prone to overfit. However, for many problems we do not have a large dataset to use, so we have to expand our existing dataset - and there are multiple methods for doing that besides simply collecting more data.

One of them is to augment the dataset using traditional methods - rotations, clipping, flipping, modifying the color palette. These methods increase precision quite significantly, as explored in the paper [PW17]. The same paper explores the possibility of augmenting data by performing complex style transformations on them by generative adversarial networks, and proves that it can help provide a significant edge for certain classification categories. It is generally useful to mix and match between these techniques to achieve maximum performance.

3.5.3 Binary neural networks and Xnornet

There have lately been several attempts to reduce the size of neural networks, so they can fit into embedded devices. One of these attempts is XnorNet [RORF16], in which the researchers have reduced the bit width of both the weights and the input layers to 1. Their values can thus be 1 or -1. This also simplifies the multiplications and additions required for convolutions to xnor and bitcount operations, which are much simpler to execute.

Reducing the bit width of filters works the following way: the binary values of the filter derived by moving them through a sign function(1 for positive, -1 for negative), and also a scaling factor is computed which is the average value of the filter values. The binarization of the data layers is done in a similar fashion. The researchers have demonstrated that the network can perform with faster inference and 32x smaller size with 10% precision loss on the ImageNet dataset. Later, Tang et. al. [THW17b] have shown that this difference can be reduced to 4-5%. Binary neural networks show real promise for embedded applications and are also less prone to overfit and be fooled by adversarial attacks [GTM17].

Part II

Practical work

Chapter 4

The image processing platform

4.1 The Cyclone V 5CGXFC4C7U19C8 FPGA

The Altera Cyclone V FPGA series [cyc] provide a low power, low cost variant of the FPGA market, and are best suited for embedded applications. Their high-speed transceivers (up to 3.125 Gbps) make them highly usable for image processing and other applications that require high-bandwidth. Often they are combined with an ARM CPU on a single chip to form a SoC. The increased use of Hard IPs enables the Cyclone V series to cram more computational units of smaller area and power requirements to the chip. The Hard IPs provided include PCIe gen2 IP block, a 400MHz DDR3 SDRAM controller, and variable precision DSP blocks.

In our application the Cyclone V FPGA used is standalone. Its resource count can be seen on Figure 4.1. Our main limitations will be the memory, the register and ALM count and the DSPs used in most of our applications.

Table 6. Maximum Resource Counts for Cyclone V GX Devices

Resource		Member Code				
		C3	C4	C5	C7	C9
Logic Elements (LE) (K)		36	50	77	150	301
ALM		13,460	18,860	29,080	56,480	113,560
Register		53,840	75,440	116,320	225,920	454,240
Memory (Kb)	M10K	1,350	2,500	4,460	6,860	12,200
	MLAB	182	424	424	836	1,717
Variable-precision DSP Block		57	70	150	156	342
18 x 18 Multiplier		114	140	300	312	684
PLL		4	6	6	7	8
3 Gbps Transceiver		3	6	6	9	12
GPIO[4]		208	336	336	480	560
LVDS	Transmitter	52	84	84	120	140
	Receiver	52	84	84	120	140
PCIe Hard IP Block		1	2	2	2	2
Hard Memory Controller		1	2	2	2	2

Figure 4.1: The available resource count in our Cyclone V GX FPGA

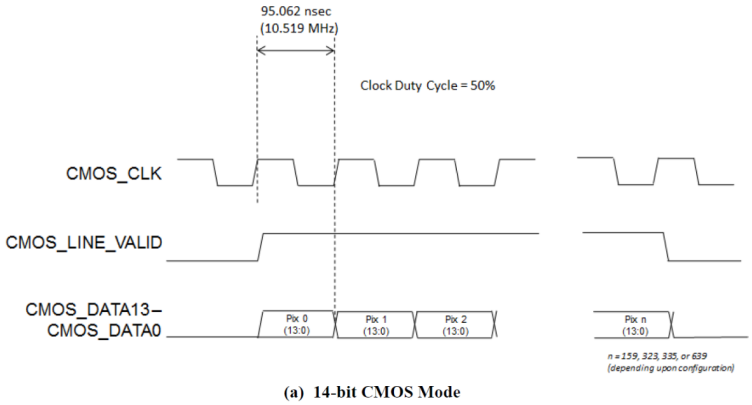


Figure 4.2: The data protocol of the Tau2 camera

4.2 The Flir Tau2 camera

4.2.1 Interface requirements and data protocol

The Tau2 camera has several modes of operation; the one we used for our application was a 3.3V CMOS parallel connection with 14 data bits, a clock, and a line valid and frame valid signal(Figure 4.2). The clock frequency is 21.04 for the camera resolution 640x512 and the frame rate 60Hz; for all other resolutions and framerates it is 10.519 Hz. As the input clock is relatively slow and the data is parallel, the data is ready for use right away.

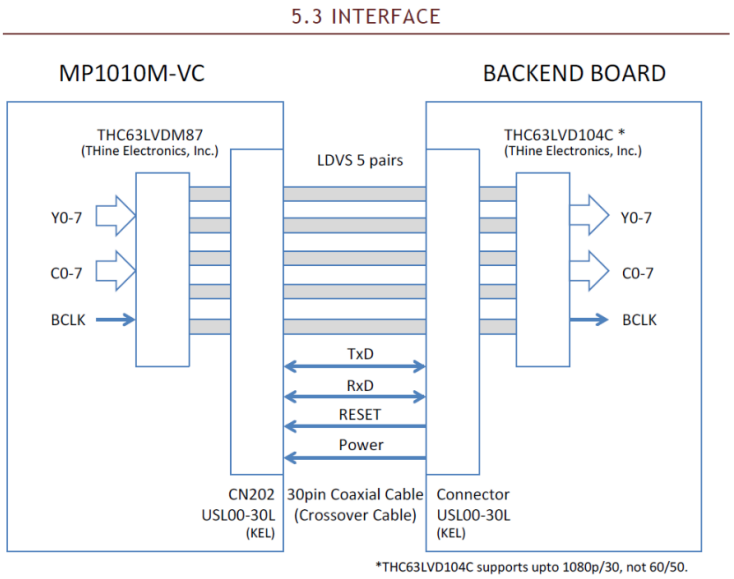


Figure 4.3: The hardware interface of the Tamron MP1010M-VC camera

4.3 The Tamron MP1010M-VC Ultra-Small Camera Module

4.3.1 Interface requirements

The Tamron MP1010M-VC camera[Tam] uses a 2.5V, 4+1 LVDS pair interface with differential termination to transmit its video data(see Figure 4.3). The pixel clock is 74.25 MHz, and 7 bits are sent on a single clock cycle; as such the data rate is 519.75 Mhz. For transmitting and receiving control signals it uses serial communication and the VISCA protocol. The commands required for controlling the camera and using its functions are described in detail in the technical manual[Tam].

4.3.2 Data protocol

The camera uses the Bt.1120 YCbCr data protocol, and separates the sent bytes 1-7: 7 bits sent on one lane and 1 on the other - the other 6 bits are used for synchronization(see Figure 4.4). To use the camera data in the FPGA an LVDS deserializer is needed - we will explore its implementation in the section Deserialization of the Tamron camera signal.

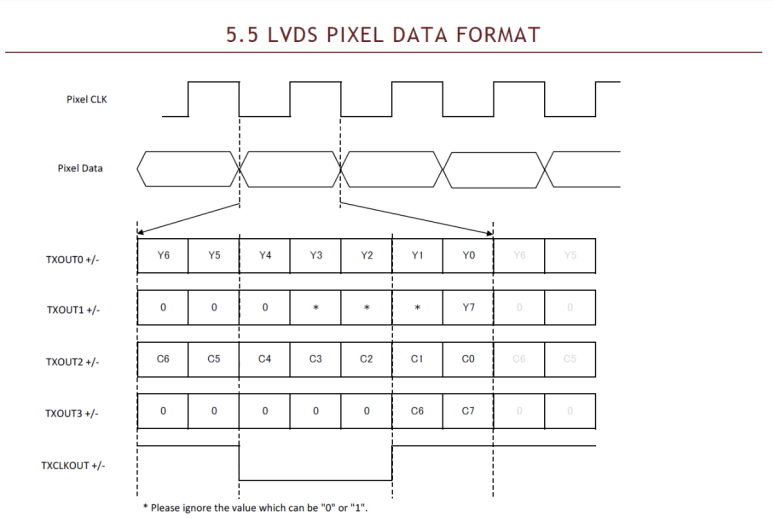


Figure 4.4: The data protocol of the Tamron MP1010M-VC camera

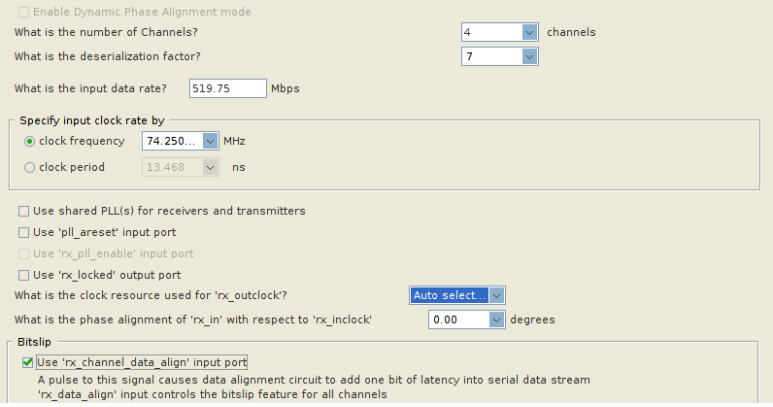


Figure 4.5: The software setup required for correct deserialization of the camera data

4.4 Deserialization of the Tamron camera signal

4.4.1 Hardware requirements and software setup

Cyclone V FPGAs provide a built-in hard LVDS deserializer module for applications like our camera input. The hard SERDES module needs 2.5V LVDS inputs, and internal differential termination to work properly; also, the relative phase shift of the data, the clock frequency, data rate and the deserialization factor has to be set in the Quartus software(as seen on Figure 4.5).

Figure 4.6: A picture taken with the Tamron camera from the balcony of the Workswell s.r.o. office on Albrechtův vrch

Also, for the bits to be in correct order, we need the bitslip functionality of the deserializer, to shift the output until it is aligned. According to the Tamron protocol, I designed a small piece of logic sends a signal to the bitslip pin of the deserializer that shifts the alignment by one place - this should over time ensure that the output is aligned as intended.

After setting up the deserializer and doing a basic check on its functionality, I have created a small module that decodes when the frame start and line start happens according to the protocol Bt.1120. I have then expanded this module to be able to write into the DDR3 memory connected to the Cyclone FPGA; thus I was able to store images into that DDR3 memory. I then have used the already functioning PCIe setup to read out the stored images; an example image I have taken can be seen on Figure 4.6.

As the data is read to the QtCreator software and then the image is generated from that, writing a piece of code that checks the image pixel by pixel is straightforward, as we can random access all pixels. The approach I took for this problem was to use the FREEZE functionality[Tam] of the camera, which makes it send the same image over and over again and then compare the pixels of the first two images. The test was successful - all pixels in the image sent from the camera were the same.

Chapter 5

Limitations, precision and portability of object-recognition algorithms

5.1 The training and comparison dataset

To find the most suitable image classification/detection algorithm that can be effectively implemented in our FPGA, first of all I needed a dataset to compare their effectiveness. My task was to implement a classifier which can classify objects having the feature complexity of human silhouettes, cars, car logos. I have found that human classification is the harder problem from the three, thus I have worked with a dataset of human figures. I have chosen the INRIA dataset[INR], because I found it to be the most well-known and used dataset of pedestrians. I cropped the human figures in the dataset to be 64x32 pixels and transformed them into greyscale, as the Y component of the YCbCr image format holds most of the information about the content of the picture, and thus the input data width to the algorithms can be halved.

As most algorithms cannot be entirely parallelized and storing a 64x1920 pixel buffer in the FPGA would be highly inefficient, I have decided to store the picture of the camera in the DDR3 RAM and then buffer small subpictures to the FPGA, as described in [YL06] and seen on Figure 5.1. As such, some parallelism can be used and many pixels can be reused, but it is not needed to store the whole width of the image. This, however, comes with a cost - the throughput from the external memory needs to be larger. For a buffer window of 96x192 the throughput increases 1.17-fold; this number can be reduced by using a larger window, which in turn comes with larger on-chip

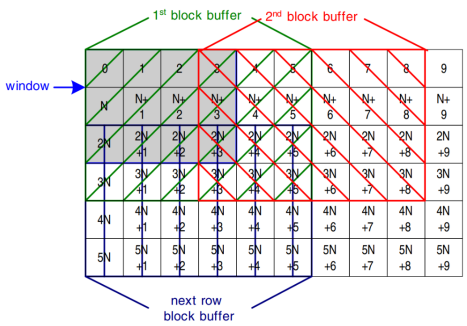


Figure 5.1: The sliding window scheme used in our application, picture from [YL06]

memory use.

After some research I have decided to try out and measure the effectiveness of 3 machine learning algorithms for the task:

1. The Viola-Jones algorithm because of its success on face recognition and low computational cost
2. Support Vector Machines with Histogram of Oriented Gradients, because of their success on pedestrian detection and low computational cost
3. Neural Networks, because of their late success on image classification problems

5.2 Viola-Jones algorithm using Haar-features and Adaboost

One of the first highly successful methods for object classification and detection has emerged in 2001 by Viola, P. and Jones, M. [VJ01]. It used the Adaboost algorithm combined with using Haar-like features. Haar-features, as described in [VJ01] are created by summing the magnitudes of pixels in rectangular areas in certain areas of the input image. Different areas of the image are summed(highlighted in black or white respectively, as seen on as seen on Figure 5.2), the feature is finally constructed by subtracting one given group(e.g. white) from the other. The process is made easier by constructing the incremental image. Its basis lies in the principle, that the value of the pixel of the incr.im. at coordinates x,y is the sum of all pixel values at coordinates

$0 \to x, 0 \to y$ from the input image. This reduces the summation/subtraction of large areas in the picture into just a few operations.

I have used the OpenCV[ope] implementation of the Viola-Jones algorithm to try and measure the precision of the algorithm. The implementation uses Haar-features, as they were the least computationally expensive variant of the algorithm. The lowest error rate I was able to achieve was 12-13%. The Adaboost algorithm selected 95 features and created a tree from them - each Haar feature being defined by its type and bounding boxes.

The main problem I stumbled into when trying to come up with and FPGA implementation for the classifier was that the locality and types of the features weren't following any exact patterns - as such, when trying to implement how the incremental image would be stored, features would be accessing the address space randomly. One way to deal with this would be to not implement the evaluation of the features in a parallel manner, thus slowing down the process 95-fold. A 95-fold slowdown is unacceptable for us, as we cannot compensate with lifting the frequency to those speeds(the maximum increase is about 4x), and we would have to implement parallel storages to have faster performance, which would not be feasible either. The other way would be to store the incremental image in registers, which wouldn't be advisable either - merely storing an image of the size 32*64 with a bitwidth of 16 would use up 32000 registers. If we used a larger window for our sliding window application, as described in section The training and comparison dataset this number would rise to approx. 295000 registers - and that doesn't even remotely fit into the 75,440 registers that our FPGA provides.

Thus, the classifier would not be practical to implement in our particular FPGA. In the last few years the method has been pushed to its limits, and lately surpassed by several others on a task - there is not likely to be any major brakthrough in this field anytime soon. Also, the idea of classifier hierarchy is not too advantageous for FPGA implementation, as hierarchical computations that depend on each other cannot be parallelized. This would only benefit the first implementation, but even if the average processing time is halved, the slowdown would still be 42-fold which is still unacceptable.

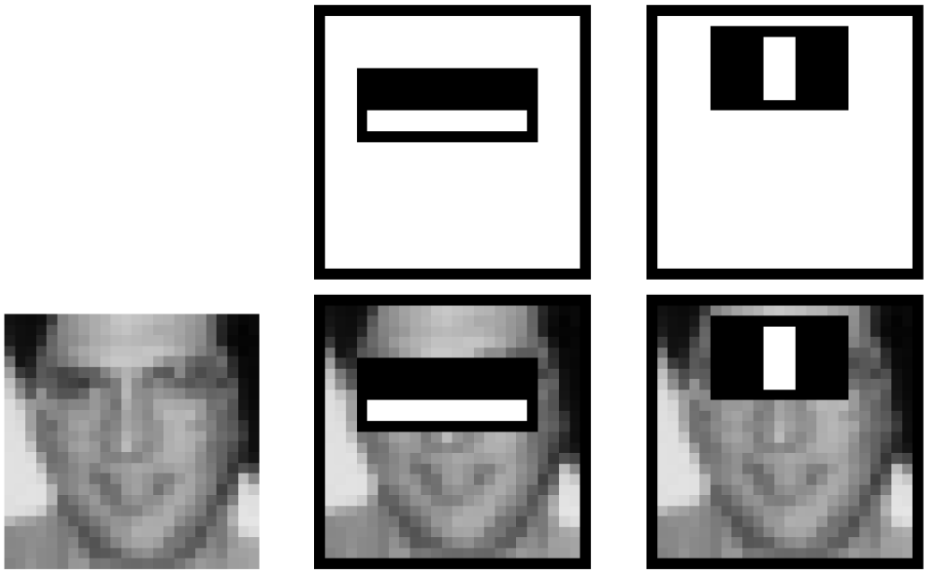


Figure 5.2: Example of a Haar-feature for face recognition

5.3 Support Vector Machines with Histogram of Oriented Gradients

Histograms of Oriented Gradients(HOGs) combined with Support Vector Machines(SVMs) have first been used for pedestrian detection by Dalal, N. and Triggs, B. [DT05]. They combined the feature complexity of HOGs with the low computational cost of SVMs to form a fast and precise classifier.

The concept of HOGs, although known long before that, has first been used by Freeman, W. and Roth, M. for hand gesture recognition [FFRR94]. The basis of the method rests on detecting edges and evaluating their directions in a small portion of the picture. The gradients are then sorted into bins and the population of these bins form the feature set that the classifier works with.

To try out, test and measure the precision of this method, I have used an OpenCV[ope] implementation. I trained the classifier on my modified INRIA dataset several times; the best results I got had a 11-12% error rate. Even though the computation to histograms of oriented gradients is fairly complicated(we need square root and tan operations, both fairly heavy in FPGAs), it could in theory still be done. The gradients are only calculated from a small, e.g. 8x8 portion of the input image, each having 8 bins in the

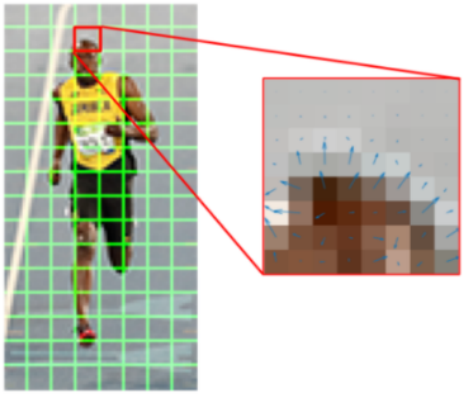


Figure 5.3: Oriented gradients in a small portion of the input image

end, thus for a 64x32 input image we would have 256 bins.

All in all, if we implemented the computation of histograms in some efficient way, implementing HOG SVMs in the FPGA would be a doable solution.

5.4 Neural Networks

Neural Networks provide the state-of-art solutions for image classification applications, and have held this feat since 2012. They outperform other methods by a large margin - their only weakness are their high memory requirements and computational complexity(see Limitations and proposed solutions). This makes their base versions unsuitable for embedded applications. However, the promise of fitting this high-precision technology into embedded devices has driven much research into this field, and recently it is possible to have small, low-bitcount neural networks with fast inference. Moreover, the rise of convolutional neural networks makes even forward passes on a neural network highly parallelizable, which enables their implementation in GPUs and FPGAs.

I have tried a full precision, small network(three convolutional layers each having 32 nodes, followed by a fully connected layer having 32 nodes, then softmax) with a tensorflow implementation to get a feel for strong the improvement would be over traditional methods and how much the error would rise when decreasing the bit width of the weights. The full network achieved an error rate of 2-3%, which was astounding compared to the other methods; I really felt compelled to try the task on more compact networks and test their efficiency.

I have tried a Theano implementation of the Xnornet(see Binary neural networks and Xnornet). This version of binary neural networks has offered low computational cost by simplyfying the convolutions to xnor + bitcount operations, low memory requirements by only needing binary weights, reasonably high performance and continous research going on in its area. Thus it was both effective and had high chance of new methods emerging making it more effective/precise in the near future. I have measured the classification error of a really small network that could reasonably fit into our FPGA, and got an error rate of 6-7%. Compared to the performance to the full precision network, this result is dismal, but it still outperforms the other methods by almost twofold. As such, if it can be implemented efficiently into the FPGA, it would definitely be worth implementing.

The model I created combines three convolutional layers with maxpooling and normalization. Maxpooling makes the dimensionality of the input image smaller, thus reducing the load on the fully connected layer. The weights would not be implemented in memory, and the buffers needed are small(approx. 2400 bytes needed per convolutional buffer), as such implementing binary neural networks seems like a viable solution.

5.5 Conclusion

Because of the benefit in precision, lower operation complexity and overarching research support, I have chosen to implement binary neural networks as the most viable solution for our FPGA. The memory requirements are fairly low(approx. 30kB), and the LUT requirements are really hard to estimate, but with a reasonable guess the classifier should fit into our FPGA. In case we have problems with space, we will work to reduce the size of problematic modules according to their resource usage.

Chapter 6

Implementing object recognition in FPGA

6.1 Proposed system architecture

As mentioned in the Conclusion section of choosing the right algorithm to implement, I have chosen binary neural networks to implement in the FPGA. The architecture of the whole system would be as seen on Figure 6.1. The data from the Tamron camera enters the FPGA, then some input processing takes place(deserialization, etc.), then if needed the resolution can be reduced to enable larger sliding windows or lower memory bandwidth requirements. Only the greyscale component of the camera signal is sent to the external memory, where the entire image is stored.

This image is read by the input module of the neural network, stored in multiple resolutions for multiple sliding window sizes and then sent to the neural network for processing. The neural network operates on a higher frequency than the reading from the memory to allow multiple sliding window sizes. It evaluates the input data and outputs the coordinates of the processed subimage, its scale and the classification result. These results are then filtered, to group windows that are practically the same into a single classification and reduce the number of false positives. In the end, the filtered results are sent to an outside CPU for processing or otherwise used.

It is important to mention that both this system architecture and the convolutional network architecture are completely independent of the exact camera used, the only limitations would be in the size of the detection

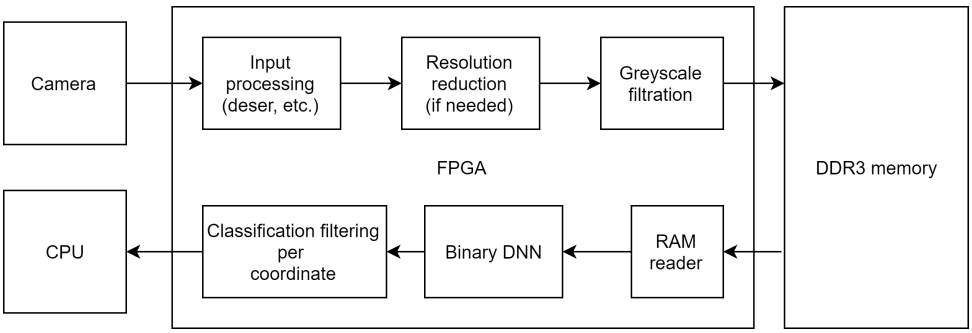


Figure 6.1: The architecture of the FPGA system

region. Thus, we could use this system to with a thermal, or other camera on a different object and it would function just as well if we had a large enough dataset to train the classifier. This makes this single implementation immensely powerful and opens up a plethora of possibilities for future use.

6.2 Convolutional network architecture

Through experimentation and size estimations I have chosen the following architecture for the binary neural network, as seen on Figure 6.2. There are three 3x3, depth 8 convolutional layers, each followed by a batch normalization layer and a maxpool layer. The detailed functionality of these layers is further described in the neural network theory section Types of layers. Global features are then evaluated with a binary fully connected layer. It is followed by a 2-node fully connected layer - the last in the network. This one is full precision and the outputs of these nodes are proportional to the probabilities of the given classes - e.g. the largest number represents the most probable class.

The reasons to implement these layers and in this architecture are the following: the convolutional layers evaluate and construct the features of the input image; the normalization layers ensure the regularization of the samples, normalizing too high or too low activations and thus enabling a higher learning rate; the maxpool layers reduce the number of required computations and enable a higher window of influence for successive convolutional layers; and finally the fully connected layers evaluate the features of the convolutional layers based on their locality.

The main benefit of this architecture is the resolution reduction provided by the maxpool layers - this way, we only need to store an 8x4 size input image of depth 8 for the fully connected layer instead of having to store 64x32. This

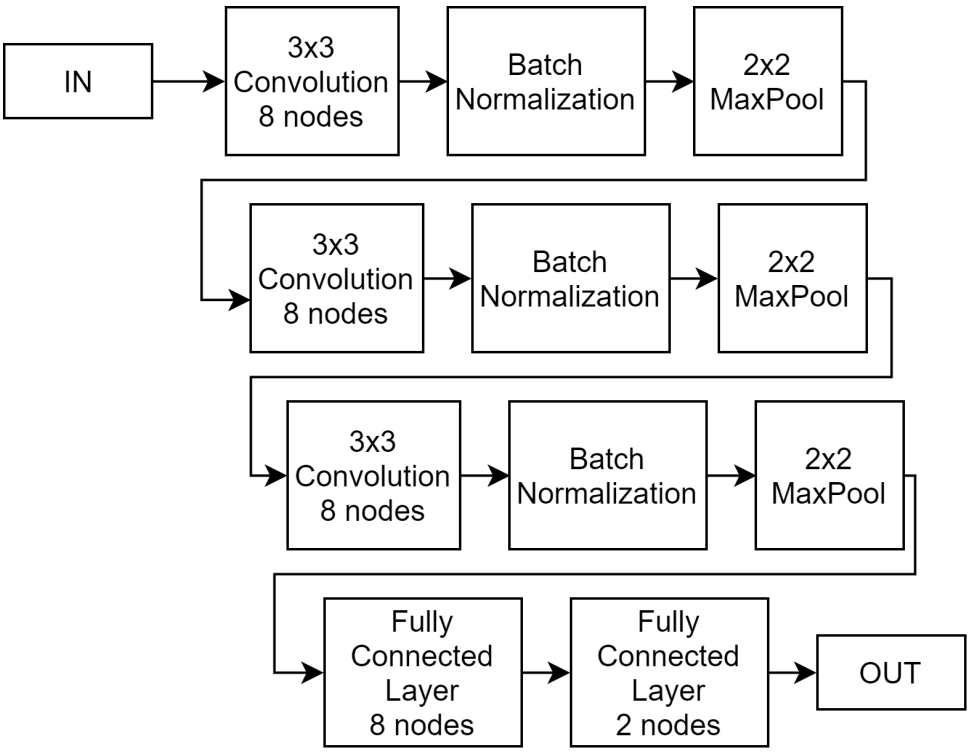


Figure 6.2: The architecture of the implemented binary convolutional network

is a huge difference in memory usage and computational complexity, reducing a 16384-width bitcount operation to 256. The other benefit of maxpool operations is that they reduce the number of operations needed further in the network to 25% - this means that after three maxpool layers the number of required operations drops to 1/256th compared to the original. We can use this to have either deeper convolutional layers or serialize computations to preserve FPGA resources, ideally both.

An other benefit of the architecture is that local information is processed in the convolutional layers and propagated through the network; the consequence of repeated maxpool and convolutional layers is that the area of influence of the features increases significantly. Thus, even though the information that arrives at the fully connected layer is low resolution, it is the result of the combination of fine-grained computations on the image. Thus it can represent complex features while remaining low resolution.

An added benefit is that when the network is put into use, the batch normalization layer transforms into a simple scale and addition operation with constant(learned) parameters. Thus we can merge it with the activations of the convolutional layer preceding it, saving hardware resources.

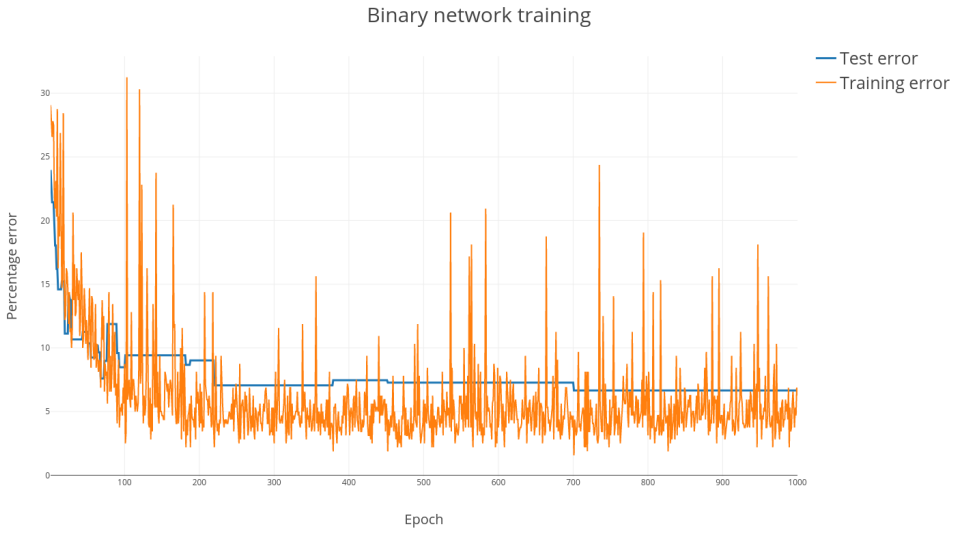


Figure 6.3: The learning process of the binary convolutional network

6.3 Precision in software and in FPGA hardware

As described in the thesis assignment I have first trained the network in software and then used the trained network for the FPGA implementation. I have chosen Theano as the machine learning platform to evaluate binary neural networks on, because it's hardware-independent; as such I could try the project out even when my GPU drivers weren't set up correctly. The project I used as the basis of my implementation[xno] has been evaluated on the MNIST and CIFAR-10 datasets with good results(3.2% and 13.8% error rates). Compared to the state of the art(0.21% and 3.47% error respectively) the precision loss is quite large - however, we cannot expect near state-of-the-art performance from such a compact network.

After some trial-and-error optimization I trained the neural network with batch size 64, initial and final learning rate 0.003 and 0.0000003 respectively, most effectively for 1000 epochs. The epochs took approx. 2 seconds each on an NVIDIA GeForce 970m, overall 33 minutes. The lowest error I was able to achieve was 6-7%, Figure 6.3 shows the learning process of the network.

After implemented in hardware, the error can rise somewhat, as the FPGA network uses fixed point computations, while the Theano implementation uses floating point operations. Unfortunately, even though the classifier system has been fully implemented, I did not manage to implement an automatic testing system in time for exactly measuring the error - this has to be implemented in the future.

Figure 6.4: Examples of true and false classifications from the INRIA dataset. We can see that the network is overexcited over high contrast vertical edges. This can be counteracted by expanding training examples of this type in the training set.

6.3.1 Robustness and performance on other datasets

I have tested the trained network on an other dataset than INRIA too, the DaimlerChrysler pedestrian dataset, achieving 15.66% error rate. The error rate is high compared to the 6-7% on the INRIA dataset and can be improved upon in multiple ways. The most obvious one would be to use a larger training dataset. This can be achieved through just simply finding and preparing a dataset with more examples, finding a dataset of commonly falsely classified examples or expanding the existing dataset using its own samples using rotations and other operations. This would improve performance significantly.

6.4 Hardware footprint

As a first step, I have implemented the network in a completely parallel manner. This means that if a convolutional layer has a depth of larger than one, these layers are computed in parallel, and all computations of neurons in fully connected layers are computed in parallel. This setup is essentially wasting resources, because it is not utilizing the reduction of computations discussed in the section Convolutional network architecture in any manner; however, it is the simplest implementation of the network.

The hardware footprint estimate on our network SimpleXnorNet given

Analysis & Synthesis Resource Utilization by Entity

<<Filter>>

	Compilation Hierarchy Node	Entity Name	Combinational ALUTs	Dedicated Logic Registers	Block Memory Bits	DSP Blocks
1	|SimpleXNORNet	SimpleXNORNet	35584 (99)	26452 (25)	430017	100
1	|fclayer:fc1|	fclayer	10278 (327)	6320 (46)	2726	8
2	|convLayer:c2|	convLayer	7206 (444)	3475 (400)	164306	4
3	|convLayer:c3|	convLayer	7027 (444)	3514 (400)	164300	8
4	|convLayer:c1|	convLayer	2898 (0)	1211 (16)	20534	0
5	|fullwidthFC:fullwidthFC_i|	fullwidthFC	1716 (1618)	748 (748)	0	32
6	|multiRowBuf:cb2|	multiRowBuf	1263 (1249)	2442 (2434)	18520	0
7	|multiRowBuf:cb3|	multiRowBuf	1262 (1248)	2442 (2434)	9304	0
8	|normalize:norm1|	normalize	824 (648)	384 (256)	0	16
9	|normalize:norm2|	normalize	802 (618)	384 (256)	0	16
10	|normalize:norm3|	normalize	799 (615)	384 (256)	0	16
11	|maxPool:mp2|	maxPool	384 (351)	1577 (408)	12807	0
12	|maxPool:mp1|	maxPool	366 (337)	1580 (410)	24786	0
13	|maxPool:mp3|	maxPool	351 (317)	1573 (405)	6656	0
14	|multiRowBuf:cb1|	multiRowBuf	194 (180)	321 (313)	4696	0

Figure 6.5: The detailed resource usage of the completely parallel implementation

by the Quartus Prime Lite software can be seen on Figure 6.5. The overall ALUT(Adaptive LookUp Table) usage is 35584 ALUTs, which translates to 21188 ALM(Adaptive Logic Modules). Our FPGA has 18860 ALMs - thus, the completely parallel design does not fit into our device.

If we inspect the issue in detail, we can see that most of the resources are used by the fully connected layer and the convolutional layers that have an input depth larger than one - the 2nd and 3rd layers. If we further inspect the convolutional layers(Figure 6.6), we can see that most of the logic utilization comes from the parallel implementations of the binarization and activation modules; some come from the convolutions themselves. The same is true for the fully connected layer: the large logic usage is the result of all the neurons having their computations implemented in parallel.

This property, combined with the computation reduction discussed in section Convolutional network architecture allows us to reduce the last two convolutional layers and the fully connected layer to approx. 1/8th of their current area(1/4th for the second layer, as it is preceded by only one maxpool layer). Moreover, as discussed in the same section, we can do away with the normalization layers completely, saving approx. 2400 ALUTs and 1000 registers.

Thus, our overall area savings using these two simple techniques can be at least 20614 ALUTs: the logic usage would thus be reduced to 42.1% of the fully parallel implementation, which means it would use 8913 ALMs, 47.2% of the resources of the whole FPGA. If needed, the resource utilization can then be further reduced by reducing the bitwidth of operations or carefully reimplementing problematic modules. These have the capacity to drive up classification error or development time, but can be a trade worth having.

Analysis & Synthesis Resource Utilization by Entity

<<Filter>>

	Compilation Hierarchy Node	Entity Name	Combinational ALUTs	Compilation Hierarchy Node	Entity Name	Combinational ALUTs
1	|SimpleXNORNet	SimpleXNORNet	35584 (99)			
1	|fclayer:fc1|	fclayer	10278 (327)	|convLayer:c3|	convLayer	7027 (444)
1	|addConv:nodes[1].aC|	addConv	935 (2)	|binarizeInput:binarize[7].bI|	binarizeInput	476 (239)
2	|binarizedBuffer:bB_i|	binarizedBuffer	909 (13)	|binarizeInput:binarize[0].bI|	binarizeInput	463 (239)
3	|addConv:nodes[0].aC|	addConv	865 (2)	|binarizeInput:binarize[1].bI|	binarizeInput	463 (239)
4	|addConv:nodes[3].aC|	addConv	811 (2)	|binarizeInput:binarize[2].bI|	binarizeInput	463 (239)
5	|addConv:nodes[7].aC|	addConv	762 (2)	|binarizeInput:binarize[3].bI|	binarizeInput	463 (239)
6	|addConv:nodes[4].aC|	addConv	760 (2)	|binarizeInput:binarize[4].bI|	binarizeInput	463 (239)
7	|addConv:nodes[6].aC|	addConv	757 (2)	|binarizeInput:binarize[5].bI|	binarizeInput	463 (239)
8	|addConv:nodes[2].aC|	addConv	735 (2)	|binarizeInput:binarize[6].bI|	binarizeInput	463 (239)
9	|addConv:nodes[5].aC|	addConv	665 (2)	|activation:convolve[2].act|	activation	302 (63)
10	|activation:nodes[0].act|	activation	320 (35)	|activation:convolve[3].act|	activation	302 (63)
11	|activation:nodes[1].act|	activation	320 (35)	|activation:convolve[6].act|	activation	302 (63)
12	|activation:nodes[2].act|	activation	320 (35)	|activation:convolve[7].act|	activation	302 (63)
13	|activation:nodes[3].act|	activation	320 (35)	|activation:convolve[4].act|	activation	300 (61)
14	|activation:nodes[4].act|	activation	320 (35)	|activation:convolve[5].act|	activation	299 (60)
15	|activation:nodes[5].act|	activation	320 (35)	|activation:convolve[0].act|	activation	272 (33)
16	|activation:nodes[6].act|	activation	320 (35)	|activation:convolve[1].act|	activation	272 (33)
17	|activation:nodes[7].act|	activation	320 (35)	|binConv:convolv...epth[0].binConv|	binConv	13 (13)
18	|calcBeta:calcbeta_i|	calcBeta	124 (85)	|binConv:convolv...epth[1].binConv|	binConv	13 (13)
19	|ReLU:nodes[7].relu1|	ReLU	18 (18)	|binConv:convolv...epth[2].binConv|	binConv	13 (13)
20	|binarizeFC:binFC|	binarizeFC	8 (8)	|binConv:convolv...epth[3].binConv|	binConv	13 (13)
21	|ReLU:nodes[0].relu1|	ReLU	6 (6)	|binConv:convolv...epth[4].binConv|	binConv	13 (13)
22	|ReLU:nodes[1].relu1|	ReLU	6 (6)	|binConv:convolv...epth[5].binConv|	binConv	13 (13)
23	|ReLU:nodes[2].relu1|	ReLU	6 (6)	|binConv:convolv...epth[6].binConv|	binConv	13 (13)
24	|ReLU:nodes[3].relu1|	ReLU	6 (6)	|binConv:convolv...epth[7].binConv|	binConv	13 (13)
25	|ReLU:nodes[4].relu1|	ReLU	6 (6)	|binConv:convolv...epth[6].binConv|	binConv	12 (12)
26	|ReLU:nodes[5].relu1|	ReLU	6 (6)	|binConv:convolv...epth[3].binConv|	binConv	11 (11)
27	|ReLU:nodes[6].relu1|	ReLU	6 (6)	|binConv:convolv...epth[4].binConv|	binConv	10 (10)

Figure 6.6: The detailed resource usage of the convolutional and fully connected layers

Chapter 7

Conclusion

As stated in the Introduction and premise, the overarching goal of the thesis was to create the basis for a real-time object detection system based on a FHD camera and a low-cost FPGA. As the scope of this project is quite large, the work done in this thesis and specified in the assignment is only a part of it - albeit the most significant part.

In the assignment the problem was dissected into multiple parts:

1. Choosing the right algorithm for our image processing needs (chapter 5)
2. Getting the camera signal to the FPGA (section 4.3)
3. Processing it and verifying its integrity (section 4.4)
4. Training in software and contructing the classifier in FPGA SystemVerilog code (chapter 6)

The only part where I did not meet the requirements of the assignment was trying multiple kinds of objects. I have chosen not to focus on this aspect as it seemed much more important to have a stable, working, efficient and parametrizable classifier first. After this is done, trying multiple types of objects is child's play.

As we have previously stated however, we need to add some extensions to the work implemented as of the assignment to have a fully functional,

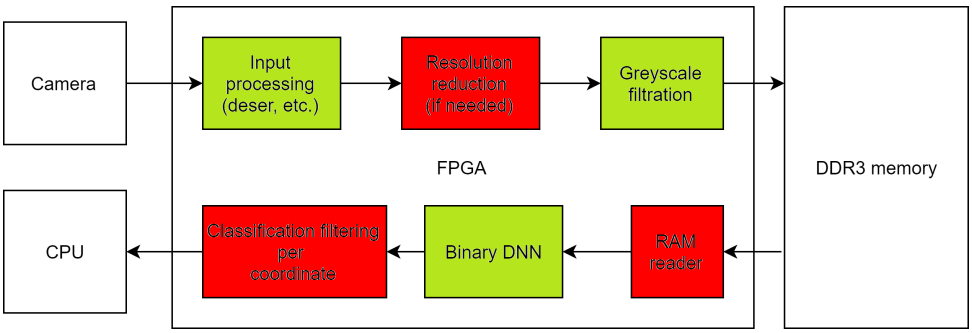


Figure 7.1: The state of implementation of the image processing system

realistically usable system. The current state of these components is shown on Figure 7.1: completed parts are shown with green, parts still to implement are shown in red. The following sections summarize the current state of the system and give a clearer picture of the remaining tasks needed to construct a fully functional system.

7.1 Achieved results

7.1.1 Setting up and testing the Tamron input interface and deserialization

The setup of the Tamron input interface is further described in section 4.4. The results of it are the following:

1. The termination of the inputs and the hard deserializer in the FPGA are set up correctly so the camera data is stable and correct
2. We can send an image to the external DDR3 memory and read it out for preview and byte-to-byte comparison, which gives us the expected results
3. As the format of the Tamron image is YCbCr, greyscale filtering is trivial: we just use the Y component

This means that the route in the system architecture to the external DDR3 memory is almost ready; the implementation of the resolution reduction is further discussed in section Resolution reduction.

7.1.2 Choosing the architecture, training the binary convolutional network in Theano

The contruction of the neural network architecture was a compromise between precision and used FPGA resources; it is further discussed in section 6.2. The preparation of the training set and the training in software are further discussed in sections 5.1 and 6.3. In this phase of development I have achieved the following feats:

1. Found a suitable neural network architecture which is likely to fit into the FPGA and has reasonable precision
2. Trained the aforementioned network to an error of 6.77%
3. Wrote a python script for converting the trained model weights to synthetizable SystemVerilog code
4. Wrote a python script for transforming the training/test dataset to a format accessible from the ModelSim simulation software

It is quite hard to compare the resulting precision to the state of the art, as there are multiple datasets used in the industry, and the precision is highly dependent on how challenging the dataset itself is. A 2018 study, [GMS+18] has achieved 75% precision, however, on a different dataset.

All in all, the software training and model translation to FPGA code is ready for use on other datasets. Also, if we want to test our model's precision in FPGA code, we can easily do that by transforming our dataset to data digestible by ModelSim.

7.1.3 Implementing and testing the binary neural network classifier in an FPGA

The implementation of the neural network in the FPGA wasn't a straightforward task, but it had the asset of having identical operations as the Theano implementation. This meant that it was step-by-step controllable, thus enabling fast and efficient debugging of the SystemVerilog code. The following layers have been implemented in synthetizable SystemVerilog, from scratch:

1. Convolutional layer as of XnorNet[RORF16]
2. Batch Normalization layer(inference only)
3. Maxpool layer
4. Fully Connected Layer as of XnorNet[RORF16]
5. Fully connected, full precision layer, replacing softmax

The implementation of the modules is parametric by default: the resolution of the sliding window, computational bitwidth and other parameters are adjustible and the implementations are only limited by the FPGA resources.

The current, completely parallel implementation unfortunately **does not fit into our target FPGA**. However, with some additional work, its size can easily be reduced to 42% of its current size, taking up 47% of the FPGA's resources. Future work in this topic is discussed in section 7.2.1.

7.2 Future work

7.2.1 Reducing the hardware footprint

As described in section 6.4, the hardware footprint of the initial system is larger than anticipated, so large in fact that the design does not fit into the target FPGA. However, as further discussed in the section, the problem can quite easily be mended by two, relatively simple optimizations:

1. Merging the normalization layers with the activations in the convolutional layers
2. Serializing computations in convolutional and fully connected layers

Normalization operations can be eliminated because the ReLU activation function is in part linear - thus we can merge the two scales and shifts into one. The serialization of computations is possible because the maxpool layers reduce the dimensionality of the input image and thus the necessary computations. This enables to have the same computational module calculate

the output of several neurons in a single cycle, as the clock slows down 4x by each maxpool layer. We can use the additional computational time to either reduce resources, or implement a larger network with the same resource usage - ideally both.

7.2.2 Resolution reduction

The most basic and useful type of resolution reduction would be to halve the resolution of the image; the basis of such a computation is already implemented in the maxPool module, except we would swap the max() operation to averaging. The resource count of averaging over 4 samples would be negligible, as dividing by 4 in binary is trivial.

More complex resolution methods would also be possible, however we would have to carefully assess the cost of averaging, as the costs could potentially outweigh the benefits of such an implementation.

7.2.3 RAM reader and multisize buffers

After the camera image is stored in RAM, it is not arbitrary in which order it should be extracted and fed into the module - the optimal buffering scheme is discussed in section 5.1. Also, because detecting objects on a single scale has limited use in practice, we could either store a part of the buffered image with a different resolution, or have it read from memory multiple times - the former being the better option, if the buffering windows are sufficiently small. Thus, multiscale buffering still needs to be implemented for the system to be usable.

7.2.4 Classification post-processing.

As discussed in section 6.1, there may be some post-processing needed to manipulate the detection sensitivity and false positive rate of the network. These manipulations highly depend on the particular application and may involve methods used for correlating detections between frames; as such, they are not explored in this thesis.

Appendices

Appendix A

Bibliography

[BGFAO17] Héctor Barreiro, Ignacio García-Fernández, Iván Alduán, and Miguel A. Otaduy, *Conformation constraints for efficient viscoelastic fluid simulation*, 2017, pp. 221.1–221.11.

[cit] *History of FPGAs*, `https://web.archive.org/web/20070412183416/http://filebox.vt.edu/users/tmagin/history.htm`, Accessed: 2007-04-12.

[cyc] *Cyclone V overview*, `https://www.altera.com/products/fpga/cyclone-series/cyclone-v/overview.html`.

[DT05] Navneet Dalal and Bill Triggs, *Histograms of oriented gradients for human detection*, Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1 - Volume 01 (Washington, DC, USA), CVPR '05, IEEE Computer Society, 2005, pp. 886–893.

[End01] Herbert Enderton, *A mathematical introduction to logic (2nd ed.)*, Boston, MA: Academic Press, 2001.

[ESTA13] Dumitru Erhan, Christian Szegedy, Alexander Toshev, and Dragomir Anguelov, *Scalable object detection using deep neural networks*, CoRR **abs/1312.2249** (2013).

[FFRR94] William T. Freeman, William T. Freeman, Michal Roth, and Michal Roth, *Orientation Histograms for Hand Gesture Recognition*, In International Workshop on Automatic Face and Gesture Recognition, 1994, pp. 296–301.

[FH17] Nicholas Frosst and Geoffrey E. Hinton, *Distilling a neural network into a soft decision tree*, CoRR **abs/1711.09784** (2017).

[GMS+18] Farzin Ghorban, Javier Marín, Yu Su, Alessandro Colombo, and Anton Kummert, *Aggregated channels network for real-time pedestrian detection*, CoRR **abs/1801.00476** (2018).

[GSS15] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy, *Explaining and harnessing adversarial examples*, International Conference on Learning Representations, 2015.

[GTM17] Angus Galloway, Graham W. Taylor, and Medhat Moussa, *Attacking binarized neural networks*, CoRR **abs/1711.00449** (2017).

[HMD15] Song Han, Huizi Mao, and William J. Dally, *Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding*, CoRR **abs/1510.00149** (2015).

[HQ09] Juwayriyah Hussain and Paul Quintana, *Protecting the FPGA Design From Common Threats*, Tech. report, Altera Corporation, 07 2009.

[HZYN18] Qiangui Huang, Qikun Kevin Zhou, Suya You, and Ulrich Neumann, *Learning to prune filters in convolutional neural networks*, CoRR **abs/1801.07365** (2018).

[IKT+17] Andrey Ignatov, Nikolay Kobyshev, Radu Timofte, Kenneth Vanhoey, and Luc Van Gool, *WESPE: weakly supervised photo enhancer for digital cameras*, CoRR **abs/1709.01118** (2017).

[INR] *INRIA dataset*, `http://pascal.inrialpes.fr/data/human/`.

[IS15] Sergey Ioffe and Christian Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, CoRR **abs/1502.03167** (2015).

[JM16] Ondřej Drbohlav Jiří Matas, *Nonparametric methods for density estimation*, November 2016.

[JM17] ———, *Support vector machines*, November 2017.

[Jo17] Jana Kostlivá Ondrej Drbohlav Jan Šochman, Jirí Matas, *Adaboost*, November 2017.

[KMM+17] Simon Kallweit, Thomas Müller, Brian McWilliams, Markus Gross, and Jan Novák, *Deep scattering: Rendering atmospheric clouds with radiance-predicting neural networks*, ACM Trans. Graph. (Proc. of Siggraph Asia) **36** (2017), no. 6.

[McN12] Steven McNeil, *Solving Today's Design Security Concerns*, Tech. report, Xilinx, Inc, 07 2012.

[NVS+17] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, and Guy Boudoukh, *Can fpgas beat gpus in accelerating next-generation deep neural networks?*, Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (New York, NY, USA), FPGA '17, ACM, 2017, pp. 5–14.

[OMS17] Chris Olah, Alexander Mordvintsev, and Ludwig Schubert, *Feature visualization*, Distill (2017), https://distill.pub/2017/feature-visualization.

[ope] *OpenCV library*, `https://github.com/opencv/opencv`.

[PW17] Luis Perez and Jason Wang, *The effectiveness of data augmentation in image classification using deep learning*, CoRR **abs/1712.04621** (2017).

[RDGF15] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi, *You only look once: Unified, real-time object detection*, CoRR **abs/1506.02640** (2015).

[RHGS15] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun, *Faster R-CNN: towards real-time object detection with region proposal networks*, CoRR **abs/1506.01497** (2015).

[RORF16] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi, *Xnor-net: Imagenet classification using binary convolutional neural networks*, CoRR **abs/1603.05279** (2016).

[SPW+17] Jonathan Shen, Ruoming Pang, Ron J. Weiss, Mike Schuster, Navdeep Jaitly, Zongheng Yang, Zhifeng Chen, Yu Zhang, Yuxuan Wang, R. J. Skerry-Ryan, Rif A. Saurous, Yannis Agiomyrgiannakis, and Yonghui Wu, *Natural TTS synthesis by conditioning wavenet on mel spectrogram predictions*, CoRR **abs/1712.05884** (2017).

[sta] *CS231n Convolutional Neural Networks for Visual Recognition*, `http://cs231n.github.io/`, Accessed: 2018.

[SVS17] Jiawei Su, Danilo Vasconcellos Vargas, and Kouichi Sakurai, *One pixel attack for fooling deep neural networks*, CoRR **abs/1710.08864** (2017).

[SWvdH10] Chunhua Shen, Peng Wang, and Anton van den Hengel, *Optimally training a cascade classifier*, CoRR **abs/1008.3742** (2010).

[SZS+13] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus,

Intriguing properties of neural networks, CoRR **abs/1312.6199** (2013).

[Tam] Tamron Europe Gmbh, *Full hd camera module mp1010m-vc technical reference manual*.

[THW17a] Wei Tang, Gang Hua, and Liang Wang, *How to train a compact binary neural network with high accuracy?*, AAAI, 2017.

[THW17b] ———, *How to train a compact binary neural network with high accuracy?*, AAAI, 2017.

[UF12] Habib Mehrez Umer Farooq, Zied Marrakchi, *Tree-based heterogeneous fpga architectures*, Springer-Verlag New York, 2012.

[VH17a] Ondřej Drbohlav Václav Hlaváč, Jiří Matas, *Bayesian decision theory*, October 2017.

[VH17b] ———, *Perceptron classifier, empirical vs structural risk minimization*, November 2017.

[VJ01] Paul Viola and Michael Jones, *Rapid object detection using a boosted cascade of simple features*, 2001, pp. 511–518.

[WLZ+17] Ting-Chun Wang, Ming-Yu Liu, Jun-Yan Zhu, Andrew Tao, Jan Kautz, and Bryan Catanzaro, *High-resolution image synthesis and semantic manipulation with conditional gans*, CoRR **abs/1711.11585** (2017).

[xno] *Theano implementation of the XnorNet*, `https://github.com/gplhegde/theano-xnor-net`.

[YL06] Haiqian Yu and Miriam Leeser, *Automatic sliding window operation optimization for fpga-based*, Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (Washington, DC, USA), FCCM '06, IEEE Computer Society, 2006, pp. 76–88.

Appendix B

Additional materials - CD

The directory structure of the additional materials provided on CD is:

1. The code and data used for the software training and test is in the folder SW_train_test
2. The Quartus project and Verilog/SystemVerilog code for implementing the neural network in FPGA is in the folder FPGA_code
3. The modified INRIA dataset on which the classifier was trained is available for preview in the folder INRIA_modified

```
CD
├── SW_train_test
│   ├── data
│   ├── train
│   │   └── external
│   └── test
├── FPGA_Code
│   ├── SimpleXnorNet
│   │   └── simulation
│   │       └── modelsim
└── INRIA_modified
    ├── train
    │   ├── pos
    │   └── neg
    └── test
        ├── pos
        └── neg
```

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Rózsa Tibor** Personal ID number: **452919**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Measurement**

Study program: **Cybernetics and Robotics**

Branch of study: **Sensors and Instrumentation**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Object Recognition Using an FPGA on an Embedded Platform

Bachelor's thesis title in Czech:

Rozpoznávání objektů v obrazu viditelné kamery pomocí FPGA

Guidelines:

1) Get familiar with algorithms for software-based object recognition and the limitations and portability of given algorithms to FPGAs.
2) Get familiar with the software Quartus, the limitations of the FPGA CycloneV 5CGXFC4C7U19C8, and the data protocols and interface requirements of the cameras Tau2 and Tamron.
3) Prepare and test the LVDS-deserialisation module for the input data(4-lane, 519.75MHz) from the camera Tamron. Verify the deserialisation and transfer reliability by previewing the picture from the camera, and by pixelwise comparison. The input data from the Tamron camera are of the protocol BT.1120, YCbCr.
4) Implement object recognition(human silhouette, cars, car logo/brand) in the picture of the camera Tamron. Implement the algorithm and training in software and then use the algorithm/training data to implement the classifier in the FPGA.
5) Discuss the achieved results.

Bibliography / sources:

[1] Computer Vision: Algorithms and Applications Richard Szeliski
[2] Digital Video Processing for Engineers Suhel Dhanani Michael Parker
[3] FPGA Implementations of Neural Networks Amos R. Omondi Jagath C. Rajapakse
[4] EURASIP Journal on Applied Signal Processing 2005:7, 1047?1061
[5] arXiv:1603.05279v4 [cs.CV] 2 Aug 2016

Name and workplace of bachelor's thesis supervisor:

Ing. Jan Kovář, Workswell s.r.o., Libocká 653/51b, 16100 Praha 6

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **07.12.2017** Deadline for bachelor thesis submission: _____

Assignment valid until:
by the end of summer semester 2018/2019

Ing. Jan Kovář Supervisor's signature	Head of department's signature	prof. Ing. Pavel Ripka, CSc. Dean's signature