1 2 9 0

UNIVERSIDADE Ð
COIMBRA

Rafael Alves Vieira

Yolo v3 Tiny on Reconfigurable Logic for Underwater Environments

Dissertation Submitted in Partial Fulfilment for the Degree of Master's in Electrical and Computer Engineering Supervised by Professor Jorge Nuno de Almeida e Sousa Almada Lobo and presented to the Department of Electrical and Computer Engineering of the Faculty of Sciences and Technology of Coimbra University.

September 2022

Faculty of Sciences and Technology of
Coimbra University

Yolo v3 Tiny on Reconfigurable Logic for Underwater Environments

Rafael Alves Vieira

Dissertation Submitted in Partial Fulfilment for the Degree of Master's in Electrical and Computer Engineering Supervised by Professor Jorge Nuno de Almeida e Sousa Almada Lobo and presented to the Department of Electrical and Computer Engineering of the Faculty of Sciences and Technology of Coimbra University.

September 2022

Acknowledgements

In this text I intend to thank everyone who helped me in this work.

Major thanks to my supervisor, Dr. Jorge Nuno de Almeida e Sousa Almada Lobo for taking interest and supporting me in subjects that prioritized my interest of study.

Many thanks to Dr. Pedro Moreira for the opportunity to start an internship at Synopsys, the freedom to pursue this dissertation topic, all the support provided.

I am grateful to the Institute of Systems and Robotics for providing me with the necessary material for the development of this work.

I want to give my special thanks to three friends and course mates in particular. A big thanks to Nuno Gonçalves Mendes for helping me to assemble a system to measure electrical current from a computer, to Nuno Marques for providing me with a place to stay during the last month of this work, and to José Pedro Azevedo, who was also supervised by the same supervisor, and helped me especially in the final phase of the work with Vitis AI.

Also thanks to my friends, course mates, and my family for giving me moral support.

Abstract

Object detection is an important and frequently applied task in the areas of medicine, security and transport, where the solutions provided by machine learning achieve great results, but are often obtained through heavy computational effort.

To a large extent, the oceans of our planet still remain unexplored. Exploring such a wide environment would be easier and more efficient using remotes devices, or robots. In order to use small robotic devices for object detection, one method is to rely on the remote device to gather data, and then process it inside a cloud computing server. However, in situations where real-time processing of data is required, cloud computing isn't the best path to take, since we will be dealing with a lot of latency, bandwidth shortage, and energy consumption when sending and receiving data. For these kinds of situations edge computing is a better option, meaning that, in the same instant, it can obtain data and process right away. Object detection algorithms are typically run by Graphics Processing Units (GPUs), and using one in a device with limited power can shorten the autonomy by a considerable amount.

In this dissertation, we use one of the most well known CNN arquitectures, YOLO (You Only Look Once), (in this case YOLO v3 Tiny) to detect marine species, comparing both approaches (cloud and edge computing). To represent data processing done by the cloud computing approach we run the detection algorithm on a GPU. As for the edge computing approach we rely on an reconfigurable logic circuit (Field Programmable Gate Array (FPGA)), which allows us to explore trade-offs between power consumption, latency, frames per second, and classification metrics. The effectiveness of a pre-processing that improves the visibility of underwater images is also analyzed and explored: it was found that filtering contributes the most for CNNs as a data augmentation technique, rather than improving detection metrics by using it as a pre-processing algorithm. It also consumes a high amount of energy, most of the times, much more than the detection operation.

To run the detection algorithm in the FPGA, we used three different frameworks: PYNQ, FINN and Vitis-AI. Only PYNQ and Vitis-AI were successfully implemented, due to some limitations of FINN. PYNQ represents an implementation without parallelization or quantization, achieving 1.88 FPS and spending 3.83 Joule per frame. While Vitis-AI runs a quantized and parallelized version, achieving 70.05 FPS and 0.31 Joule per frame. When comparing the results in edge computing with the cloud computing approach, by using a GPU we achieved 246 FPS and 0.55 Joule per frame. Besides Joules per frame, the nominal power as well as the idle operational power needs to be taken in consideration. However, an accurate comparison with a cloud computing approach would require transfer of data between the edge and cloud device.

Keywords: FPGA, Yolo, Object Detection, Edge Computing, Underwater Image.

Resumo

A detecção de objectos é uma tarefa importante e frequentemente aplicada nas áreas da medicina, segurança e transporte, onde as soluções fornecidas por *machine learning* alcançam grandes resultados, mas são frequentemente obtidas através de um elevado esforço computacional.

Os oceanos do nosso planeta ainda permanecem, em larga escala, inexplorados. Explorar um ambiente tão vasto seria mais fácil e eficiente utilizando dispositivos remotos, ou robôs. A fim de utilizar pequenos dispositivos robóticos para detecção de objectos, um dos métodos é usar o dispositivo remoto para recolher dados, e depois processá-los dentro de um servidor de *cloud computing*. No entanto, em situações em que o processamento de dados em tempo real é importante, *cloud computing* não é a melhor solução a seguir, uma vez que iremos lidar com muita latência, falta de largura de banda, e consumo de energia ao enviar e receber dados. Para este tipo de situações a *edge computing* é uma melhor opção, o que significa que, no mesmo instante, podemos obter dados e processá-los de imediato. Os algoritmos de detecção de objectos são tipicamente executados por *Graphics Processing Units* (GPUs), e a sua utilização de um num dispositivo com uma fonte de energia limitada pode encurtar a autonomia numa quantidade considerável.

Nesta dissertação, utilizamos uma das arquitecturas CNN mais conhecidas, YOLO (*You Only Look Once*), (neste caso YOLO v3 Tiny) para detectar espécies marinhas, comparando ambas as abordagens (*cloud e edge computing*). Para representar o processamento de dados feito pela abordagem de *cloud computing*, executamos o algoritmo de detecção numa GPU. Quanto à abordagem de *edge computing*, usamos numa *Field Programmable Logic Gate Array* (FPGA), que nos permite explorar diferentes equilíbrios entre consumo energético, latência, frames por segundo, e métricas de classificação. A eficácia de um pré-processamento que melhore a visibilidade das imagens subaquáticas é também analisada e explorada: verificou-se que a filtragem contribui mais para as CNNs como técnica de *data augmentation*, em vez de melhorar a métrica de detecção utilizando-a como um algoritmo de pré-processamento. Também consome uma grande quantidade de energia, na maioria das vezes, muito mais do que a operação de detecção.

Para executar o algoritmo de detecção na FPGA, utilizámos três frameworks diferentes: PYNQ, FINN e Vitis-AI. Apenas PYNQ e Vitis-AI foram implementadas com sucesso devido a limitiações da FINN. PYNQ representa uma implementação sem paralelização ou quantização, atingindo 1,88 FPS e gastando 3,83 Joule por frame. Enquanto Vitis-AI executa uma versão quantizada e paralelizada, atingindo 70,05 FPS e 0,31 Joule por frame.

Com a GPU foram alcançados 246 FPS e 0,55 Joule por frame. No entanto, uma comparação precisa com uma abordagem de *cloud computing* exigiria a transferência de dados entre os dispositivos de *edge* e *cloud computing*. Também é nessecário de ter em consideração a potência nominal e a potência em períodos de inatividade.

Palavras-chave: FPGA, Yolo, Detecção de objectos, Edge Computing

Contents

List of Figures

List of Tables

List of Acronyms

ANN Artificial Neural Network.

CLAHE Contrast-Limited Adaptive Histogram Equalization.

CNN Convolutional Neural Network.

DPU Deep Learning Processing Unit.

FC Fully Connected.

FPGA Field Programable Gate Array.

GC Gamma Correction.

GPU Graphics Processing Unit.

HLS High Level Synthesis.

IOT Internet of Things.

IOU Intersection Over Union.

IP Intellectual Property.

MAP Mean Average Precision.

MLP Multi-Layer Perceptron.

PAN Path Aggregation Network.

QNN Quantized Neural Network.

RGHS Relative Global Histogram Stretching.

ROWS Removal of Water Scattering.

SAM Spatial Attention Module.

SPP Spatial Pyramid Pooling.

YOLO You Only Look Once.

Chapter 1

Introduction

1.1 Problem Background

Not too long ago, tasks that were dependant on human interaction, such as the analysis of x-ray images, or overall environmental data acquisition, have become more automated, since computer vision algorithms greatly facilitate these types of operations and partially replace human interaction. Even more recently, the tasks carried out in the computer vision field are migrating to a new approach, which uses an analogy based on data instead of the traditional geometry based methods where concepts like shape, pose, disparity, motion, optical flow, depth, volume and others that describe the structure of the world, were an essential part of the solution for computer vision problems. The new data-driven technique makes use of contemporary Deep Neural Networks and more specifically Convolutional Neural Networks which keep providing more interesting results. There are also other tasks that have only been made possible with the development of recent technology, such as the exploration of undiscovered environments like Mars or our planet's oceans, using remotely operated robots that require an efficient balance between autonomy and performance.

Computer vision has always been limited by the hardware it operates on, and balancing it with efficient energy consumption can be an exciting challenge. For instance, the company EvoLogics has an autonomous underwater bionic vehicle called *BOSS - Manta Ray* [1] built for underwater exploration and monitoring. This robot currently uses a GPU for the arithmetic calculations required by object detection tasks. The use of a GPU is questionable in this situation, as in some cases it can be replaced with more energy efficient hardware.

1.2 Motivation and Scope

The autonomy of a robot is, of course, very important (low autonomy means more human interaction). So, ideally, we want as much autonomy as possible, and it becomes important to pay attention to energy consumption and analyze trade-offs between the quality of the results and the energy consumed by these devices. In this dissertation the possibility and feasibility, benefits and disadvantages of commissioning an FPGA with the arithmetic behind object detection is analyzed and tested. It should be noted that the vast majority of systems that use a GPU also use a CPU and Memory together, and both of their consumptions will be taken into account. Like all hardware, the work done by the

CPU, GPU, Memory and the like, can be combined and/or simulated on an FPGA by customizing the hardware according to the resources available, and exploring whatever trade-off there is between resources used and results obtained. An FPGA is reconfigurable hardware that is usually programmed using Hardware Description Languages (HDLs). However, since developing Convolutional Neural Networks (CNNs) using HDLs to explore these metrics is a very extensive job, we decided to work around the problem using frameworks like Python Productivity for Zynq (PYNQ) [2]. This raises the abstraction level a bit and limits hardware customization to some extent but doesn't limit as much the possibilities of what there is to be explored.

Underwater images are usually hindered by radiance propagation. The captured images in underwater environments are affected by a blue, or some times, more greenish tint. There are many different types filtering algorithms that make underwater images more pleasant for human perception, and because humans may help in the classification of marine species, the studied CNN, in this case YOLOv3-Tiny, may benefit from a pre-processing filtering of the input data.

1.3 Objectives

The main goal is to promote an energy-saving solution for the use of an accurate object detection algorithm in small underwater robots.

To accomplish this goal, the following steps were set out:

1. Find a good architecture to use on FPGA which brings good classification metrics and doesn't require extensive hardware resources to operate at a good frame rate.
2. Explore various image processing algorithms to improve visibility in underwater images.
3. Explore the influence and benefits that filtering brings to the chosen architecture and dataset, and possibility of inserting filtering as a pre-processing for CNN inputs, or post-processing to use only as a visibility enhancement for human perception.
4. Find a good way to run the chosen architecture on the FPGA without having to use HDLs, as there are many open-source tools to help increase the level of abstraction for FPGA programmers.
5. Measure and compare power consumptions, accuracies, quantizations, and frame rates on the FPGA and GPU, including or excluding filtering, drawing conclusions.

1.4 Related Work

1.4.1 Underwater object detection

The use of CNNs in FPGAs for underwater environments is addressed in [3], but uses CNNs only for obstacle detection, like rocks, or plants, and not for marine life monitoring as we intend.

The work described in [4] accelerates a ZYNQ processor for fish detection in underwater images, which is close to our goal, but uses MobileNet for detection and UNet for underwater image enhancement. Using an encoder/decoder like UNet as a CNN preprocessing is the same as using extra layers that focus purely on image enhancement.

To train a CNN that improves the visibility of images, it is necessary to have ideal images, i.e. images just like the original input but with good visibility, possibly filtered by image processing algorithms. However, UNet is a different case, since it requires a dataset with mask annotations.

In spite of UNet being famous for it superior results in this matter, we shouldn't have much problems relying on traditional image enhancement algorithms, since they tend to be approximated by the encoder/decoder algorithm. In the work [4] it wasn't possible to achieve better energy efficiency than a GPU, but it was still better than a CPU.

A reduced version of YOLOv3 for underwater environments is presented in [5]. It is a good architecture to use on FPGAs and other devices alike. It can be considered as a good architecture to use in our work as it greatly reduces the number of FLOPs with little reduction in other metrics. However, this architecture was not used but kept in mind for future work, since there is more documentation available for vanilla YOLO architectures.

The studies conducted in [6] and [7] compare the performance of some versions of the YOLO architecture on the same dataset that we will be using in this dissertation. They also propose their version of Tiny YOLO (based on version 4), so the work described is used as a comparison to our results when it comes to model accuracy and speed.

1.4.2 Framework related comparisons

There are several different frameworks for Intelecutal Property (IP, or block of logic data) generation in FPGAs. These frameworks are often used to generate accelerators. Sometimes, the accelerator is already given, and needs to be programmed, as it is the case for VitisAI's Deep Learning Processing Unit used in [8], or the NVIDIA Deep Learning Accelerator (NVDLA) used in works like [9].

The most recently used tools for this matter are VitisAI and FINN, developed by Xilinx. FINN operates on top of PYNQ, a framework to program the zynq7000 processor present in most of Xilinx boards, with python language. A comparison between these frameworks is discussed in [8], where it is stated that VitisAI performs slightly better than FINN, but making custom accelerators outperforms the use of frameworks (at least for the custom accelerator used in the work [8]). There are several other works including the FINN framework [10] [11], the same can't be said about VitisAI since it's more recent than FINN. It is stated in [8] that VitisAI provides a more sequential approach than FINN which is a more fine-grained oriented accelerator in spite of being outperformed. This is because FINN may be more suited for low powered FPGAs since the HW resources used depend on the model being accelerated, and quantizations lower than 8 bits are supported.

The work discussed in [12] explains trade-offs between using High Level Synthesis (HLS) and Overlays where it was found that HLS provides lower power consumption in comparison with Overlays, but using Overlays provides much faster processing. Using HLS would suit better this dissertation's objectives, which stand for prioritizing lower power consumption. However, since relying only on HLS is still a very extensive job, a better approach lies on using tools like FINN and VitisAI. Further discussion and explanation about how they operate are detailed in **Chapter 3** and **Chapter 4**.

1.5 Key Contributions

The dissertation's practical contributions are included in the topics below for the benefit of the interested public or those who wish to further pursue this work.

1. The theoretical explanation can provide a shortcut for someone new to the field to gain the necessary information to carry on with this work.

2. Problems were discovered in the brakish dataset [13], which is commonly utilized in state-of-the-art for detection algorithms in underwater environments. These issues stemmed from a lack of variation in the inputs and were resolved by using data augmentations.

3. The data augmentations improved detection results and allowed our model to compete with other state-of-the-art models that employ more sophisticated architectures. Currently, the model with the best mAP in the state of the art [6] for this dataset is 93.56% with YOLOv4, while 92.40% mAP was reached in this dissertation with YOLOv3-Tiny.

4. The comparisons between FINN and Vitis-AI are given as a consequence of the knowledge collected throughout this project. Additionally, there are descriptions of how these frameworks function that are gathered in one place and are more concise than those found online.

5. In Vitis-AI, an alternate workflow is encouraged, which avoids the issue where Darknet is not supported in the Pytorch environment for Vitis-AI.

6. Finally, results for the implementation using the *Zynq UltraScale+ MPSoC ZCU104* FPGA [14] are provided. The accelerated version with reconfigurable logic consumes 56.364% less energy than the GPU, utilizes less hardware resources, maintains a decent mAP of 82.6% , and has a good performance around 70.05 FPS.

1.6 Overview of the Dissertation

Chapter 1 - Introduction for the dissertation, presenting the context, motivation and objectives of this work, as well as discussion of similar works related with this dissertation and contributions that it brings to the community. **Chapter 2 - Theoretical Background**, explaining theory behind convolutional neural networks, discussing and comparing YOLO architectures, and finally explaining filtering algorithms used for underwater images. **Chapter 3 - Frameworks and Design Methodologies** contains an overview of the tools used in this work to run a detection algorithm for YOLOv3-Tiny in an FPGA. **Chapter 4 - Implementation Description and Methods** describes implementation methods in order to make it possible for other people to reproduce similar results. **Chapter 5 - Results and Discussion** discusses the results that were obtained during the fulfillment of the objectives described in **section 1.3**. **Chapter 6 - Conclusion and Future Work:** the project's results, as well as the expectations for future work, are provided with the goal of continuing the research conducted in this work.

Chapter 2

Theoretical Background

2.1 Convolutional Neural Networks

In this following section we will give an overview of how CNNs operate. For further details see [15].

When studying the Deep Learning field, it is common to learn about CNNs as it is the most famous algorithm. CNNs are on the top of their game when it comes to identifying relevant features on images. Therefore, they're a widely used in computer vision, for object detection fields. A CNN is a type of supervised ANN as it requires pre-labeled inputs for the learning process.

A CNN is first planned and described by an architecture that should be carefully built and tuned depending on the application. An example architecture is represented in the figure below:

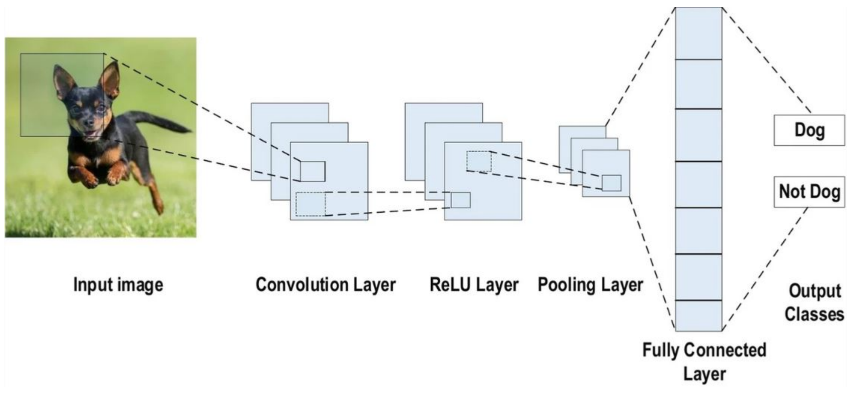


Figure 2.1: Example architecture from [15].

The architecture approach in CNNs was inspired by the analysis of the complex sequence of cells present in the visual cortex of animals, more specifically in a cat's visual cortex [16].

Usually there are three types of layers, described in detail in the following sections: the **convolutional layer** (uses 2D kernel convolution on the image), the **pooling layer** (also 2D) and the **fully connected layer** (1D). So, in the image above we have three layers (the ReLU is an activation function).

The most common way to build an CNN architecture is to stack convolutional layers followed by pooling layers, to reduce the number of parameters within the activation [17]. Inside a convolutional layer, the received input is filtered with kernels which use the weights of the network as coefficients. In other words, kernels are learnable filters. The output of a convolutional layer is the output of neurons that are connected to regions of the input image, through these convolutions done with kernel filtering. These outputs are interpreted by an activation function represented in the figure (labeled as ReLU layer, the rectified linear unit function). As any other activation function, it decides the output of a neuron. It is used to activate the outputs of convolutional and fully connected layers. With that, we can use more layers on top, or in case we are on the last layer, we can classify the input, based on the value of our weights. Each of these layers and many other elements are analyzed in detail in the following sections.

2.1.1 Convolutional Layers

Convolutional Layers are the backbone of a CNN. They are essentially a group of filters. The layers' parameters and channel depth determine the dimenson of a set of learnable channels (neurons) [18]. During the forward pass (training) all channels calculate the dot product between the kernel and the input, shifting the kernel across the input by a stride of one (other stride values can be used), as shown in **figure 2.2**, where the kernels are the green matrices, inputs are on the left and the outputs are the feature maps on the right side. The blue matrices represent the input feature map area that is being multiplied by the kernel. In this example no padding was used, because of this, the feature map dimension was reduced at the end of the convolutional layer. Using padding also brings better results when analysing the inputs' border information. The dimensions of the feature map are also dependant on the kernel stride, having a higher stride produces lower dimensions. All of these parameters (stride, channel depth, padding and kernel size) are very important during practical implementation and define how many features and parameters we create in a single convolutional layer.

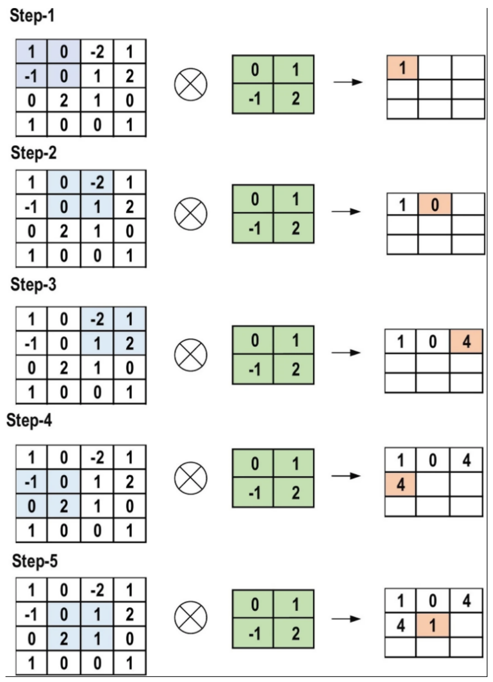


Figure 2.2: Calculations on a convolutional layer [15].

The result of this forward pass operation is a **feature map**. At every corresponding location of the input, each feature that is to be extracted will be part of a feature map after the convolution with the kernel [19], as shown in **figure 2.2**, also represented by the following expression from [19]:

$$y_j = b_j + \sum_i x_i k_{ij} \tag{2.1}$$

Where y_j is the output feature map, and j iterates through the number of channels in the layer. k_{ij} represent the filters' weights, x_i the input features (or pixels) and b_i the bias.

At the beginning of training, the kernel values are usually randomly assigned, but this can vary depending on the weight initialization method that is being used. And then, during training the weight and bias values are adjusted by the gradients during back propagation. And with some repetition the kernel improves its values enough to able to extract significant features.

In networks like the Multi Layer Perceptron (MLP), each neuron of a FC layer links with all neurons in the following layer. This doesn't happen in CNNs, because in between layers only a few connections are made. This way, even though we are learning from a complex image data, we have lower memory requirements in comparison with other ANNs because of the selective connections between weights. There are also a set of different parameters (such as the convolutional layer depth, stride, and kernel size that we talked before), techniques like max pooling, and the number of layers in the architecture that control the complexity of the network and have direct impact on the memory usage. Also in contrast to other ANNs there are no weights between two neurons of adjacent layers, therefore, adjacent layers share each other's weights. This makes sense as all weights operate with all pixels of the input, treating them as a single group of weights will reduce training time.

2.1.2 Pooling Layers

The pooling layer downsamples the feature maps it receives as input. These inputs are originated from the convolutional layer output, after going through the activation phase. There are different kinds of pooling, depending on the situation one type of pooling can be more effective than other, but essentially all of them shrink the input feature map into a smaller one, maintaining relevant information. The most common types of pooling are max, min and global average, for an example see **figure 2.3**. As the name implies, in the average pooling, the average of the values is calculated and it replaces the values under the kernel with the average. Max pooling replaces them with the maximum value and min polling with the minimum. Parameters like kernel size and stride are also tunable in this layer.

2.1.3 Activation Function

The main task of an activation function is to map the input to the output, in all kinds of networks. Given the output of a layer with weights, in this case, the convolutional and fully connected layers, we can use them in this state as inputs to the activation function and decide to activate or certain neurons, with reference to a specific input. This decision is made by calculating the weighted sum of the neurons received from learnable layers, along with the bias. Other important task done by the activation function is to cause non-linearity on the input/output maping, in order to learn extra difficult things through back propagation, thus this also means that this function bust be differentiable. There are different types of activation functions, being the following types some the most common:

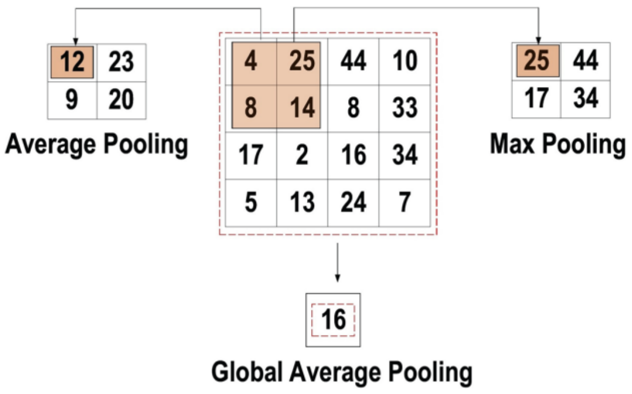
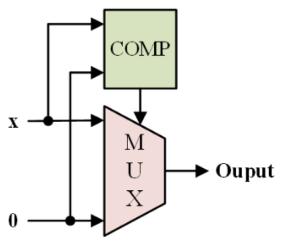
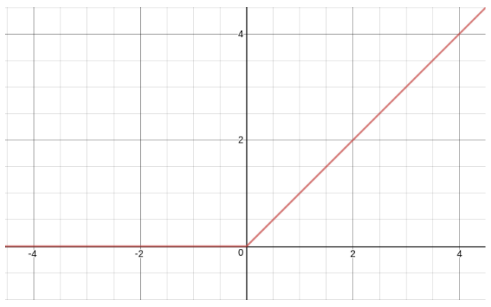


Figure 2.3: Different kinds of pooling [15].

- Sigmoid: Maps the input into values between zero and one, represented by the following expression:
$$f(x)_{sigm} = \frac{1}{1+\exp^{-x}}$$
- Tanh: Maps the input into values between -1 and 1, represented by the following expression:
$$f(x)_{tanh} = \frac{\exp^{x} - \exp^{-x}}{\exp^{x} + \exp^{-x}}$$
- ReLU: This is the most famous activation function in CNNs, and it is a very simple one, with low computational load. It maps all values to positive numbers. In hardware all it needs is a comparator and a multiplexer as shown in **figure 2.4**. It can be described by the following expression:
$$f(x)_{ReLU} = max(0, x)$$



(a) ReLU block diagram approach [20]



(b) Graphical representation [21]

Figure 2.4

As we can see, for negative inputs, the output is zero. And for positive inputs, the output is equal to the input. This can cause some problems in case we have a lot of negative inputs, the network will "die" as most of the gradients flowing through back propagation will not contribute to the learning process (the output will always be zero). This can be caused by multiple reasons (for example a large bias value). The following type of activation function introduces a simple fix to this problem.

- Leaky ReLU: This activation function is a improvement to the ReLU, in the special case where there's a risk of having many negative values flowing through. Unlike the ReLU, it doesn't ignore the negative values. Instead of having the constant function $f(x) = 0$, $x < 0$ we have a linear function with a parameterized small slope as shown in the **figure 2.5**. It can be represented through the following equation:

$$f(x)_{LeakyReLU} = \begin{cases} x, & \text{if } x > 0. \\ mx, & \text{otherwise.} \end{cases} \tag{2.2}$$

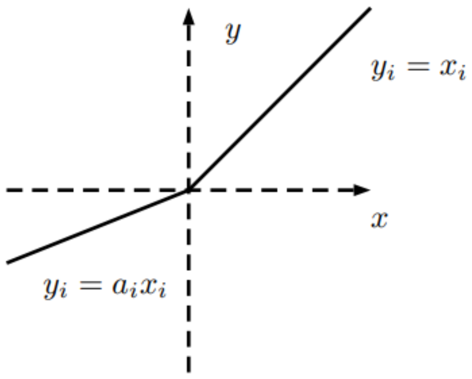


Figure 2.5: Leaky ReLU graphical representation [22].

The factor m is usually a very small value (such as 0.001), in order to create a slight slope.

2.1.4 Fully Connected Layers

Using only convolutional and pooling layers we always end up having 2D data, which can not be easily classified. By flattening this 2D data into a 1D vector, it can be used as input in a Fully Connected Layer. These layers work with 1D vectors and are used in ANNs like MLP. In these layers each neuron is connected to all neurons of the neighbouring FC layers. Reducing the final number of neurons to equal the number of classes, the class with higher accumulated value is the predicted one, as it can been seen in **figure 2.6**.

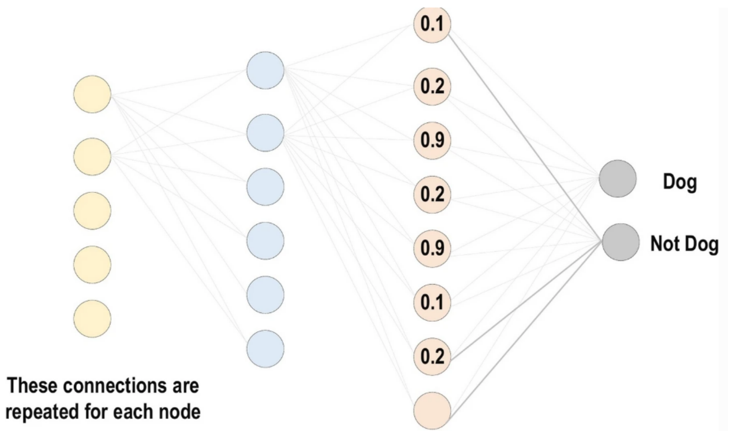


Figure 2.6: Fully connected layer [15].

2.1.5 Loss Function

Loss functions, in the vast majority, use the predicted value obtained from the output layer and the labeled output coming from the dataset (which is why CNNs are a supervised model) to calculate an error value. The error value is used by an optimizer to adjust the values of the weights during back propagation, and thus making the network able to adapt itself to make better decisions. Several types of loss functions exist, but the ones we will be using in the YOLO architecture are applied a little different than usual, and will be discussed in more detail in the following **section 2.2** since we will be adapting the position and size of bounding boxes.

2.1.6 Regularization

Obtaining state of art results in CNNs requires some sophistication of the architecture, and more complex models are more likely to suffer from overfitting. A model is over-fitted when it doesn't perform well on unseen data but excels on training data. To detect overfitting, a testing dataset can be used, and see how the model performs on it, or use a validation set and notice that the validation loss is a lot higher than the training loss during the training process. In such case, the model is too complex for the input data, and the network has too many parameters. The solution lies in adapting the architecture to have less layers or use regularization methods, some of which are listed below.

Dropout

Dropout consists in randomly eliminating nodes from the neural network at the end of each training epoch, increasing generalization in the model by allowing it to learn different sets of independent features, reducing overfitting.

Data Augmentation

Data augmentation is commonly used to solve overfitting or to obtain better results in an already just-fitted model. If we use data augmentation to solve overfitting, it can be seen as the other way around to solve the problem, because it lies in adapting the input data instead of the architecture or the training algorithm, by expanding the size of the dataset and creating more diverse data. There are many data augmentation methods usually based on applying different transformations on images such as rotations, resize, translations, crops, or even filtering.

Batch Normalization

Batch normalization may not have as much influence in regularization as the previous methods, but it is a common practice and has a lot of impact during CNN training. It uses a Gaussian Distribution (unit Gaussian to be exact) to ensure the performance of activation function outputs, as in each activation layer there is a variation in the activation distribution that defines the internal covariance shift. This shift keeps accumulating every time the weights are updated via back propagation causing a slower convergence in the loss function. The output of the activation can be normalized by subtracting the mean and dividing it by the standard deviation (batch normalization), and this is usually done as a "pre-processing" for each convolutional layer. Batch normalization also prevents vanishing gradients, decreases training time and decreases the consequences brought by poor weight initialization.

2.1.7 Optimizer

When training a CNN and any other neural network, the core of their learning process is the optimization algorithm that is being used to minimize the value of the loss function. The classical optimization function is the gradient descent, it reaches the local minimum easily but struggles to reach the global minimum value. The gradient descent updates the network parameters every training epoch by calculating the negative derivative of the loss function, allowing it to obtain a valid descending direction from that point.

When using this method, there has to be a condition to decide when to stop, and this means admitting that it has found the minimum. For this, the optimizer can, for example, stop the iterative process when the gradient starts to disappear.

Many enhancements were applied to the gradient descent, creating other optimization algorithms such as AdaDelta, Adagrad and Momentum.

2.1.8 Back Propagation

In the previous **section 2.1.5** it was stated that the error value is used by an optimizer to adjust the values of the weights during back propagation, and thus making the network able to adapt itself to make better decisions. This process itself is called back propagation, as it is a technique for changing weight values to account for each error discovered during learning. Like it was explained in the optimizer **section 2.1.7**, the gradient of the loss function with respect to each weight is computed, and the new value of the weight is updated by adding the value of the gradient, like it is demonstrated in **equation 2.1.8**, which represents the gradient descent optimization function (where α is the learning rate, **g** is the loss function, w^k is the new value of the weight, and w^{k-1} the old value). However, since the gradient is computed one layer at a time, iterating backwards from the last layer, this process is called back propagation.

$$w^k = w^{k-1} - \alpha \nabla g(w^{k-1}) \tag{2.3}$$

CNNs can learn thanks to the back propagation algorithm, which uses the optimizer and loss function. This method is widely used by many supervised feed forward neural networks.

2.2 YOLO - You Only Look Once

The first three versions of YOLO were developed by the original creator, and the following ones were made by other members of the community, the most recent version is YOLO v7 [23]. The complexity of the network grows in each version, and it becomes less and less trivial to implement the architecture on an FPGA, and there isn't much development regarding this topic starting from YOLOv5 onwards. The evolution from one version to another will be explained in more detail in the first four versions as they are the ones that are more commonly used on FPGAs.

Yolo v1

Introduction: The first version of YOLO [24] was published in 2016 and became well-known among other famous architectures for being faster. It is called YOLO - You Only Look Once because it predicts bounding boxes and associated class probabilities in only one evaluation. It also had better generalization compared to other state of art architectures at the time, like R-CNN [25] that uses region proposal techniques to create possible bounding boxes in an image, before applying a classifier to these boxes. After classification, bounding boxes are improved, duplicate detections are removed, and the boxes are given new scores based on additional objects in the scene as a post-processing. Because each component of these intricate pipelines must be trained separately, they are slow and challenging to optimize. Other architecture that was compared to YOLO in the work [24] was deformable parts models (DPM) [26] that employs a sliding window method, where the classifier is run across the image at evenly spaced intervals. The article refers and compares a lot these two architectures as they were famous architectures that had the same finality as YOLO.

Detection: In order to train a YOLO network, the dataset needs to have annotations about the true location of the object, for each object in each image having the details about its' bounding box such as coordinates of the box's borders (the minimum and maximum height and width values).
As it is explained on [24], the first step is to divide the input image into an SxS grid. If the center of the object is inside a certain cell, that cell becomes responsible for detecting the respective object.
Each grid cell is able to predict numerous bounding boxes for a same object, applying confidence scores for each box. The objective is to obtain a confidence score to equal the intersection over union between the true box and the predicted one. So, to predict a box and a confidence score, it needs to predict 5 values for each box: x, y, w, h, c, where x and y are the coordinates of the center, w and h are the width and height, and c is the confidence. Regardless the number of boxes inside a cell, only one set of class probabilities is predicted for each cell. Then, in order to have the class specific confidence score value for a certain box, the class probability and the confidence prediction is multiplied.

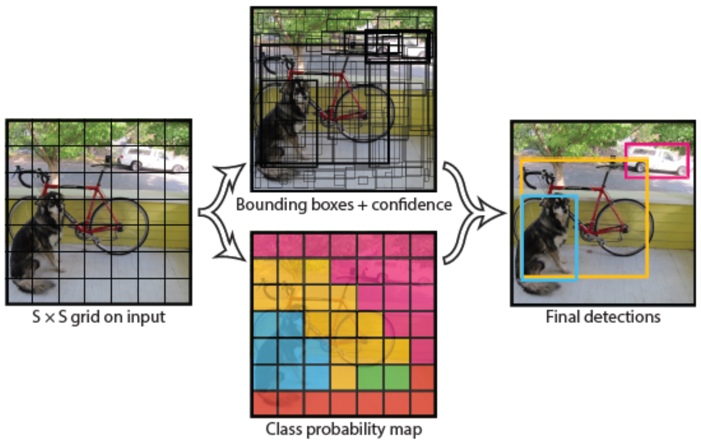


Figure 2.7: YOLO Detection system [24]

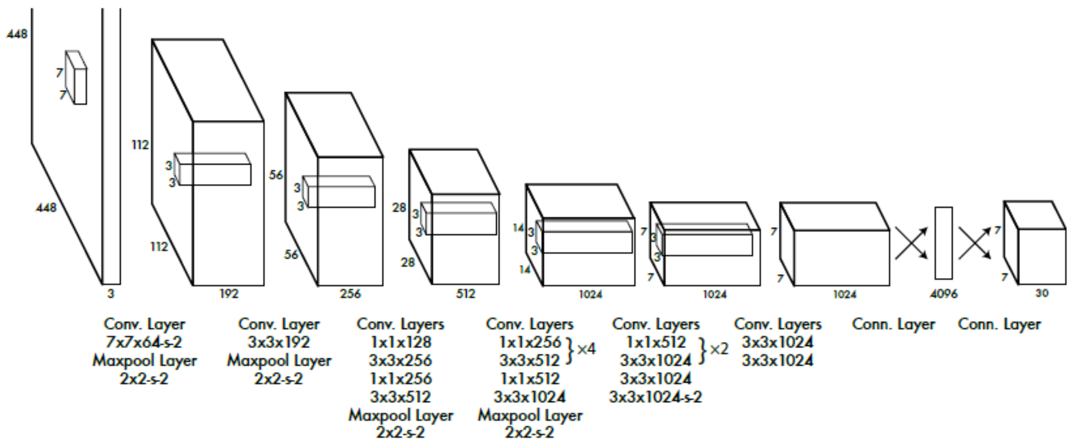

448
7
7
448
3
112
112
3
3
192
56
56
3
3
256
28
28
3
3
512
14
14
3
3
1024
7
7
3
3
1024
7
7
1024
4096
7
7
30
Conv. Layer 7x7x64-s-2 Maxpool Layer 2x2-s-2
Conv. Layer 3x3x192 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x128 3x3x256 1x1x256 3x3x512 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x256 3x3x512 ×4 1x1x512 3x3x1024 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x512 3x3x1024 ×2 3x3x1024 3x3x1024-s-2
Conv. Layers 3x3x1024 3x3x1024
Conn. Layer
Conn. Layer

Figure 2.8: YOLO v1 Architecture[24]

Loss Function: The **loss function** (represented in **figure 2.9)** is a combination of summed squared errors. Parameters λ_{coord} and λ_{noobj} exist to create different weights for cells with and without objects respectively. Otherwise, cells that have no objects would overpower the gradient for cells that have objects, making convergence sometimes impossible during training. The default value of the parameters are $\lambda_{noobj} = 0.5$ and $\lambda_{coord} = 5$. Is also important to note that the loss function penalizes classification error only when an item is present in the grid cell, and it penalizes bounding box coordinate error only when that predictor is accountable for the ground truth label, i.e. it has the greatest IOU of any predictor in that grid cell. Needless to say, at each epoch the parameters of the box (present in the loss function) are adjusted according to the loss value produced, in order to maximize the IOU.

$$
\begin{gathered}
\lambda_{\textbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
+ \lambda_{\textbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right] \\
+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2 \\
+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2 \\
+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2
\end{gathered}
$$

Figure 2.9: YOLO v1 Loss Function where "$\mathbb{1}_i^{obj}$ denotes if object appears in cell i and $\mathbb{1}_{ij}^{obj}$ denotes that the jth bounding box predictor in cell i is responsible for that prediction" [24].

Issues: As stated in [24], there are several issues with this version of YOLO:

1. Since each grid cell can only predict two boxes and only have one class, YOLO places

substantial spatial limits on bounding box predictions. As a result, the number of adjacent items the model can anticipate is constrained.

2. The model finds it difficult to generalize to objects with novel or odd aspect ratios or configurations since it learns to predict bounding boxes from data.

3. Given that the architecture comprises many downsampling layers from the input picture, it also employs somewhat coarse characteristics to predict bounding boxes.

4. When training using a loss function that approximates detection performance, errors in small and big bounding boxes are handled equally. A minor error in a large box is unimportant, while a minor error in a tiny box has a far bigger impact on IOU score.

Yolo v2

In the previous topic, a more detailed introduction was made to the yolo architecture, and now, on the following versions v2, v3 and v4, the discussion will be more centered on the improvements applied in-between versions. So, lets move on to discuss the improvements made from yolov1 to yolov2 as known as yolo9000 [27]:

1. **Batch Normalization:**
   The benefits of applying batch normalization have been discussed before in the previous **section 2.1.6**, and by adding batch normalization to convolutional layers in the design, the MAP enhanced (mean average precision) by 2%. It also aided the model's regularization, and overall overfitting has been decreased.

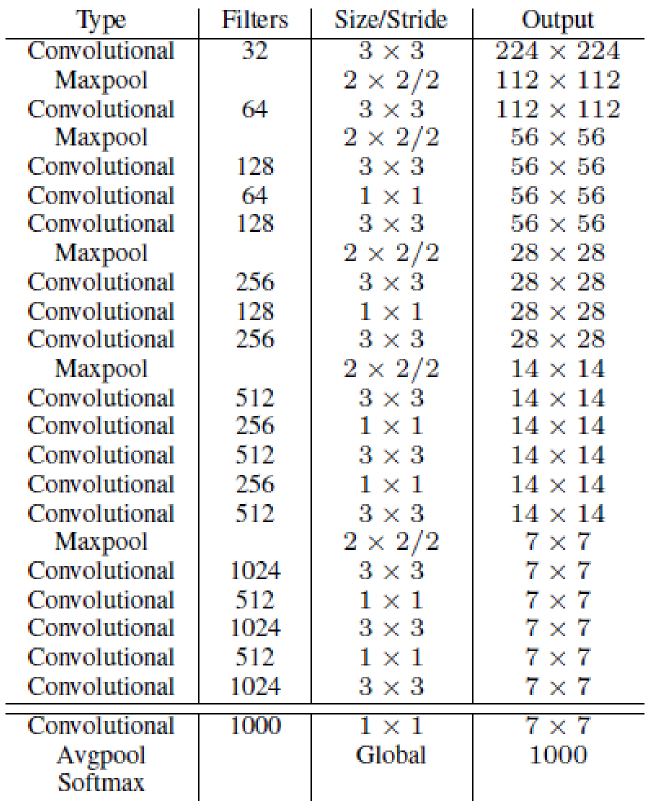
2. **Higher Resolution Classifier**
   In YOLO v2, the input size was raised from 224*224 to 448*448. The mean average precision (MAP) has increased by up to 4% as a result of the image's larger input size.

3. **Anchor Boxes:**
   The anchor boxes were introduced in YOLO v2, which is one of the most noticeable modifications, as they allow one grid cell to detect multiple objects. These anchor boxes are in charge of predicting the bounding box, using k-means clustering in order to adapt to a specific dataset. More details about how they're implemented can be found at [27].

4. **Fine-Grained Features:**
   In YOLO v1 there were several problems when detecting tiny objects in images. Although using higher resolution helped, dividing the image in 13*13 grid cells and adding a passthrough layer that brings features from an earlier layer at 26x26 resolution allowed the recognition of tiny things in a picture while simultaneously being successful with bigger objects, by concatenating higher with lower resolution features.

5. **Multi-Scale Training:**
   The model in YOLO v2 is trained using random photos with varied dimensions ranging from 320*320 to 608*608. As a result, the network can accurately learn about and predict the objects from different input dimensions.

6. **Darknet 19:**
YOLO v2 uses the Darknet 19 architecture, which consists of a softmax layer for object classification, five max pooling layers, and 19 convolutional layers. Below is a picture of Darknet 19's architecture.



Type	Filters	Size/Stride	Output
Convolutional	32	3×3	224×224
Maxpool		$2 \times 2/2$	112×112
Convolutional	64	3×3	112×112
Maxpool		$2 \times 2/2$	56×56
Convolutional	128	3×3	56×56
Convolutional	64	1×1	56×56
Convolutional	128	3×3	56×56
Maxpool		$2 \times 2/2$	28×28
Convolutional	256	3×3	28×28
Convolutional	128	1×1	28×28
Convolutional	256	3×3	28×28
Maxpool		$2 \times 2/2$	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Maxpool		$2 \times 2/2$	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	1000	1×1	7×7
Avgpool		Global	1000
Softmax			

Figure 2.10: Darknet 19 architecture [27].

Yolo v3

This version is labeled as an incremental improvement, the improvements are summarized below, for more details refer to [28].

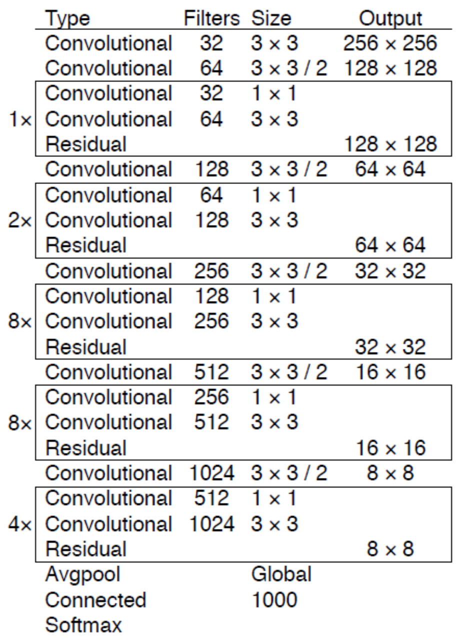
1. **Bounding Box Predictions:**
The objectness score for each bounding box is now predicted using logistic regression, outputting 1 if the box overlaps the ground truth object more than the other bounding boxes. There is also a threshold to ignore bounding boxes that are not the best, but overlap the ground truth object (working similar to a non-maximum suppression). By doing so, only one bounding box is assigned for each object.

2. **Class Predictions:**
Instead of using softamx for classification, YOLOv3 uses logistic regression providing multi-label classification, allowing one object to have two different labels (for example, if we are detecting underwater species, we can label a codfish as fish and codfish, and a crab as a crustacean and crab).

3. **Predictions Across Scales:**
Following the example from Feature Pyramid Networks [29], YOLOv3 generates

three bounding boxes for three different scales of the input image, with features being collected from each prediction, providing a better performance in different scales. Each prediction is made up of a bounding box, objectness, and 80 class ratings. Upsampling from preceding layers enables for the extraction of entire semantic information and finer-grained information from earlier feature maps, a process similar to that utilized in YOLOv2.

4. **Darknet 53:**
   A new, deeper and more complex feature extractor than the previous **Darknet 19** is used in YOLO v3 as its backbone, composed by some shortcut connections and multiple 1x1 and 3x3 filters.



	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
	Convolutional	32	1 × 1	
1×	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
	Convolutional	64	1 × 1	
2×	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
	Convolutional	128	1 × 1	
8×	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
	Convolutional	256	1 × 1	
8×	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
	Convolutional	512	1 × 1	
4×	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Figure 2.11: Darknet 53 architecture [28].

Yolo v4

YOLOv4 is a significant improvement of YOLOv3, only published in 2020 and made by other developers that chose to continue Joseph Redmon's work in YOLOv3 after Joseph announced that he wouldn't be developing any more YOLO versions due to the misuse of this technology. YOLOv4 achieved much better results at the cost of a more complex system. The MAP improves around 10% and the frame rate (FPS) improves as much as 12% [30]. The new architecture is represented in **figure 2.12**. The new architecture is composed of 4 blocks, listed below. There are more methods enumerated and explained in the YOLOv4 article [30], used to improve accuracy during and after training called bag of freebies and bag of specials that are not covered here in detail, as this represents only a quick overview to understand what kind of improvements could be applied to YOLOv3:

1. **Backbone:**
   It represents the feature extractor based on **Darknet 53**, CSPDarknet53 (CSP meaning Cross-Stage-Partial-connections). CSP is used to split the a layer in two

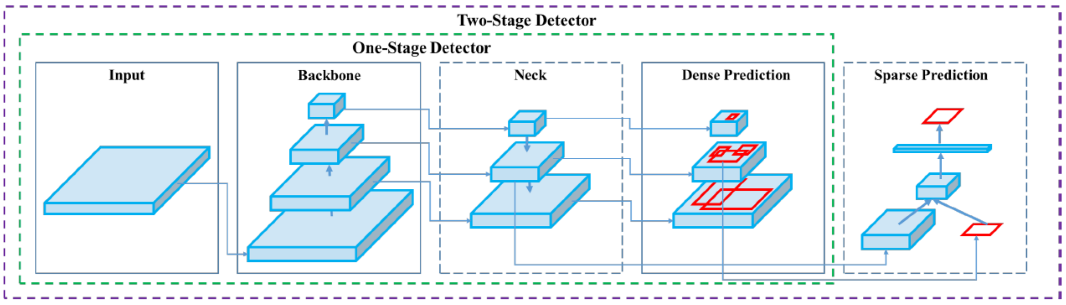


Figure 2.12: Object detector [30].

partitions, represented in the figure below, where the first partition doesn't go through Darknet53's convolutional layers and the second does. The outputs are then concatenated providing better gradient back-propagation.

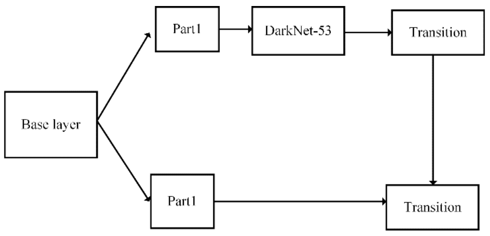


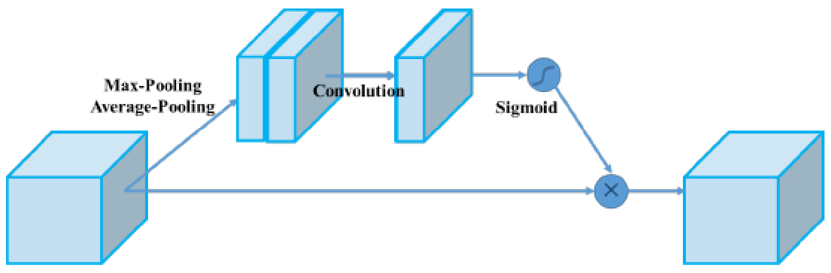
Figure 2.13: CSPDarknet53 diagram from [31].

2. **Neck:**
   The neck represents some additional layers between the backbone and the dense prediction. The inputs that reach the neck pass through a path aggregation network (PAN) [32], a spatial attention module (SAM) [33], and spatial pyramid pooling (SPP) [34] which improve accuracy in the model by concatenating information. The neck uses modified versions of these methods, represented in the figures **2.14**, **2.15** and **2.16**.

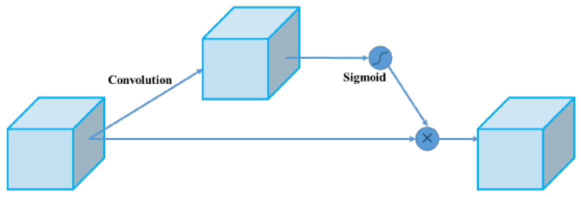
3. **Head:**
   The head can be a two stage or a one stage detector. One stage detectors only use the dense predictor, while the two stage detectors use dense followed by sparse predictors. If we are using the anchor based approach, the dense predictor is based on architectures like YOLOv3 [28] providing the object detection and classification utility with bounding boxes, and the sparse predictor based on the R-CNN series [25][35][36][37][38]. If using an anchor free approach the dense predictor is based on architectures like CornerNet [39], CenterNet [40] or MatrixNet [41], while the sparse predictor is based on architectures like RepPoints [42].

Conclusion: Why YOLO v3 - Tiny?

Unlike what was done in the first versions, not much was said about the issues of newer versions, but they certainly exist. Besides, a new version is not always created with the sole purpose to solve the problems of the previous one. Often it is enough to have better results with the changes that were made in the architecture to realize that it solved some issues, substantially improving the previous version.

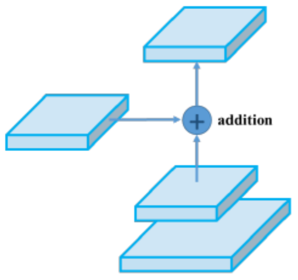


(a) SAM [33][30]

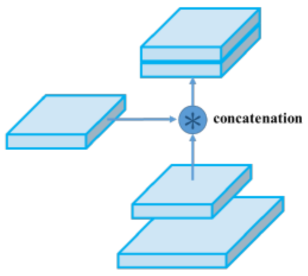


(b) Modified SAM [30]

Figure 2.14: Spatial Attention Module and modified Spatial Attention Module used in YOLO v4 [30]



(a) PAN [32][30]



(b) Modified PAN [30]

Figure 2.15: SAM and Modified PAN used in YOLO v4 [30]

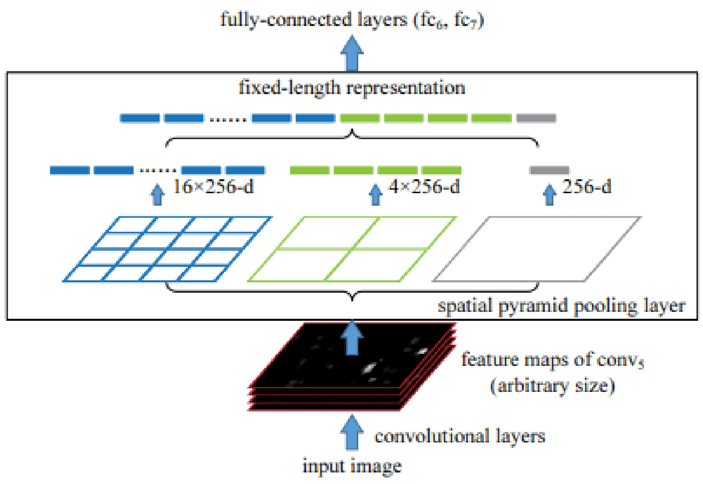

fully-connected layers (fc6, fc7)
fixed-length representation
16×256-d
4×256-d
256-d
spatial pyramid pooling layer
feature maps of conv5
(arbitrary size)
convolutional layers
input image

Figure 2.16: Spatial Pyramid Pooling [34]

For all versions there are more modest, reduced versions created in order to reduce the computational resources needed to run both training and classification algorithms, still using the ideas of the main architecture. These versions are often titled "tiny", "fast" or "small", or perhaps designated by other names that translate the same idea.

In this dissertation we focus on using YOLOv3-Tiny architecture, depicted in **figure 2.18**. The reason why a newer version was not chosen was because newer versions are more complex and require more hardware resources. Furthermore, the fact that it is an older version does not mean that it is incapable of bringing satisfactory results. The YOLOv3 as well as the YOLOv3-Tiny were published in 2018 and are good detectors. Just through what was explained above about YOLOv4, it can be said that there are tremendous increases in complexity on the newer versions, and the more complex something is, the less flexible it becomes to adapt, specially when working around challenges arisen from accelerating it on an FPGA. Because of this, versions higher than version 3 were somewhat left out. As far as comparing version 3 to version 2, there were some reasons that justified for using version 3 over version 2:

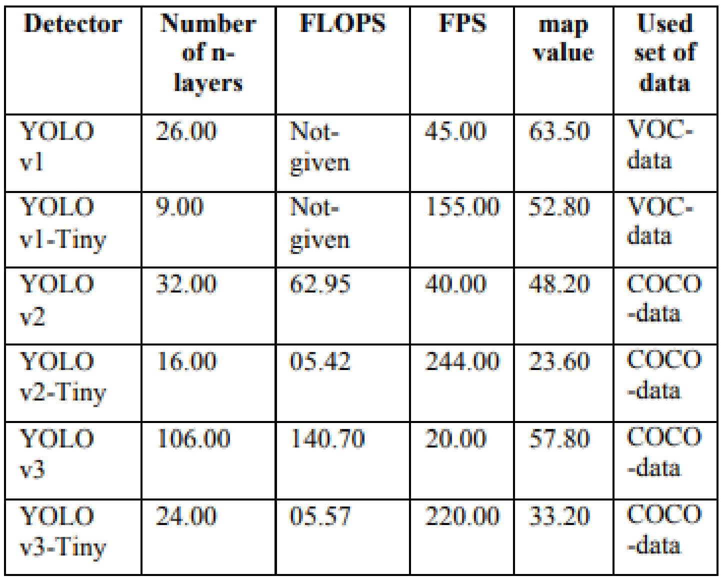
- When using older versions, one is more likely to encounter difficulties in getting around the use of outdated tools, or projects made by the community a few years ago that worked then, and no longer work. Sometimes because certain software versions, with time, became unavailable to the public.

- The attention given by the community to problem solving usually gravitates more around recent versions, and therefore it is easier to find support from the community for solving problems arising in more recent versions. Moreover, newest versions of software in general are usually compatible with other software launched around the same time.

- As it can be seen on the YOLO v1, v2 and v3 author's website [43] when referring to mAP in the COCO dataset, the YOLOv2-Tiny has 23.7% while YOLOv3-Tiny has 33.1%, with a 24 FPS drop and a small increase in FLOPs. These values do not fluctuate much from the ones obtained in [44] which can be seen in **figure 2.17**. The YOLOv2-Tiny uses 9 convolutional layers while the YOLOv3-Tiny uses 13, the number of weights and parameters in YOLOv3-Tiny is lower. YOLOv3-Tiny architecture proposes better mAP, compensating for the substantial drop in FPS.

- The number of FPS in this dissertation is not a benchmark on which we will be dependent, since we do not need to have high frame rate, but only to reduce energy consumption without sacrificing the accuracy of the detections.

That leaves yet another question. Why not use an architectures like the R-CNN series since it provides better accuracy sacrificing speed, in comparison with YOLO? In spite of the frame rate not being crucial, it is something that cant be ignored or allowed to be extremely low. After reading the studies and comparisons done in [45] YOLOv3 was up to 3 times faster than Mask R-CNN, still being able to provide good accuracy (although inferior to Mask R-CNN). Again, not only the speed and accuracy are important, but also the hardware resources needed to operate the detection algorithm and the complexity of the architecture.



Detector	Number of n-layers	FLOPS	FPS	map value	Used set of data
YOLO v1	26.00	Not-given	45.00	63.50	VOC-data
YOLO v1-Tiny	9.00	Not-given	155.00	52.80	VOC-data
YOLO v2	32.00	62.95	40.00	48.20	COCO -data
YOLO v2-Tiny	16.00	05.42	244.00	23.60	COCO -data
YOLO v3	106.00	140.70	20.00	57.80	COCO -data
YOLO v3-Tiny	24.00	05.57	220.00	33.20	COCO -data

Figure 2.17: Table taken from [44] comparing the first three versions of YOLO. Previously it was referred that YOLOv2-Tiny had 9 convolutional layers and YOLOv3-Tiny had 13. Which is true, the number of layers in this table does not refer to convolutional layers only, which are known to be more computationally expensive.

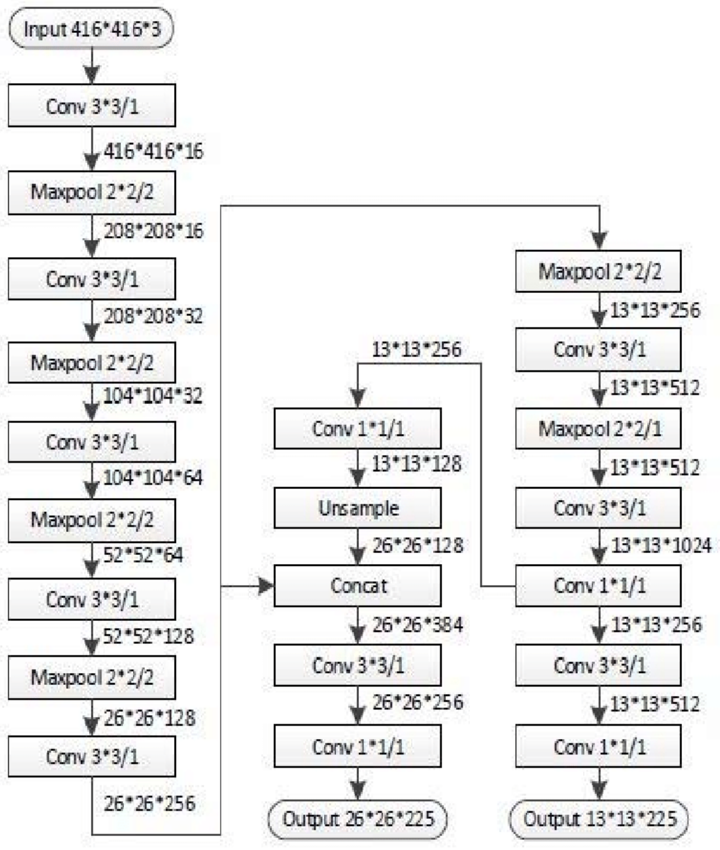


Figure 2.18: Architecture of YOLOv3 - Tiny [44].

2.3 Image Filtering for Underwater Environments

The dataset that will be used is the brackish dataset, presented in [13] where is made a comparison between YOLOv2 and YOLOv3 performances on this dataset. The dataset is open-source and linked in the article. This was the best dataset found available for this application case since it has annotations ready for YOLO. Datasets for underwater species with documented annotations in this format are rare, and the brakish dataset was the only one found available (and open-source) meeting the required criteria. Underwater images are usually hindered by radiance propagation, as seen when observing the greenish tint present in the sample below in **figure 2.19**.

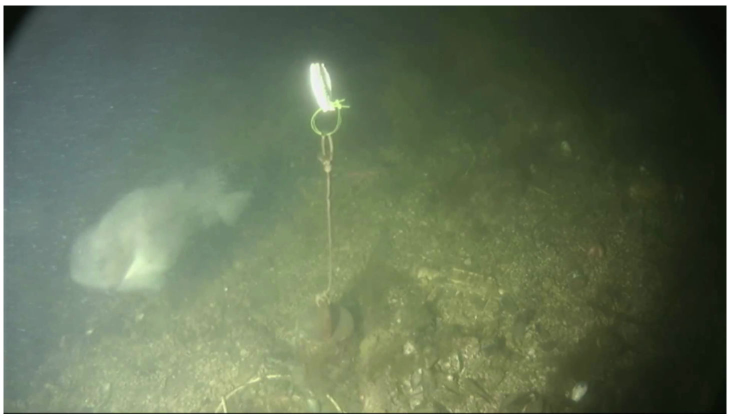


Figure 2.19: Sample from the brackish dataset [13].

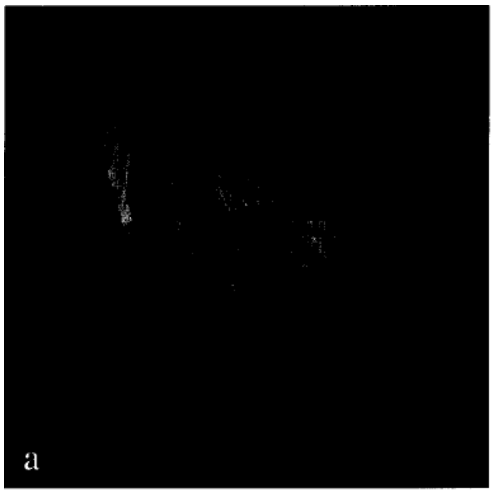
In this dissertation the following image filtering algorithms were tested and compared:

- **CLAHE** - Contrast-Limited Adaptive Histogram Equalization [46] [47] [48]
- **GC** - Gamma Correction [49] [50]
- **RoWS** - Removal of Water Scattering [51]
- **RGHS** - Shallow-water Image Enhancement Using Relative Global Histogram Stretching Based on Adaptive Parameter Acquisition [52]

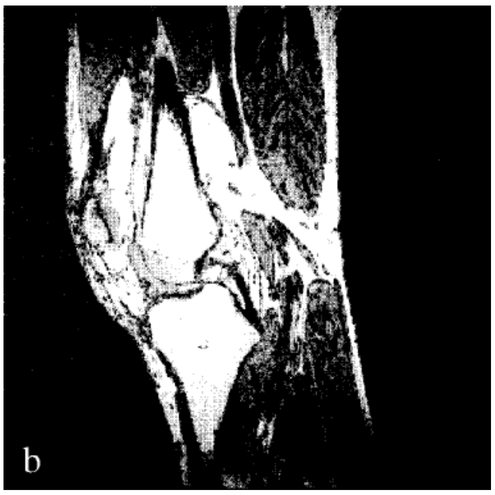
These algorithms don't focus on correcting greenish tint specifically, but are able to improve the quality of underwater images in general. Not all images in underwater environments will be affected in the exact same way as the images from this dataset, as they always depend on the camera and the surrounding environment. Also, the methods that were present in most of the SoA articles worked best for the specific set of figures used during the respective study. The first two methods are classic image enhancement algorithms: gamma correction and contrast correction. These two don't focus on underwater images but change the values of the pixels in order to provide better human perception. The other two algorithms however focus more on underwater images. The work [50] on gamma correction topic also talks about an interesting method to compensate the attenuation of the red channel in underwater images, however this was not implemented in this dissertation.

2.3.1 Contrast-Limited Adaptive Histogram Equalization

This image processing function has its roots in medical imaging, initially invented by Pizer [53], Ketcham *et al.* [54], and Hummel[55]. It is a contrast enhancement method, based on *histogram equalization*, which distributes the image's frequencies in an equalized manner. Naturally, it is specially efficient in medical gray images, but it also works well for colored images. In order to understand better the process, the demonstrations here will be using the medical images used in the work [46].



(a) Original image



(b) Same image after histogram equalization

Figure 2.20: [46].

However **figure 2.20 b)** is a lot better than the original for human perception, Histogram Equalization has two main problems:

1. Lack of adaptation to local contrast requirements, as there are some small contrast differences missing between adjacent regions.
2. Large peaks in unimportant areas like background noise (more noticeable after applying adaptive histogram equalization).

In order to accomplish better contrast between contextual regions, Adaptive Histogram Equalization (AHE) can be applied. It consists on dividing an image in a grid of rectangular contextual regions, and then calculate the optimal contrast for each region. For example, in an 512x512 image, we can create 64 contextual regions (sized 8x8 each). This will also create visible region boundaries, and in order to fix this, a bilinear interpolation is used (more details at [46]). The scheme of interpolation and result of AHE is depicted in **figure 2.21**.

After solving the contrast between adjacent regions, the other problem is left unsolved. It is now even more visible tons of large peaks in background areas (noise). To solve this problem, Contrast Limited Adaptive Histogram Equalization (CLAHE) is used, which limits the contrast enhancement on homogeneous areas like the background area. These homogeneous areas are represented by a large peak in the histogram as many pixels fall into that area (the majority of the pixels has that value range, since it belongs to the background). So, by clipping the peak of the histogram and equally distributing the pixels that were clipped over the whole histogram, these peaks can be largely attenuated. There

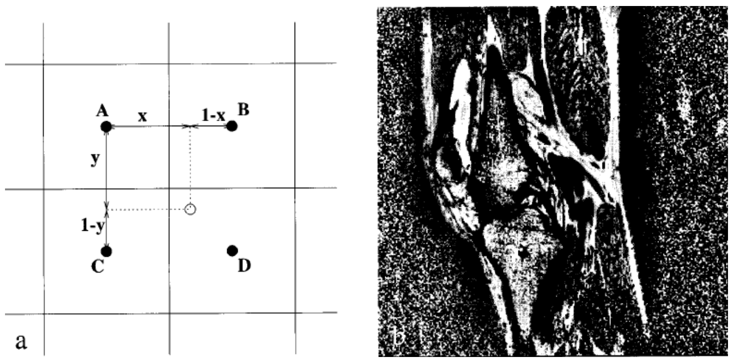


Figure 2.21: Bilinear interpolation scheme on the left side, explained in more detail at [46]. On the right side, the result of AHE [46].

is a parameter called the contrast factor that defines how much clipping will be done. Having a low contrast factor means having limited contrast enhancement but clearing most of the peaks from the background noise. So, having a balanced value for the contrast factor is important in order to use AHE without having much noise.

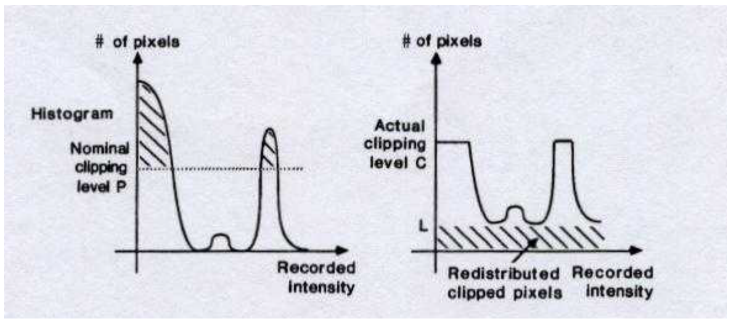


Figure 2.22: Histogram clipping with CLAHE [47].

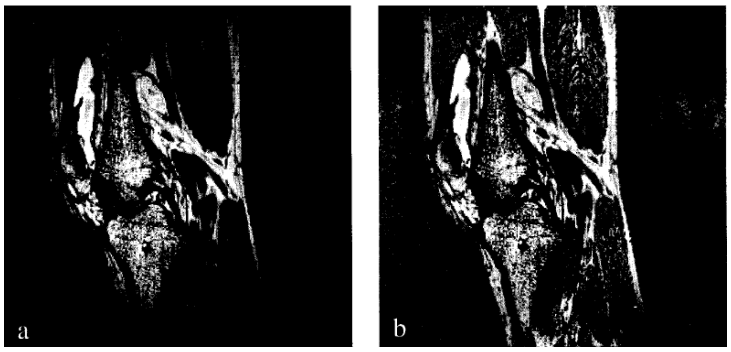


Figure 2.23: (a) Using clip limit of 3; (b) Using clip limit of 10 [46].

2.3.2 Gamma Correction

The gamma correction used was the traditional gamma correction expression [49]:

$$I_{out} = cI_{in}^{\gamma} \tag{2.4}$$

I_{in} and I_{out} are the input and output pixels, c and γ are two parameters that shape the transformation curve. Using $\gamma > 1$ makes the image darker and using $\gamma < 1$ makes it brighter. The most common case is using $c = 1$. There isn't much more to discuss about this topic, since these parameters were tuned accordingly with the type of images present in the dataset.

2.3.3 Removal of Water Scattering

In most of clear water images, there is always a color channel that has low intensity at some pixels. But, in more turbid waters, some regions where it should be really dark are illuminated by particles that are present in the path of light waves, and therefore radiate energy making dark regions brighter. The low intensities in the dark channel are mainly due to shadows, colorful objects and dark surfaces. Since natural underwater images have lots of plants, animals and rocks, these images have dark channels with really low intensities.

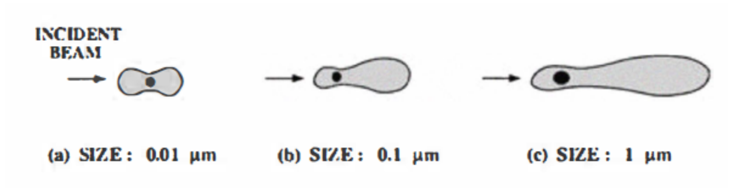


Figure 2.24: Particles in the path of light waves [51]

Mathematically, there is the following expression, where ***I*** is the observable intensity, ***J*** is the hindered intensity that needs to be recovered, ***B*** is the background light, ***t*** is the transmission function, and ***x*** is the center of the local patch being restored [51]:

$$\mathbf{I}(x) = \mathbf{J}(x)\mathbf{t}(x) + \mathbf{B}(1 - \mathbf{t}(x)) \tag{2.5}$$

So ***J*** is the dark channel that needs to be restored and it can also be represented as [51]:

$$J_{dark} = min(min(J_c(y))), \quad \mathbf{c}\epsilon\{r, g, b\}, \quad \mathbf{y}\epsilon\Omega(x) \tag{2.6}$$

Where Ω is the respective patch being processed at center x.

In order to use the dark channel prior restoration method, we need to follow 3 steps:

1. **Estimation of Background Light**
   As it is explained in [51], the background light can be estimated from the input image by picking the brightest pixels in the dark channel, as they are opaque but lightened by the background light:
   $$B(RGB) = I(x) \tag{2.7}$$
   Where
   $$x = max(J_{dark}(i, j)) \tag{2.8}$$
2. **Estimation of the Transmission**
   Several steps are made in order to estimate the transmission, details at [51]. But for practical implementations, we apply the following expression:
   $$\tilde{t}(x) = 1 - min_c(min_{y\epsilon\Omega(x)}(I_c(y)B_c)) \tag{2.9}$$

3. **Restoration**
   Then we can finally take the last step and recover the final radiance:

$$J(x) = I(x)\frac{B}{max(t(x), t_0)} + B \tag{2.10}$$

   Where t_0 is a constant with the typical value of 0.1.

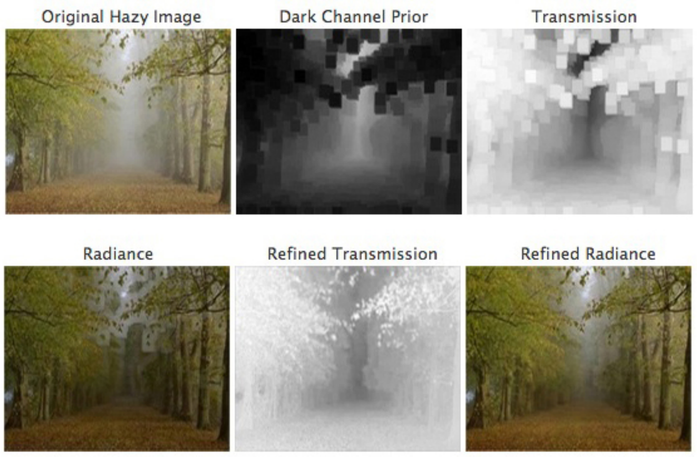


Figure 2.25: Example of radiance restoration using dark channel prior [56].

2.3.4 Relative Global Histogram Stretching

The whole process described in [52] is based on contrast correction followed by color correction, as depicted in **figure 2.26**. The article also covers concepts present in other image processing algorithms discussed before.

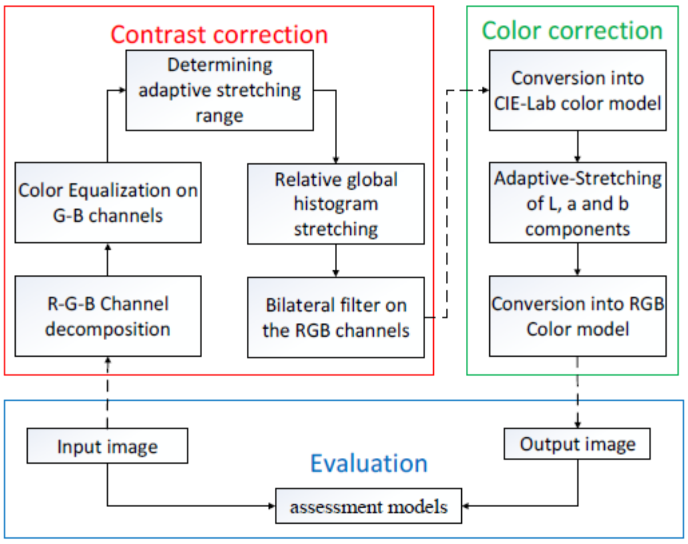

Contrast correction
Determining adaptive stretching range
Color Equalization on G-B channels
Relative global histogram stretching
R-G-B Channel decomposition
Bilateral filter on the RGB channels
Color correction
Conversion into CIE-Lab color model
Adaptive-Stretching of L, a and b components
Conversion into RGB Color model
Input image
Evaluation
Output image
assessment models

Figure 2.26: Process for performing RGHS. Described in [52]

.

Following the steps shown in the figure:

1. **R-G-B Channel Decomposition**
   This step only consists on separating the different color channels into three gray images.

2. **Color Equalization on G-B Channels**
   The 3 color channels are not equally balanced in underwater images. The absorption of the red channel happens due to bigger wave length, as it is shown in **figure 2.27**.

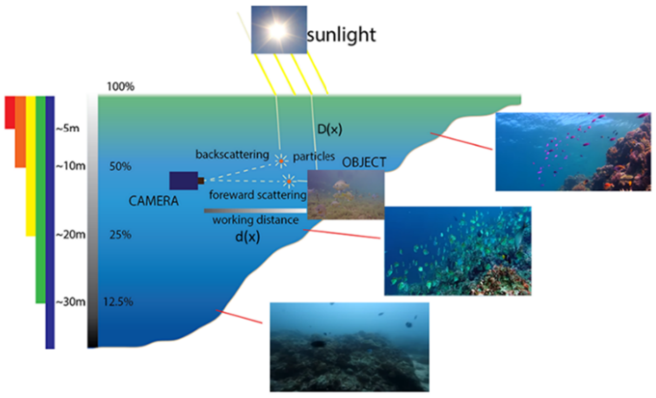


Figure 2.27: [52]

Since the presence of the red component of the spectrum is so low, it becomes hard to compensate, and even if we try to directly interact with the red light component, the image will over saturate with a red tint like it is shown in **figure 2.28**.

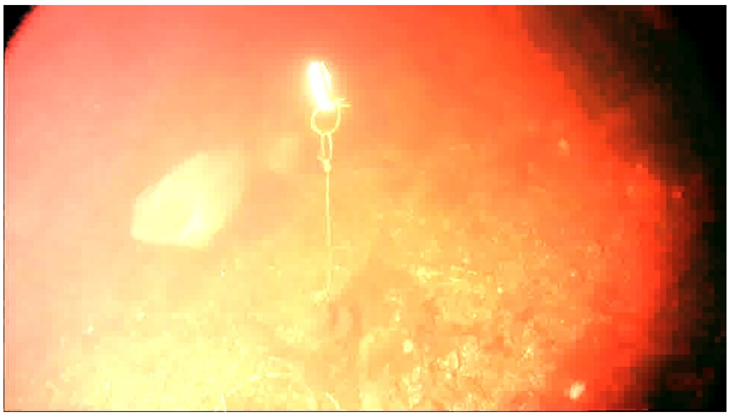


Figure 2.28: Extreme case scenario of a sample from the brakish dataset over saturated with a red color, after trying red channel compensation.

So, in order to achieve a good balance between the three color channels, the following parameters (θ_g and θ_b are multiplied with the remaining two color channels (green and blue):

$$G_{avg} = \frac{1}{255 * MN} \sum_{i=1}^{M} \sum_{j=1}^{N} I_g(i,j), \quad \theta_g = \frac{0.5}{G_{avg}} \tag{2.11}$$

$$B_{avg} = \frac{1}{255 * MN} \sum_{i=1}^{M} \sum_{j=1}^{N} I_b(i,j), \quad \theta_b = \frac{0.5}{B_{avg}} \tag{2.12}$$

3. **Determining Adaptive Stretching Range**
   This step focuses on obtaining parameters in order to perform the next step. So, in order to follow up better the whole process we recommend quick read through the next topic named Relative Global Histogram Stretching.

   - **Calculation of I_{min} and I_{max}:**
     The ideal would be choosing $I_{min} = 0.1\%$ and $I_{max} = 99.9\%$ since this represents the region of the histogram that is going to be operated on, and this way we would be eliminating an equal number of pixels from the two boundaries of the histogram and removing some extreme pixels. That is, if the histogram is normally distributed. Which many times is not the case, and therefore the following equation shows a way to calculate these boundaries:

     $$I_{min} = S.sort[S.sort.index(a) * 0.1\%] \tag{2.13}$$

     $$I_{max} = S.sort[-(S.length - S.sort.index(a)) * 0.1\%] \tag{2.14}$$

     Where

     – S - set of pixels
     – S.sort - set of pixels, but sorted in ascending order
     – S.sort.index(a) - index of the mode in histogram distribution
     – S.sort[x] - pixel in the number x position on the sorted set of pixels

   - **Calculation of O_{min} and O_{max}:** Since the image is likely to be overloaded with green and blue radiance, the calculation of the desired minimum O_{min} and maximum O_{max} intensity levels need to be calculated also in a channel and image sensitive way.
     O_{min} can be easily obtained using the standard deviation values of the Rayleigh distribution:

     $$O_{min} = a_\lambda - \sigma_\lambda \tag{2.15}$$

     Where a is the mode in a channel, and λ indicates the channel, and σ_λ is the standard deviation. The minimum value we want for a desired range needs to be between zero and $I_{\lambda min}$. In order to obtain the O_{max}, we need to maximize the haze-free image, which can be obtained like it was discussed before inside the Removal of Water Scattering topic, finally obtaining the following expression:

     $$O_{\lambda max} = \frac{I_\lambda}{kt_\lambda} \tag{2.16}$$

     Where:

     – k is an experimental value, for example 1.1 for red and 0.9 for blue and green channels.
     – t_λ is obtained using the following expression:

     $$t_\lambda(x) = Nrer(\lambda)^{d(x)} \tag{2.17}$$

     Where d(x) is the scene-camera distance, and,

     $$Nrer(\lambda) = \begin{cases} 0.80 \sim 0.85 & \text{if } \lambda = 650 \sim 750\mu m \text{ (red)} \\ 0.93 \sim 0.97 & \text{if } \lambda = 490 \sim 550\mu m \text{ (green)} \\ 0.95 \sim 0.99 & \text{if } \lambda = 400 \sim 490\mu m \text{ (blue)} \end{cases} \tag{2.18}$$

4. **Relative Global Histogram Stretching**
   In the majority of shallow water images, the values of red light intensities in histograms ($[50, 150]$ are distributed differently than the green and blue channels ($[70, 210]$). So, using the traditional histogram stretching algorithm equally in all channels will result in inaccurate results since the channels require different amounts of stretching as they aren't equally distributed. In order to create a channel sensitive histogram stretch, the following equation is used, named relative global histogram stretching [52]:

$$p_{out} = (p_{in} - I_{min})(\frac{O_{max} - O_{min}}{I_{max} - I_{min}}) + O_{min} \tag{2.19}$$

   Where:

   - p_{in} and p_{out} are input and output pixels;
   - I_{min} and I_{max} delimit the stretching range;
   - O_{max} and O_{min} delimit the desired stretching range;

5. **Bilateral Filter on RGB Channels**
   After performing the previous operation, we need to apply a bilateral filter to remove noise from the image.

6. **Conversion into CIE-Lab color model**
   This step converts the image into CIE-Lab color model, and also marks the beginning of the color correction process. This way we have more parameters (L a and b) to tune the color with, in order to improve color performance. L adjusts the brightness [0,100], *a* adjusts the red and green intensities [-128,127], and *b* adjusts the blue and yellow intensities [-128,127].

7. **Adaptive Stretching of *L*, *a* and *b* components**

   To the L component, a linear slide stretching [52] is applied to stretch the range between 0.1% and 99.9% to [0,100]. The values above 99.9% are set to 100 and below 0.1% are set to 0. To stretch *a* and *b* is used an S model curve [52]. With this, the color correction is completed, and we can convert back to the RGB model.

8. **Conversion into RGB Color Model**
   In this step we convert the CIE-Lab color model image back into RGB model.

This concludes all the steps for performing Relative Global Histogram Stretching, the most extensive algorithm regarding underwater image enhancement in this dissertation. As we will see in **Chapter 5**, this doesn't mean it will bring better results than other filtering algorithms discussed previously. Hence, by understanding the various filtering processes we came up with something better for our type of images. The efficiency of a filtering algorithm depends greatly on the type of image we're dealing with, which depend on the camera and environment. This realization also came by understanding the results in **Chapter 5**.

Chapter 3

Frameworks and Design Methodologies

3.1 PYNQ - Python Productivity for Zynq

PYNQ, or Python Productivity for Zynq, is an open-source project from Xilinx, with the purpose to provide Python programmers an user-friendly platform to work on Xilinx FPGAs with Zynq processors. In order to do so, the Python environment is designed to run inside a Ubuntu-based filesystem, with a Petalinux kernel, the high level diagram for the framework is depicted in **figure 3.1**. The operative system's image is stored in an SD card used as a boot on the targeted FPGA. The Python environment can then be accessed by connecting to the board's IP address via jupyter-notebook. It runs on Zynq-7000-based processors, where the Zynq-7000 consists of a dual-core ARM A9 processor. As this processor comes within Xilinx development boards like PYNQ-Z1 and PYNQ-Z2, it is used in combination with programmable logic (FPGA) that usually works as an accelerator for Zynq. For this, Xilinx provides Python packages to interact with the hardware modules by downloading bitsreams to configure the FPGA. These bitstreams however, are called Overlays and they represent the result of a compiled hardware module using Vivado. So, in order to develop them it still requires engineers with the expertise to do so. These Overlays can be called as a python function, thus allowing them to be used multiple times inside a program. Not to mention, they run on the FPGA, in parallel with Zynq. A block diagram representing this processing system is depicted in **figure 3.2**.

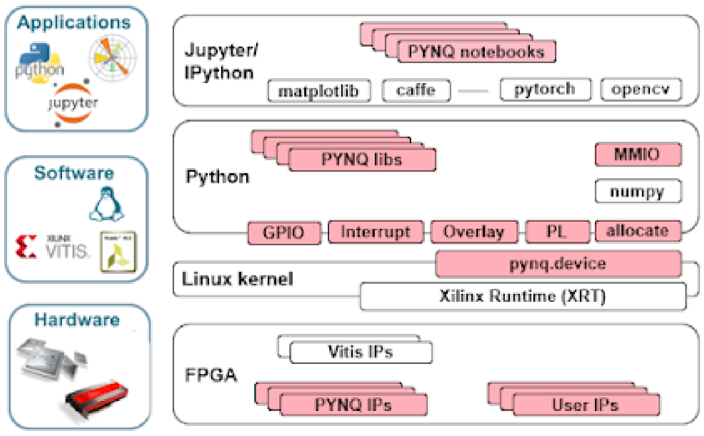


Figure 3.1: PYNQ framework diagram [2]

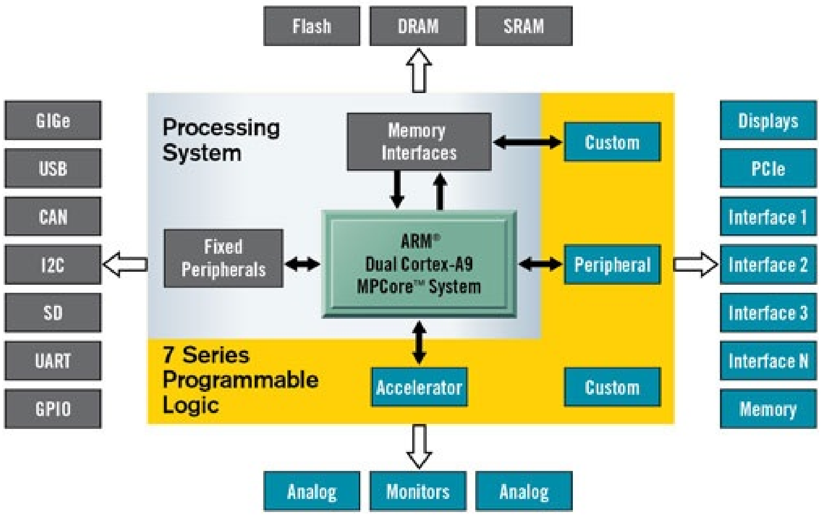


Figure 3.2: Zynq-7000 processing system block diagram [2]

3.2 FINN - Fast, Scalable Quantized Neural Network Inference on FPGAs

FINN refers to the **FINN compiler** (which maps Quantized Neural Networks to FPGA architectures) and the **FINN project** in general with all the other tools involved to reproduce the steps represented in the diagram at **figure 3.6**. There are two approaches to QNNs: **Quantization-Aware Training** and **Post-Training Quantization**. **Post-Training Quantization** means that we only quantize the network after training, with common frameworks like Pytorch or Tensorflow with no quantization during training. Quantization-Aware Training focuses on training a quantized architecture, using quantized activation functions, quantized inputs and ouputs and almost everything stated in **Chapter 2** regarding CNNs, therefore creating a quantized model and weights from scratch. The following text about the FINN compiler is an aggregated summary of what is presented in [57], [58] and [59]. In order to follow the work flow discussed below, we recommend taking a look at **figure 3.6**. The original documentation present on the website [57] doesn't fully explain the whole process, but this knowledge can be complemented by visiting the source code (linked on the website [57]) and information in FINN related articles [58], [59], [10] and [11].

3.2.1 Quantization Aware Training and Brevitas Export

Quantization Aware Training

Though Brevitas is a research project and not officially a Xilinx product, it is an essential step, using it in order to have a quantized FINN-compatible model. Brevitas [60] is a PyTorch research library for quantization-aware training. Two different types of quantization are represented in the **figure 3.3** where **Q** represents the quantized domain and **r** the continuous domain. In Uniform Quantization we always have the same distances between quantized values. The quantization concept is depicted in the figure below, during quantization, unquantized values are mapped into lower precision values.

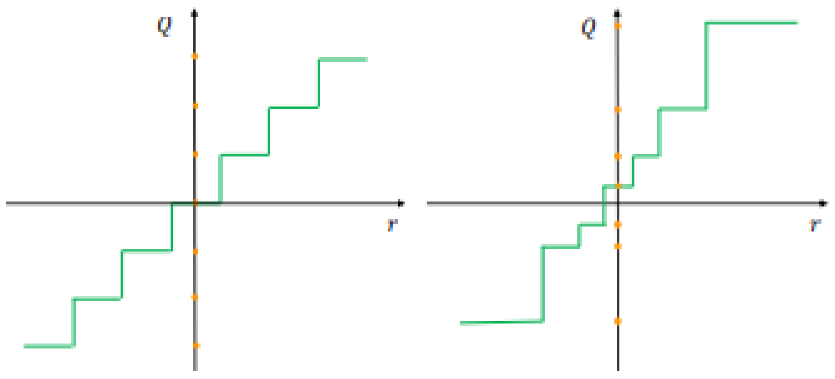


Figure 3.3: Uniform (left) vs Non-uniform quantization (right) [61]

Brevitas is loaded with features providing multiple options for quantized neural networks, such as selecting between fixed point and floating point activations, bit widths inside layers, input and output quantizations, and much more. For instance, as it is done in Pytorch, instead of using *torch.nn.conv2d* for a convolutional layer, we use *brevitas.nn.QuantConv2d* for a quantized 2d convolutional layer receiving as input parameters defining what kind of input, weight, output or bias quantizations we want. The type of quantization supported is **uniform quantization**.

Brevitas Export

FINN uses ONNX models as input. ONNX - (Open Neural Network Exchange) is used by FINN as an intermediate representation for neural networks. The model is first trained using Pytorch and Brevitas, and then exported to ONNX using FINN python libraries. It can also be exported as ONNX using other methods, but it won't produce a ONNX compatible model with the FINN ModelWrapper. After obtaining the ONNX, the model is loaded into a ModelWrapper and even though the model was created using FINN libraries, not every architecture is compatible and can be loaded into the ModelWrapper. The ModelWrapper is a tool used to analyse and manipulate ONNX models with helper functions, like setting tensor shapes, data types and initializers. After training a quantized neural network with brevitas and exporting it to an ONNX model, we can advance to the network preparation step.

This step is the backbone behind the connection between the ONNX model and hardware. The model is prepared by the ModelWrapper in order to be ready for several transformations, that will convert the ONNX nodes into custom nodes that correspond to FINN's HLS functions in the further steps. Before the Streamlining Transformations represented in **figure 3.6**, we need to run **Tidy-up transformations** which are also applied in between steps during the network preparation, as a post-processing routine. These transformations ensure that nodes have unique names, named tensors, etc.

Streamlining Transformations

The streamlining step consists in erasing floating point operations, with methods stated in more detail at [62]. Although we already perform quantization aware training, there are residual floating point operations in the QNN's forward pass that will increase the memory footprint and latency on devices that are not optimized for floating point operations. So, the QNN needs to be streamlined first, creating some custom nodes in the ONNX model in addition to the standard nodes it had before. Only after streamlining, the ONNX can be converted to FINN's HLS layers.

Convert to HLS Layers

In this step, the streamlined model is converted to a combination of HLS-layers which represent a combination of HLS and verilog modules. For example, a common operation on the streamlined model: *XNORPopcountMatMul* represents the dot product between matrices (very common specially in CNNs), and in this step it converts to a *MatrixVectorActivation* layer, that can be implemented in different modes.

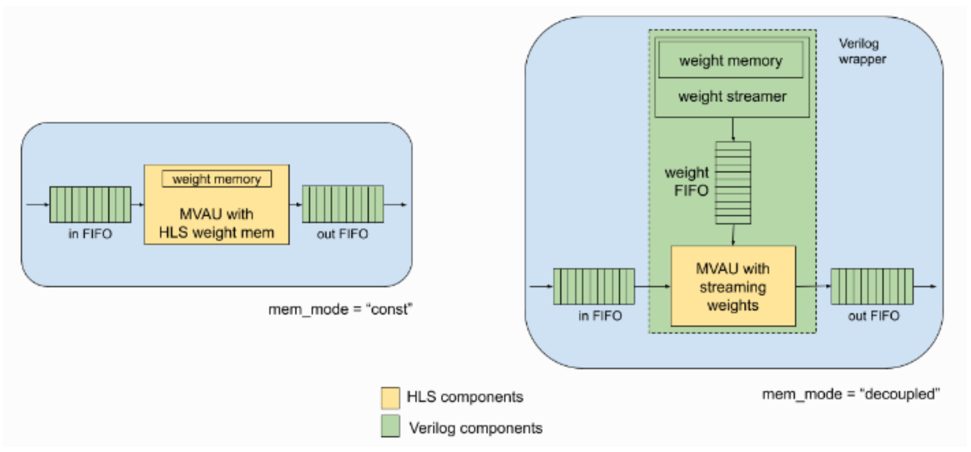


Figure 3.4: *Const* and *Decoupled* modes for MatrixVectorActivation layers, where MVAU is a Matrix Vector Activation Unit [57]

Here, the *const* mode uses less HW resources but can result in poor allocation decisions by Vivado HLS and long synthesis times for array shapes that aren't in a memory friendly format. While *decoupled* mode isn't as used and tested as *const* mode (hence offering issues during this step of network preparation) it provides better control over memory primitives and removes the weight array shapes from HLS synthesis (weights are declared in .dat files instead of being inside the HLS block). But it requires one more port in the MVAU, a weight streamer, and a weight FIFO, and since these layers will be widely used, using *decoupled* memory mode will create a bigger toll in HW resources. Note that there was no hardware generation yet, only adaptations in the model in order to transform the representations inside the ONNX model into HLS layers.

Dataflow Partitioning

Dataflow partitiong step consists in seperating the ONNX graph in two: one containing HLS layers (further processed in the FINN flow) and one containing non-HLS layers.

Folding

To explain folding we first need to introduce the Matrix–Vector–Threshold Unit (MVTU), which is the computational core for FINN accelerators. The vast majority of compute operations in a BNN are conducted in MVTUs, and can be expressed as

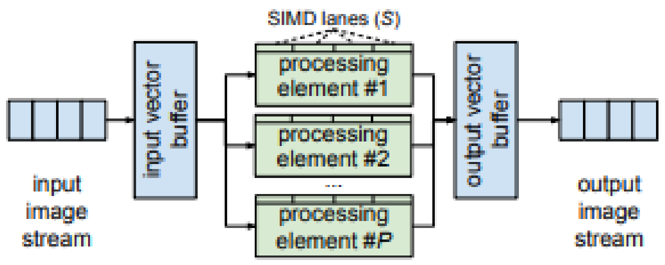


Figure 3.5: Overview of a Matrix Vector Threshold Unit [59]

matrix–vector operations followed by thresholding. An overview of the MVTU architecture is in **figure 3.5**.

As stated in [59], each PE (processing element) corresponds to a hardware neuron and each SIMD (single instruction multiple data) lane acts as a hardware synapse. If we set up a MVTU with a number of hardware neurons and synapses equal to the number of neurons and synapses in a BNN layer, this would create a fully parallelized layer. By scaling this to a whole CNN, the accelerator could classify images with a few clock cycles. However, this is impossible to achieve for bigger CNNs since amount of hardware resources on an FPGA is limited, and it is necessary to **fold** the BNN onto fewer hardware synapses and neurons.

By selecting PE and SIMD values in the FINN compiler we can adjust how much folding we want. Setting both values to 1 will create a network with only HLS layers and maximum folding. In computer architecture branch folding improves perfomance by predicting certain branches and removing them from the instruction stream. Here, the PE and SIMD assigned values are input parameters to the layers in the folding step, thus, selecting parallelism for each layer. More about folding, see [58] and [59].

This way, with the previous HLS layers partitioned during the previous step, and the new HLS layers, we have a network of only HLS layers, ready to advance to the hardware build phase.

3.2.2 Hardware Build and Deployment

A model composed only by HLS layers can be processed by the FINN compiler, which generates a bitfile and a driver, both can be imported by PYNQ running on a Zynq-FPGA device, thus making use of the designed FPGA accelerator, in this case running the ONNX model forward pass, which is an adaptation from the previously trained quantized convolutional neural network.

Driver

The driver's purpose is basically data transferring. It runs on the Zynq processor, and it's responsible for packing the input tensors in the designed format, giving orders to initiate data transfer using PYNQ APIs, and finally unpacking the output tensors before transferring them back to the Zynq's associated DRAM. The first step of the Hardware Build is making this diver.

Next, we need to create additional nodes for the driver to operate, so in this next step

FINN inserts DMA engines as ONNX nodes into the flow graph (the ONNX flow graph is the graphical representation of the ONNX model, obtained through the ModelWrapper). These engines will be moving data into and out of the accelerator's DRAM by the driver's command. FINN also inserts data width converters between consecutive nodes if required (in case we need extra bits for intermediate operations, or have different quantizations in different layers).

Partitioning

In this step FINN separates the ONNX into *StreamingDataflowPartitions*. Each partition will become an IP block in the next step, with the respective DMA node inside. In case no number of partitions is specified, all of the IP blocks are placed under a single partition. But this is not recommended, since partitioning is an important step to facilitate compatibility with other FINN versions. To connect the streaming nodes, FINN

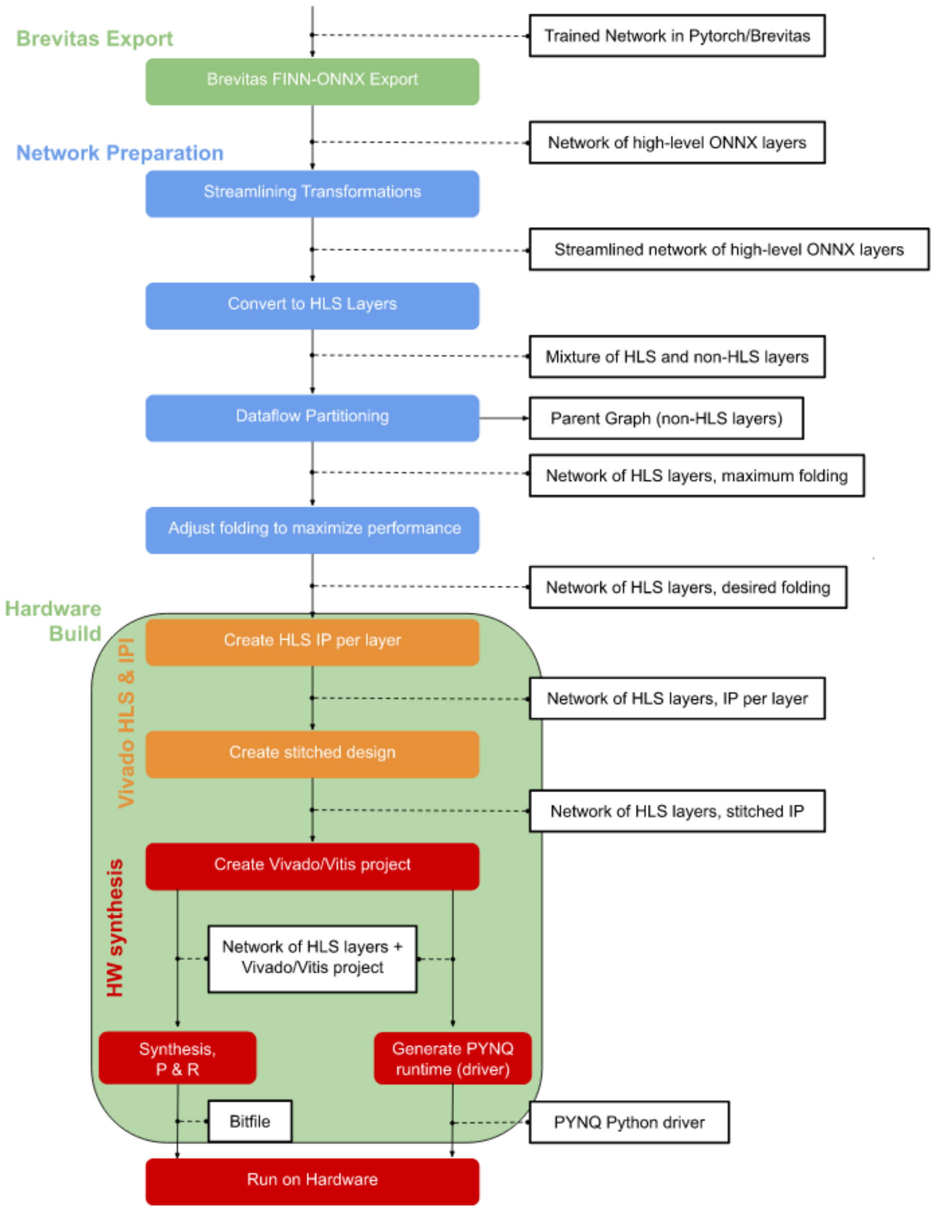

Brevitas Export
Trained Network in Pytorch/Brevitas
Brevitas FINN-ONNX Export
Network Preparation
Network of high-level ONNX layers
Streamlining Transformations
Streamlined network of high-level ONNX layers
Convert to HLS Layers
Mixture of HLS and non-HLS layers
Dataflow Partitioning
Parent Graph (non-HLS layers)
Network of HLS layers, maximum folding
Adjust folding to maximize performance
Network of HLS layers, desired folding
Hardware Build
Vivado HLS & IPI
Create HLS IP per layer
Network of HLS layers, IP per layer
Create stitched design
Network of HLS layers, stitched IP
HW synthesis
Create Vivado/Vitis project
Network of HLS layers + Vivado/Vitis project
Synthesis, P & R
Generate PYNQ runtime (driver)
Bitfile
PYNQ Python driver
Run on Hardware

Figure 3.6: FINN end to end diagram [57]

inserts FIFOs in between, where the size of each FIFO and input shape is stated by the node's attributes. After connecting all nodes, FINN creates IP blocks for each partition containing the IP blocks for each layer already connected with each other, since they're generated by Vivado HLS using FINN's transformations libraries. The top level modules are then stitched and generated in Vivado IPI. We can then create a Vivado project with the top module and generate the bit file after running synthesis.

3.3 Vitis-AI

With the same purpose as FINN, Vitis-AI creates an overlay for the ARM processor, working as an accelerator for inference in CNNs. This accelerator is called Deep Learning Processing Unit (DPU).

3.3.1 Deep Learning Processing Unit

The DPU is a group of parameterizable IP cores, pre-built and stitched together inside an SD card image, therefore also being inside a linux based operative system, similar to PYNQ, for FPGA booting. The DPU can be reconfigured to use different hardware resources inside Vivado, but as soon as the FPGA is booted, it becomes programmed with these IP cores, the hardware utilization stays fixed, and it becomes hard to follow resource utilization when configuring the DPU with different models. So it stays as a parameterizable accelerator, allowing one hardware build to run many different CNN models. DPU architectures differ from board to board, and Vitis-AI supports a restricted variety of platforms. In this dissertation we use the *Zynq UltraScale+ MPSoC: DPUCZDX8G* which has the high level diagram shown in **figures 3.7 and 3.8**.

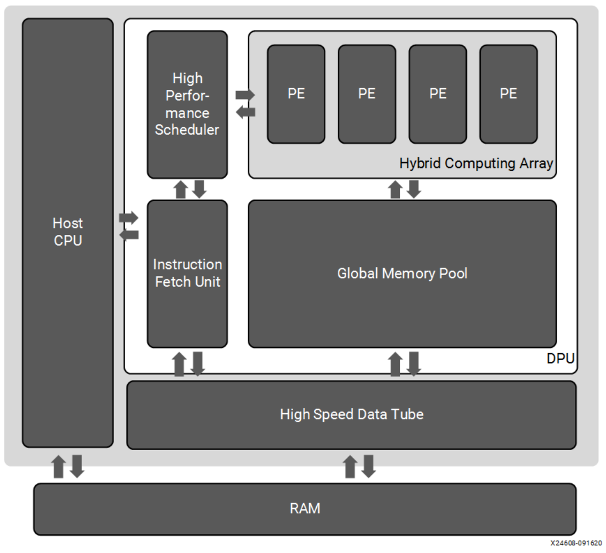


Figure 3.7: High level diagram of DPUCZDX8G [63] (PE stands for Processing engines).

Hardware Resources

Port description, register spaces and the like are described in the respective DPU product guide [64]. There are different DPU architectures, described in [64], in our case

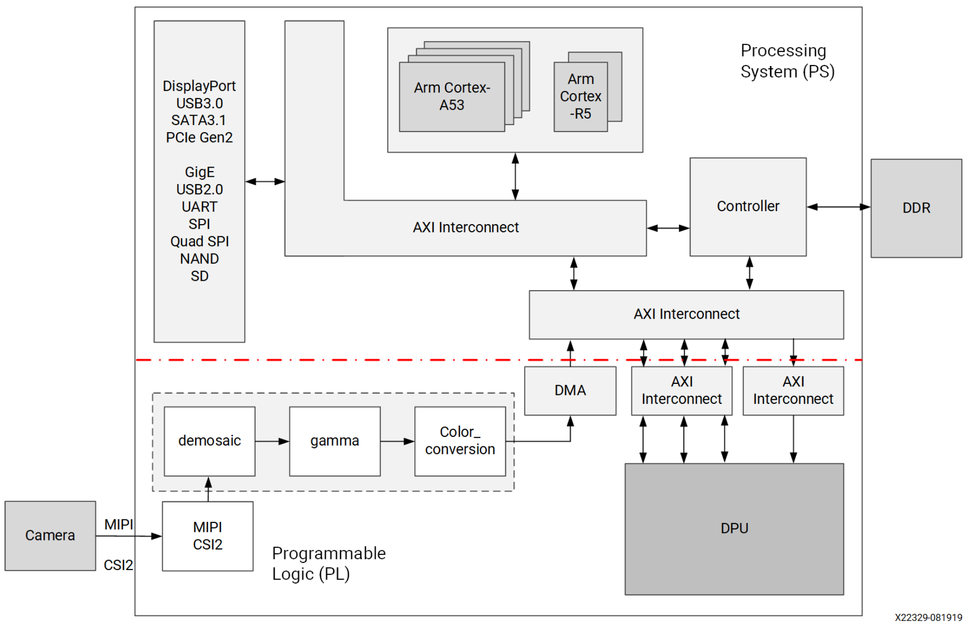


Figure 3.8: Example high level diagram of the processing system together with the programmable logic [63]. In our case we're not using a camera or external ports except the SD card.

we'll be using *B4096*. The type of architecture is chosen before compiling the project in Vivado. Changing the HW architecture means changing resource utilization, and with our configuration, we are using, **for each DPU core**, the resources in table **table 3.1**.

LUT	**Register (R/W 32 bit)**	**BRAM36K**	**DSP**
51351	98818	255	710

Table 3.1: Resources used by the DPU, for each DPU core.

As we'll be using two DPU cores, the overall resource utilization by the accelerator is twice as the ones stated in **table 3.1**, with a reference clock frequency for the XRT (Xilinx Runtime) of 300MHz.

Clocking

There are different clock domains inside the DPU, as shown in **figure 3.9**.

The register configuration module receives the DPU configuration through AXI interface (also depicted in **figure 3.8**). This module can operate at a lower frequency from the rest of the modules, since these registers aren't meant to be updated as often, or set to be the value as m_axi clock. The m_axi clock is the standard clock running through the processing system, used for the data controller inside the DPU. The data controller module schedules data flow between DPU and external memory. The arithmetic calculations inside the DPU run at a different clock rate, set to be twice the frequency used in the data controller (in this case, 600MHz).

Vivado IP schematics for clocking are provided in **Appendix A.1**, where we use two clock outputs in the clocking wizard for each DPU core. The respective schematic

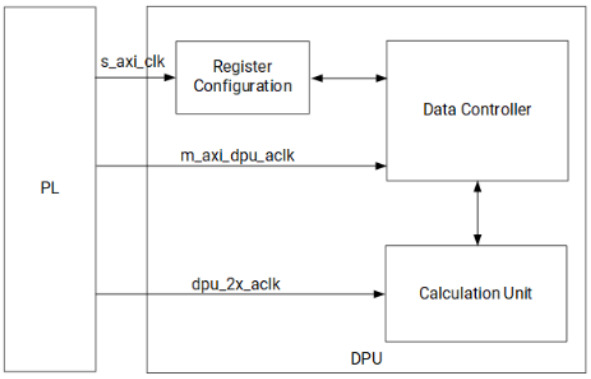


Figure 3.9: Clock domains inside the DPU [64].

in the appendix also includes the reset signals, which have to be synchronous with the clock domains discussed before. The dpu_clk_wiz module outputs clk_dsp1 and clk_dsp2 (600MHz) which are twice the frequency value from clk_in1 and clk_dpu (300MHz). The missing connections in the CPU and DPU are for master-slave interfaces. Looking closely, one can notice that the DPU IP is using three cores instead of two, which is only for a matter of representation since we're using the schematics from the product guide [64] as an example. The connections with the processing system via master-slave interfaces are in **Appendix A.2** which also includes an interruption handler module. The recommended parameters for each module are provided in the product guide [64], but one can freely change them for different applications.

3.4 Vitis-AI vs FINN

This section provides an overview of differences between FINN and Vitis-AI. For a quick description of pros and cons of using FINN or Vitis AI, take a look at **table 3.2**.

Note that Xilinx continues to issue new versions for both Vitis AI and FINN. After an extensive effort using FINN tools, there weren't any actual findings that could be used to draw any inferences. The tools are not yet stable, and only very simple accelerators were built during the development of this work, for instance, the best achievement was an accelerator for some FC layers (MLP).

Unfortunately, for more complex architectures like YOLO it requires large amounts of time and effort to achieve a functional accelerator. Hence, since the DPU from Vitis-AI is a pre-built accelerator, many issues during the compilation are avoided. Using a DPU to run a few layers of MLP like architectures would be a great misfit for its hardware resources. For this reason, a fair comparison between accelerators built with FINN and Vitis-AI was not achieved in this work.

There was a single work however, where this was made possible [8], where it is stated that FINN achieved lower resource utilization since it supports 4 bit quantization (Vitis AI uses 8 bit), however it achieved lower frame rate than Vitis AI, and 8 bit quantization in FINN overloaded the board's HW resources. In conclusion, FINN would be better suited for low power FPGAs with more humble HW resources, and it remains a potential solution for the objectives stated in **section 1.3**.

A quick detail about quantization aware training is, that for both Vitis AI and FINN, only worked correctly when training with CPU, since these quantized training processes has to occur inside a docker container, and only CPU compatible dockers were available for the docker versions required by the respective framework. This became a downside for FINN that resorted mainly in quantization aware training.

	FINN	**Vitis AI**
Concept	Builds an accelerator for a given model. Hardware resources vary from model to model.	The accelerator is already built, and needs to be configured with an xmodel.
Deployment	Generates a bitfile to be loaded as a PYNQ overlay	Generates a xmodel file that configures the DPU to run a certain model
Supported Platforms	Any Zynq devices, including low power FPGAs like PYNQ-Z1 and PYNQ-Z2	Supports limited platforms, usually FPGAs with more HW resources like Ultrascale+ series or Alveo boards.
Energy Efficiency	May provide superior energy efficiency since it supports low powered FPGAs and builds accelerators for specific case scenarios.	Due to restricted accelerator architectures, configurations, and supported platforms, may provide worse solutions than FINN when it comes to energy effiency.
Quantization	Widely used with Quantization Aware training	Widely used with both Quantization Aware Training and Post Training Quantization
Ease of use	Requires and engineer with some experience and expertise to operate, needs user interaction in all stages of the workflow defining when and how parallelization and quantization occur, and how to tune all the steps to refine the model until it generates a bitfile.	The customization of parallelization and quantization methods is optional, there is a default configuration to use as example. Diving in extensive low level flow is not required. Easier to use since it requires less expertise.
Documentation, Support, and Community	Released in 2017, the community is no longer as active as it was, and devs dont show much activity. Documentation usually only provides an overview of the project. Some tutorials and examples no longer operate correctly.	Released in 2020, with recent dev and community activity, many problem-solving threads in github issues forums. Widely documented. Ease of access to low level flow, All examples/tutorials are operational.
Frameworks for model training	Pytorch and Brevitas.	Pytorch, Tensorflow 1.15 and Tensorflow 2.0. Works better with Tensorflow 1.15.

Table 3.2: Pros and Cons of using FINN and Vitis AI

Chapter 4

Implementation Description and Methods

This chapter describes implementation methods in order to make it possible for other people to reproduce similar results.

4.1 Dataset Preparation

The raw data downloaded from [13] (brakish dataset) needs to be adapted in order to match the format of the datasets used in our YOLO model.
The images come from captured videos of underwater environments, so we need to extract the frames using the given source code in order to extract at the exact frame rate as the developers used, since we'll be using their annotations purposely made for each extracted frame.
In order to do so, we need to install ffmpeg software tool [65], used to extract the frames. To run the dataset's helper scripts we used python v3.7.0 and the only required package is *ipdb* == 0.13.9. After installing the requirements, one needs to follow the steps:

1. Run the python script *"frameExtractor.py"* which is inside the directory extracted from [13], and extract frames for all classes, resulting in 15085 images.

2. After obtaining the frames, the annotations also need to be adapted. The type of annotations used in this work is YOLO style annotations. There is a yolo annotations folder, but they need to be addapted. They are in the right format (class, x, y, width, height). The classes aren't numbered correctly since they should be labeled whithin the range of 0 to 5 instead of 1 to 6. And the values of x, y, width and height should be normalized since we'll use resize, rotate and filtering type of augmentations (the size of the image will fluctuate).

4.2 Training algorithm overview

Requirements for the setup at **Appendix section B.2**. The source code was adapted from the assignments of deep learning classes issued in this degree. The original code was made to detect cars from the kitti dataset [66] and used YOLOv3 architecture. The adaptations consist in changing the input files, fixing outdated libraries, and editing pre-processing related functions that weren't working properly. The architecture used, in this case, the **cfg file for YOLOv3-Tiny**, which can be downloaded from the author's website [43].

Pre-processing

The pre-processing consists in, during training, applying transformations to the loaded batch images. Such as:

1. Padding to a square/resizing to a shape varying between 288x288 and 448x448 (when running testing and detection algorithms we use images with size 416x416 only). The shape is chosen randomly between the ones that allow separation in grid cells, like it was explained in **section 2.2**
2. Flipping the images horizontally, therefore creating a symmetric image (the images to be flipped are chosen randomly).

Random flips and resizes can be considered as augmentations to improve accuracy.

Hyper parameters

A detailed description about hyper parameters related to the architecture, such as kernel sizes, channel depths, strides, padding and activation functions at convolutional layers, can be found in the **Appendix Chapter C**.

- **Batch Size:** 16
- **Learning Rate:** 0.001
- **Weight decay:** 0.0005
- **Momentum:** 0.9
- **Epochs:** 500

This set of hyper paremeters already comes with the cfg file with the exception of batch size and the number of epochs. The achieved MAP values on the COCO dataset are around the same 33% achieved by the author.

4.3 Filtering

Requirements for the setup at **Appendix section B.1**.

As it was said before, we applied different filtering algorithms on the dataset, producing some interesting results discussed in **Chapter 5**. The RoWS, CLAHE and GC filtering algorithms were adapted from [67]. For CLAHE, the opencv library already has a function to produce a CLAHE image by giving the tile grid size and clip limit as input parameters. GC on the other hand, was done by applying directly **equation 2.3**. Hence, CLAHE and GC are easily obtained in less than 10 lines of code. As for RoWS and RGHS, they are more complex and have a mix between opencv and the equations described back in **section 2.3**. RGHS is also avilable at [67], but it contains many mistakes in the color restoration and relative global histogram stretching steps. So, the RGHS algorithm was fixed by taking advantage of the setup in [67], and re-coding in python the equations in the article [52].

4.4 PYNQ

The FPGA will only use the detection algorithm, but just to be safe, we also did run the testing algorithm in order to be certain that the metrics stayed the same. Since there is a different operative system and python environment running inside Zynq, the packages' versions also change. The detection algorithm had to be adapted and several changes were conducted in order to satisfy all dependencies. The package requirements are at appendix chapter **PYNQ packages - ZCU 104**. To run the detection algorithm using PYNQ and CPU only (in the Zynq processor), we installed PYNQ on the Xilinx product *Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit* [14].

4.5 Vitis-Ai Workflow

Vitis-AI provides alternative work flows from the one followed in this work. **Figure 4.1** has an overview diagram representing different solutions to program the DPU. Vitis AI also supports Caffe models. To train the model, the same setup as before for PyTorch training

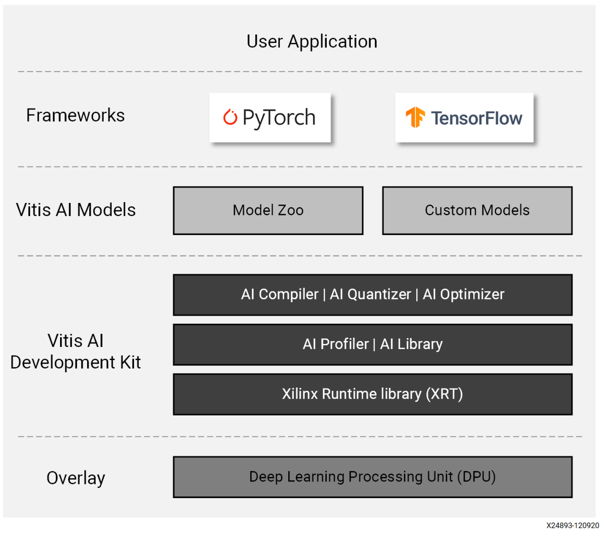


Figure 4.1: Vitis AI workflow diagram, from vitis AI user guide [63]

was used. The quantization (in our case, post training quantization) and compilation of the model occurs inside a docker container (for more information visit [63]), however, the desired DPU architecture needs to be built inside Vivado with source files and instructions from [64] (this step can be skipped if using the demo image from Vitis AI tutorials [63]). The docker image comes with different environments for different frameworks (pytorch, tensorflow, etc). Compilation and quantization scripts should be adapted (in case of using a custom model) from [68], which are written in python and are easily understood. The DPU does not support certain operations, and in the case of YOLOv3-Tiny architecture, there is a max-pooling layer of stride 1 which is not supported. Since it has stride 1, it doesn't decrease the number of parameters in between convolutional layers, and can be just commented out without causing any concern (changing the architecture also means training the model again from scratch).

The workflow followed in this work however, was:

1. Training the model with Pytorch, and saving it as a Darknet model.
2. Convert the Darknet model to Tensorflow frozen graph model.
3. Setup the Vitis Ai docker container with compatible DPU architecture and quantization/compilation scripts.
4. Quantize and compile the model.
5. Create a prototxt file for the custom model (Vitis AI SD card image has scripts compatible with compiled Caffe models). Send the model files via SCP to the board.
6. Follow examples from Model Zoo compiled models, and write C++ code to build executable files to run the model in DPU+CPU.
7. Use xdputil tool to measure performance.

It was said before that Vitis AI supports Pytorch, so why the conversion to tensorflow frozen graph model?

The Vitis AI docker environment for Pytorch is not currently compatible with Darknet architectures, and many quantization errors ocurr when the quantization script reaches YOLO Layers (bounding box outputs). The only way around this would be separating the architecture in two, and run the YOLO layers inside the CPU, since we have two YOLO output layers in YOLOv3-Tiny (if using YOLOv3 there would be three in total). Therefore creating a lot of performance loss due to increased latency, by sending and receiving data between DPU and CPU to run the second part of the model. As a work-around, there are tutorials for YOLOv4 in [68], that can be addapted to YOLOv3, and all other versions before YOLOv4, which runs inside the tensorflow 1.15 docker container. Hence, the conversion to TensorFlow frozen graph model (converted using tensorflow 1.15) is a required step to keep high performance.

4.6 FINN

Regarding to FINN, since it mostly supports Quantization Aware Training, all the process is done inside a rootless docker container, and the workflow is pretty much summarized in the previous chapter. All of the steps in **figure 3.6** are done with FINN, Brevitas and Pytorch frameworks using python language. FINN uses an Vitis HLS library and the HW generation is done through the command line, not providing a visualization of IP schematics like it is done in Vitis AI. FINN is a very promising and interesting tool, but still needs a more stabilized workflow, functioning examples/tutorials, and updated user guides.

Chapter 5

Results and Discussion

This chapter presents and discusses the results that were obtained during the fulfillment of the objectives described in **section 1.3**. The objectives were met in a sequential manner until reaching the final goal. Therefore, as the first step of finding a good architecture was already described in chapter 2, and the fourth step of finding good tools to run model inference in the FPGA was already discussed in chapter 4, the subject of discussion in this chapter consists of:

1. Acquiring a filtering algorithm that is good for the brakish dataset (**section 5.1**).
2. Exploring the influence of filtering algorithms on CNNs (**section 5.2**).
3. Measuring power consumption in the FPGA and the Desktop computer, in both cases comparing the consumption between CPU, CPU+DPU (for the FPGA) and CPU+GPU (for the Desktop computer scenario)(**section 5.3**).

5.1 Filtering

Table 5.1 has low and high quality samples from the brakish dataset, each row with the result from a filtering algorithm, except for the first row with the original samples. The low quality of some samples can be caused by many reasons, such as the loss of camera focus, or environmental changes which produce water turbidity. The hazed effect of the underwater environment is still present in images with better quality, and becomes almost unbearable in low quality images. For instance, the low quality image has two crabs, in front of the box. A human being can hardly tell so, by only looking at the original images. But this becomes more evident after applying CLAHE.

Since the images from this dataset are captured with a flashlight pointed at the center of the image, they have increased brightness in the center, and become dark near the corners. GC was used only to slightly darken the image, using $\gamma = 1.2$ (see **equation 2.3**), this effect is more noticeable in low quality images. GC affects the whole image, so, using a high γ might cause a growth in the darker areas of the image.

Contrast correction with CLAHE was the most efficient method, using a 8x8 tile grid size, and a clip limit of 4. But, by using CLAHE to enhance sharpness we are sill lacking in color correction, as it's still affected mainly by green radiation.

This green tint didn't disappear in RoWS, a good reason for this might be that RoWS (and RGHS aswell) were designed to enhance the quality of underwater images that were affected by light scattering of sunlight. In this case, these algorithms might not work as

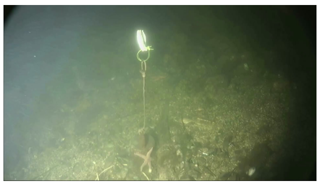
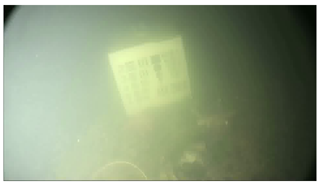
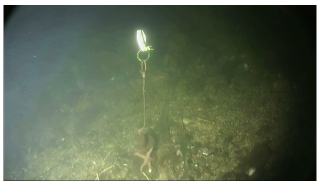
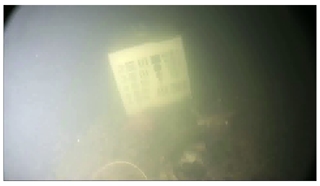
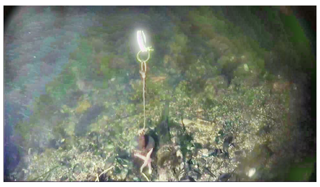
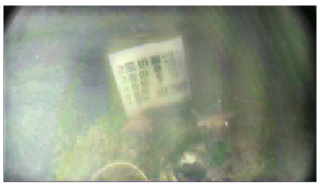
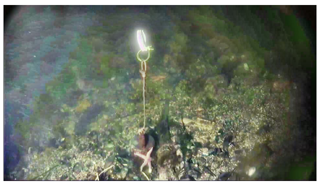
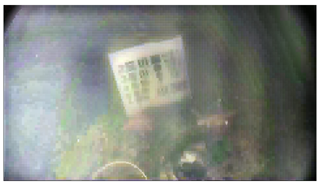
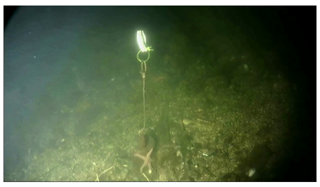
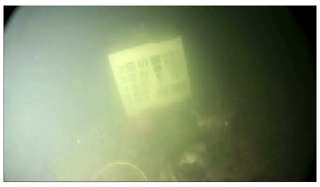
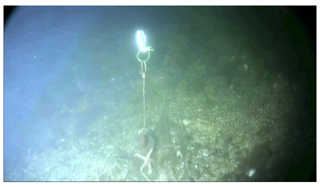
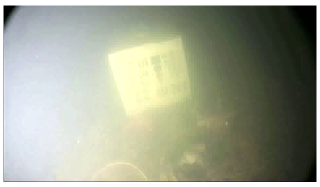
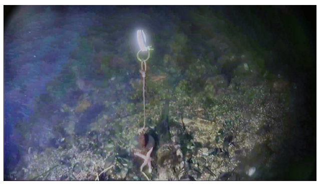
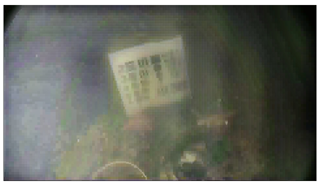
Filter	Better Quality	Low Quality
No Filter		
GC		
CLAHE		
GC_CLAHE		
RoWS		
RGHS		
Color Restored GC_CLAHE		

Table 5.1: Table of figures

well in the brakish dataset, since the image is captured in an environment illuminated by a flashlight, the science behind light scattering is not exactly the same. RoWS darkened the image a little and didn't improve much the image quality. Changing some parameters such as the radius and ϵ of the guided filter used in the refined transmission, didn't cause any noticeable changes in the image.

The light source is very close to the captured objects, so the haze effect and green color come mainly from the environment's characteristics, since there isn't enough space between the light source and the captured object for a notorious light scattering effect). Therefore, color restoration in RGHS might be a good solution for green color correction, after balancing the parameters θ_g and θ_b. The RGHS labeled images on table 5.1 are with the exact values from **equations 2.10 and 2.11**.

As for Color Restored GC_CLAHE, it possesses the original θ_g divided by a factor of 1.275 (experimental value) in order to compensate the excess presence of green color in the image, and θ_b was left as it is. This is a custom made filtering algortihm using CLAHE, GC and Color Restoration from RGHS, obtained by experimenting CLAHE and GC with RGHS. The first step is color restoration, with the previously stated modifications on the parameters θ_g and θ_b, to better compensate the green color. There is a noticeable glowing effect in brighter areas after applying CLAHE. The image is then darkened with GC $\gamma = 1.6$ in order to attenuate this effect, since experimenting with CLAHE's clip limit and tile size affected other important characteristics of the image. After GC, CLAHE is applied.

5.2 Metrics and Augmentation results for YOLOv3-Tiny

Table 5.2 shows the obtained metrics after training YOLOv3-Tiny with the brackish dataset, using 500 epochs and 16 images per batch. In spite of the architecture having only 24 layers, optimizer's steps are small, so the training usually takes between 1 and 2 days. The mAP only starts hitting values above 85% in the validation set roughly after 300 epochs, so, in an attempt to achieve higher values, the training was done with 500 epochs, in order give enough time for the optimizer to converge, with the risk of over-training the model.

While using the brakish dataset, over-training can't be detected through the provided testing set, since the images lack variety from the training set. Further in this section this issue is proven and solved with data augmentations.

The work done at [7] compares architectures of YOLO, introducing a modified version. But using only 100 epochs and a different optimizer (stochastic gradient descent) with adaptive learning rate. The new architecture introduced is based on YOLOv4-tiny and compared with version 3, version 5, and retina net, where the obtained mAP in their own architecture is 87.88% surpassing YOLOv3 by 8.68%. However, the registered mAP in the first 100 epochs, on the validation set, in this dissertation's work, only achieve around 77%, which makes sense since it is still inferior than v3.

One reason for such high mAP with the developed architecture in [7] with only 100 epochs, might be related with the used architecture allowing faster convergence, in spite of having an adaptive learning rate. Since YOLOv3-Tiny model hits better metrics than the proposed model at [7] (YOLO-UOD) when it is given a proper amount of epochs to converge, one can't conclude right away if YOLO-UOD would still hold the same metrical differences by using more epochs.

As for the work done at [6], used 150 epochs and resorted to a freezing / unfreezing training method. Freezing means selecting which layers are currently on training, for example in the first 50 epochs freezing certain parts of the network, and change the batch size when unfreezing. After unfreezing, in [6], the batch size is also of 16 and the otpimizes is Adam, like in this dissertation. But the comparison of architectures didn't include YOLOv3-Tiny. In [6] the architectures used are YOLOv4-tiny, YOLOv4 and a custom architecture, achieving 80.16% 93.56% and 92.65% mAP respectively.

Since the used GPU and CPU is different from both of these two works [6] and [7], the comparisons with frame rates are just values for curiosity standards, in this dissertation's case the achieved frame rate was 246,39 FPS with YOLOv3-Tiny using RTX-3060 GPU and i5-12400 CPU.

As for the YOLOv3-Tiny model size, also referred in [6] and [7], is 33,8 MB using pytorch style models (.pth), and 33,7 MB using Darknet style models (.weights). Which is bigger than YOLO-v4 Tiny and the custom model in [6], and YOLO-UOD in [7] by roughly 10MB.

Class	**AP**	**Precision**	**Recall**	**F1**
fish	0.977	0.946	0.984	0.964
small_fish	0.606	0.662	0.801	0.725
crab	0.966	0.951	0.988	0.969
shrimp	0.943	0.888	0.9824	0.933
jelly_fish	0.837	0.780	0.919	0.844
star_fish	0.991	0.981	0.994	0.987
		mAP: 0.886		

Table 5.2: Testing results with original brackish dataset (mAP values are not in percentage, therefore ranged between 0 and 1).

The brakish dataset is built from frames taken from videos in only two different places, both using the same camera angle, recording footages of different species.

This causes poor generalization for the trained models and make them sensible to small input variations. Which can't be concluded by using the originally provided testing set, since this testing set doesn't possess any variety from the training and validation sets.

It is composed of different images, but obtained from the same videos. A good way to verify this sensibility to variations in the input, is to slightly change the inputs and check if there is an accuracy drop. **Table 5.3** shows the resulting metrics obtained from running the testing algorithm on a GC filtered testing set, which only darkens the image by a tiny bit, as show previously in **table 5.1**.

With only a small variation in the input pixels, the mAP drops by as much as 14,1% (noting the accuracy drops on the jellyfish were bigger than the others, this might be caused by the lack of appearances of jellyfish in the captured videos, and are small in size, something that algorithms like YOLO usually struggle with). While testing the same procedure with CLAHE, which causes more radical changes in the image, the obtained mAP was close to zero (0.044 to be precise). The same goes for other filters that used CLAHE such as GC_CLAHE and the Color Restored GC_CLAHE. As for RoWS the mAP was 0.541 and for RGHS 0.386.

Compatible YOLO-style annotations are rare in underwater datasets, hence, instead of using a different dataset, the previously filtering algorithms that were explored with

Class	AP Precision	Recall	F1	
fish	0.874	0.667	0.919	0.773
small_fish	0.545	0.510	0.783	0.618
crab	0.845	0.842	0.899	0.870
shrimp	0.842	0.739	0.894	0.809
jelly_fish	0.476	0.603	0.612	0.608
star_fish	0.889	0.834	0.918	0.874
		mAP: 0.745		

Table 5.3: Testing results on GC filtered dataset, model trained on original brackish dataset.

the intent to use as a pre-processing method, were used as data augmentations for the dataset.

Augmentation Results

Using data augmentations improved the mAP to values rivaling the results obtained in [6], with a simpler architecture. **Table 5.4** has average precision values per class and per filtering method, obtained by using filtered testing sets. The augmentations model was warmed up with the weights obtained from the previous training with the original dataset, with also 500 epochs and 16 images per batch.

	AP							
	Unfiltered	**GC**	**CLAHE**	**GC_CLAHE**	**RoWS**	**RGHS**	**Color Corr. GC_CLAHE**	**All of them together**
fish	0.961	0.970	0.967	0.971	0.954	0.958	0.979	0.965
small_fish	0.740	0.722	0.693	0.682	0.708	0.712	0.714	0.702
crab	0.974	0.971	0.975	0.974	0.981	0.972	0.979	0.972
shrimp	0.956	0.914	0.943	0.945	0.978	0.971	0.916	0.944
jelly_fish	0.915	0.891	0.908	0.892	0.918	0.915	0.905	0.900
star_fish	0.995	0.997	0.988	0.992	0.997	0.992	0.982	0.991
mAP	0.924	0.911	0.912	0.910	0.923	0.920	0.912	0.913

Table 5.4: Testing results on filtered and unfiltered testing sets, model trained on augmented brackish dataset.

5.3 Energy Consumption

This section reports and compares energy efficiency from different hardware platforms, and in some cases in different scenarios. The following list serves as a quick introduction for the subjects to be covered in each of the following sections:

1. Power measurements of Xilinx's Ultrascale+ ARM processor in **section 5.3.1**. In this section the measurements are made with PYNQ libraries which read instant power values from the board's power rails at a fast sample rate. This section covers how the image quality and filtering affect power consumption.

2. Power measurements of Xilinx's Ultrascale+ ARM processor accelerated with DPU in **section 5.3.2**, running YOLOv3 Tiny model inference.

3. Measurements of a Desktop computer CPU, and CPU accelerated with GPU in **section 5.3.3**, running YOLOv3 Tiny detection algorithm.

The power measurements in **section 5.3.2** and in **section 5.3.3** were done by using the multimeter UT803 [69] connected via usb to a computer, registering current intensity values at a sample rate of 1 sample per second. The used voltage for the FPGA was 12V with a power generator. To measure the computer's power consumption, the same multimeter was used, by connecting it in series with a power supply cable, therefore measuring AC current rms values, and using 220V as the standard voltage value. A diagram of this setup is depiced in **figure 5.2**.

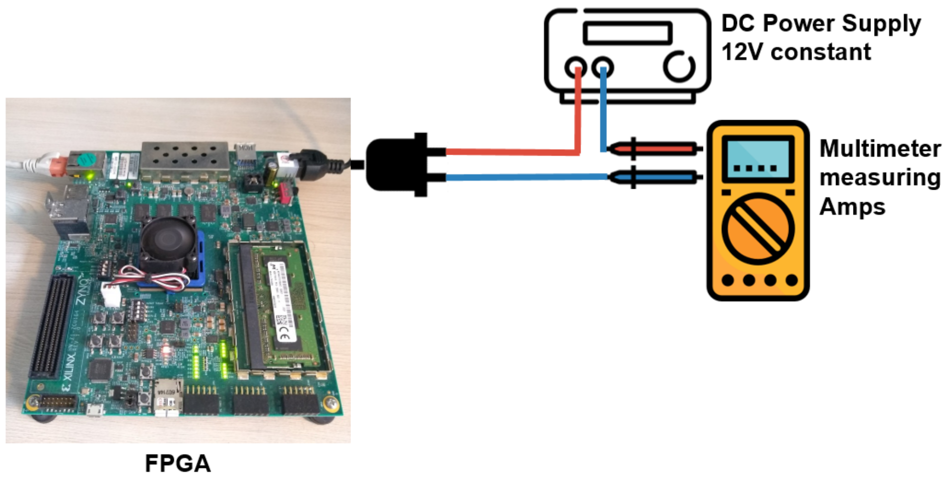


Figure 5.1: Diagram for Vitis AI power measurement [70].

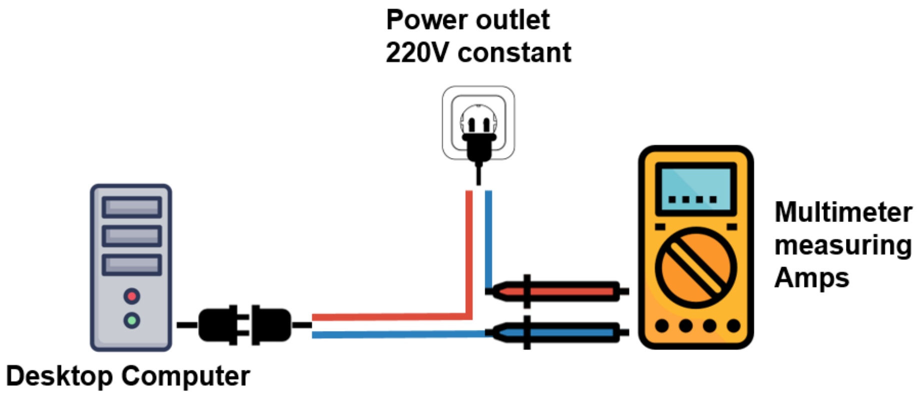


Figure 5.2: Diagram for computer power measurement [70].

5.3.1 Zynq Ultrascale+ CPU:

Energy values are calculated by integrating the power plot, which is made from discrete values, therefore using the Simpson's rule, with the scipy library (*scipy.integrate.simps*) [71]. **Figure 5.1** and **figure 5.2** have power plots with values obtained from readings from ZCU104's power rails. These measurements were made when using COCO and Brakish dataset respectively. COCO has variable input sizes, usually with a few hundreds of KBs. The frames from the brackish dataset have 9 to 10 KBs providing faster reading, writing and detection times, therefore consuming less energy.

The detection phase also represents energy consumption during model inference, using the same algorithm which can be accelerated by the DPU or GPU, hence, for matters of comparison between HW platforms, the ARM processor spends an average of 3.84 joules per frame during model inference at 1.88 FPS (when using the brakish dataset).

Table 5.5 has the power measurements and energy consumed for filtering algorithms studied in this work. To execute filtering algorithms the processor has to run operations for each pixel of the image. For more complex algorithms iterating through every pixel of the image several times. When comparing this process with model inference, the image only needs one pass through the model to obtain a result, and the model in cause ”only looks once”, therefore being fast enough so that the higher instant power being consumed justifies for the much shorter processing time. The most energy efficient filtering algorithm is CLAHE which spends nearly two times (in joules per frame) the energy spent by the model inference in **figure 5.2**.

	GC	CLAHE	GC_CLAHE	Color Corr. GC_CLAHE	RoWS	RGHS
Joule per Frame	27.861	7.511	30.177	267.861	772.824	738.256
Average Instant Power (W)	12.521	12.534	12.577	12.358	12.356	12.360

Table 5.5: Energy consumption for filtering algorithms in Ultrascale's Zynq+ processor

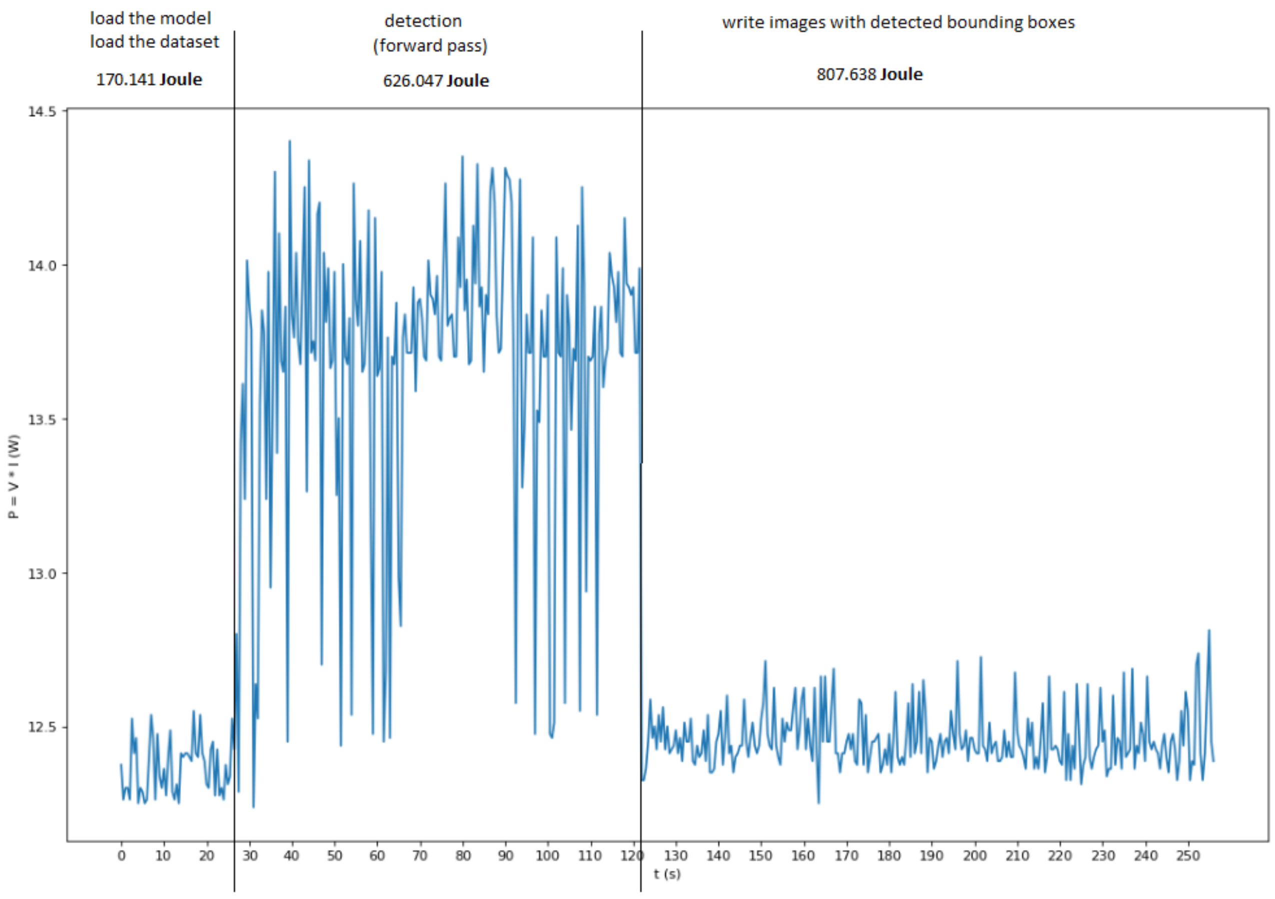


Figure 5.3: Energy consumption detecting 100 images from the COCO dataset, using *quad-core ARM Cortex-A53* (zynq+ processor) with PYNQ framework. Resulting in 1603.825 Joule in total, and an average of 16.038 Joule per frame.

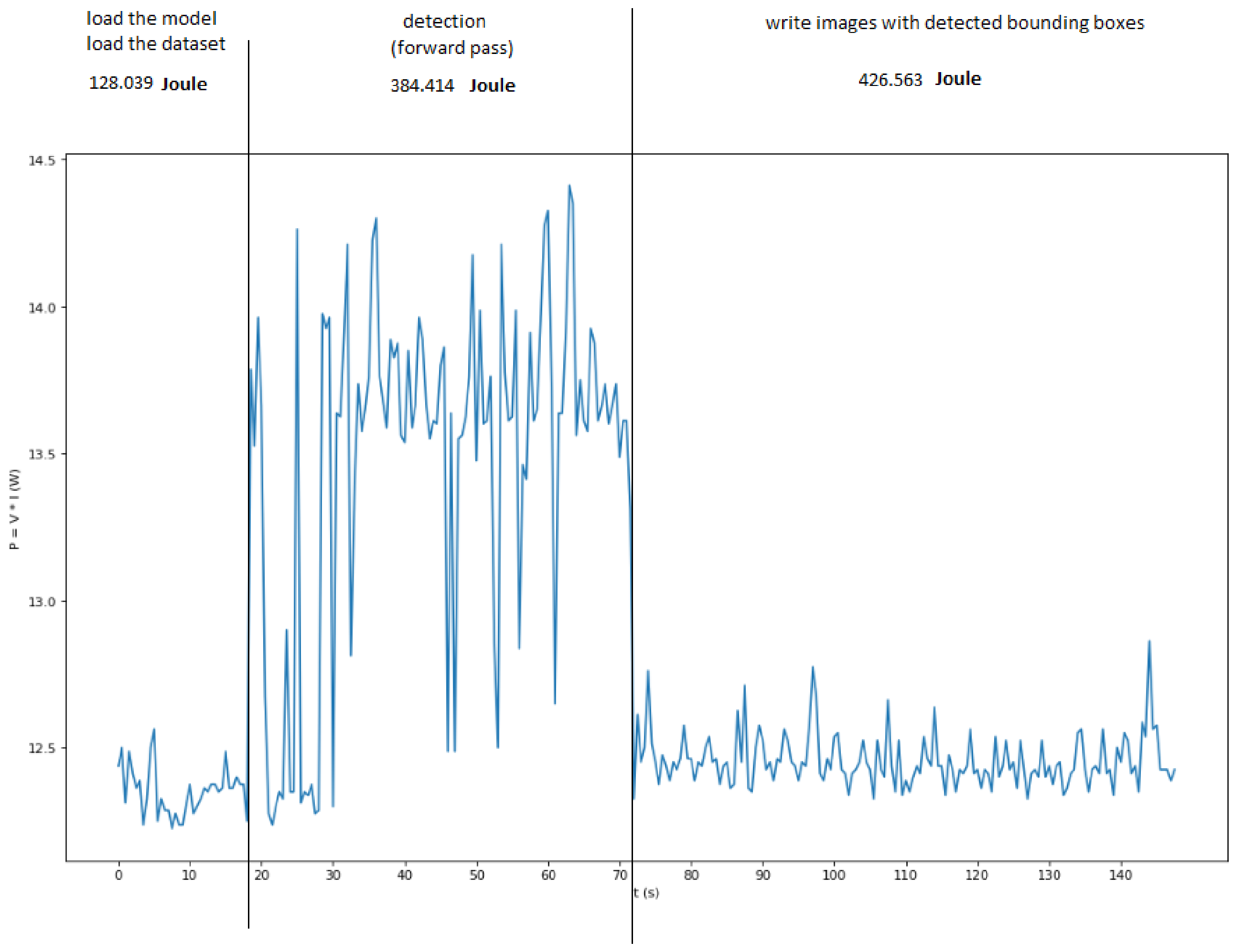


Figure 5.4: Energy consumption detecting 100 images from the brakish dataset, using *quad-core ARM Cortex-A53* (zynq+ processor) with PYNQ framework. Resulting in 945.250 Joule in total, and an average of 9.453 Joule per frame.

5.3.2 Zynq Ultrascale+ CPU + DPU:

In the DPU acceleration scenario, only one sample of the brakish dataset is used, and the FPGA runs the forward pass of that image in a loop for 100 seconds. Therefore only measuring model inference, since the time spent with loading or writing images is barely noticeable. In the first 100 seconds it runs the model inference, with an average instant power consumption of 21.938W, and in idle time spends about 17.35W. During 100 seconds it can run 7005 images, which means it has a frame rate of 70.05 FPS, and therefore 0.313 Joules per frame (calculating 21.938/70.05 since 1 watt is 1 joule per second).

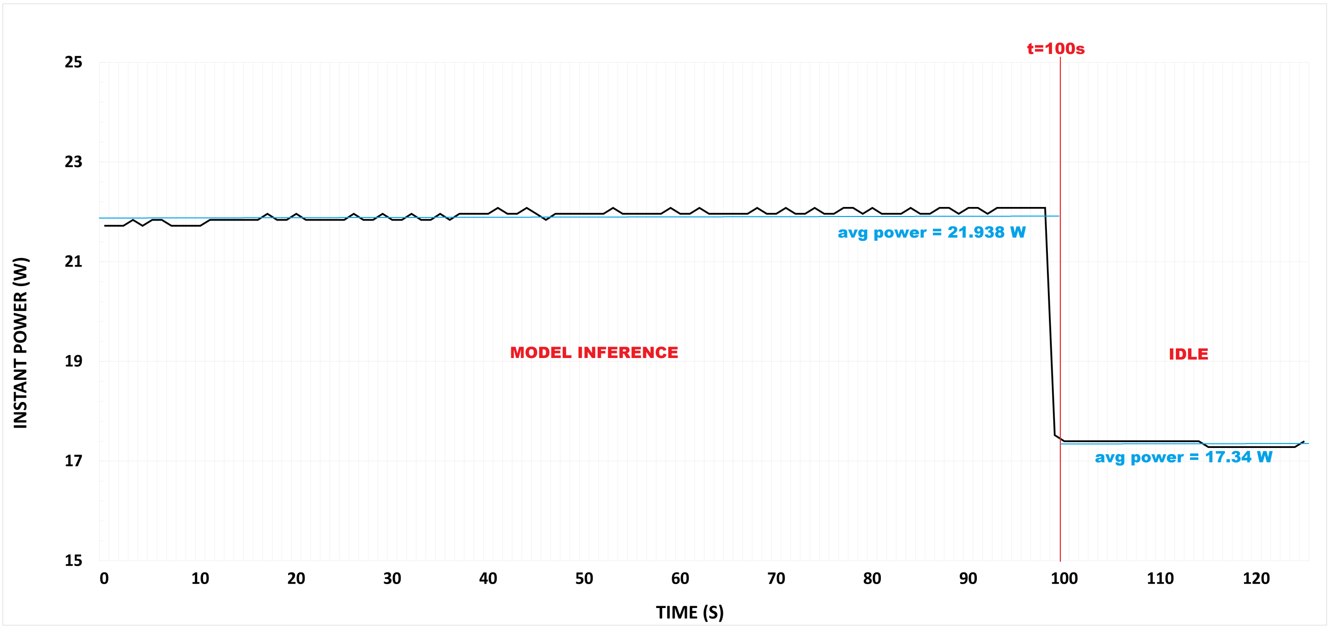


Figure 5.5: Power plot of *quad-core ARM Cortex-A53* processor accelerated with *DPUCZDX8G_ISA1_B4096* DPU running YOLOv3-Tiny model inference in loop for 100 seconds.

5.3.3 Desktop CPU and CPU + GPU:

When measuring power consumption in the desktop computer, the detection algorithm was made to run 1000 different images. The same algorithm was used in PYNQ to obtain the plots in **figure 5.1** and **figure 5.2**.

For CPU only, it took around 1.98 seconds to load the model and the dataset, and 62.13 seconds to run the detection phase, where the total inference time inside the detection phase was 59.54 seconds. During the detection phase it spent an average instant power of 141.92W. Resulting in an average of 16.79 FPS and therefore 8.45 Joule per frame.

For CPU accelerated with GPU, it took around 0.98 seconds to load the model and the dataset, and 20.75 seconds to run the detection phase, where the total inference time inside the detection phase was 4,05 seconds. During the detection phase it spent an average instant power of 136.40W. Resulting in an average of 246,91 FPS (counting inference time only) and therefore 0.55 Joule per frame.

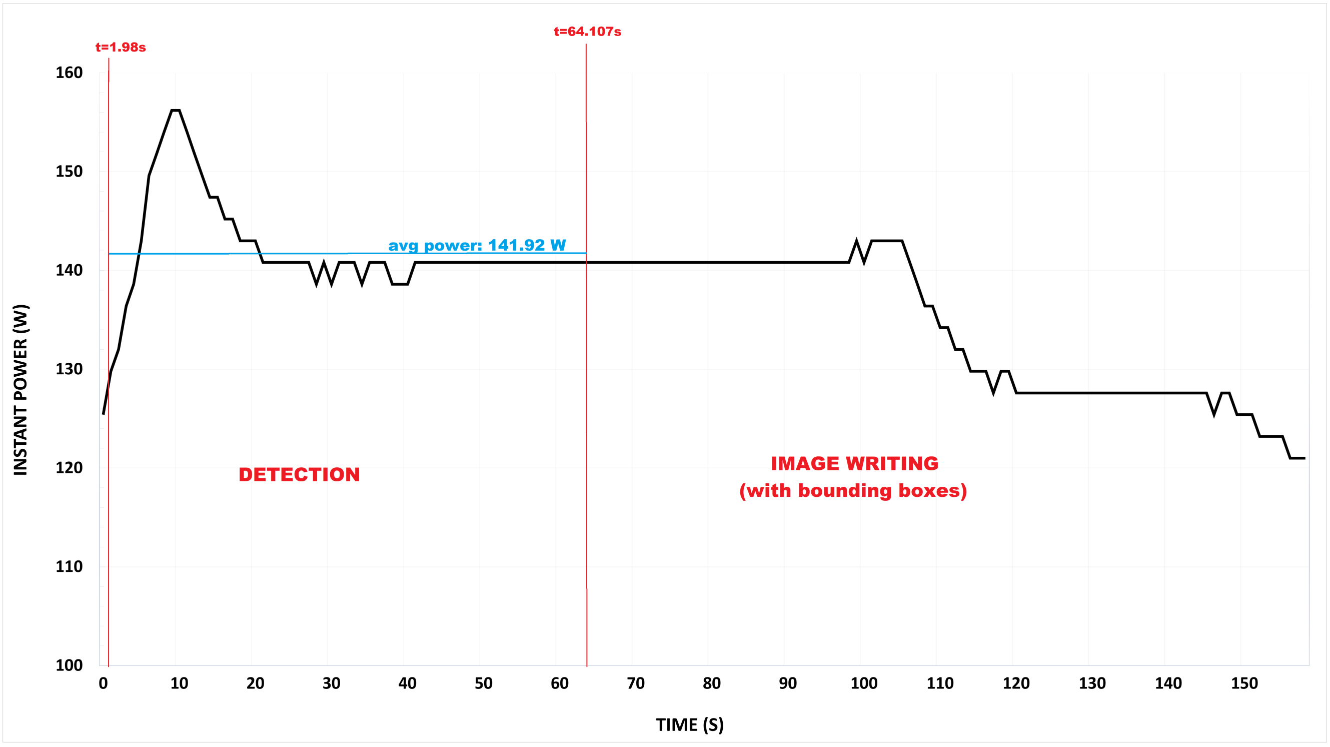


Figure 5.6: Power plot of i5-12400 CPU running a detection algorithm for YOLOv3-Tiny.

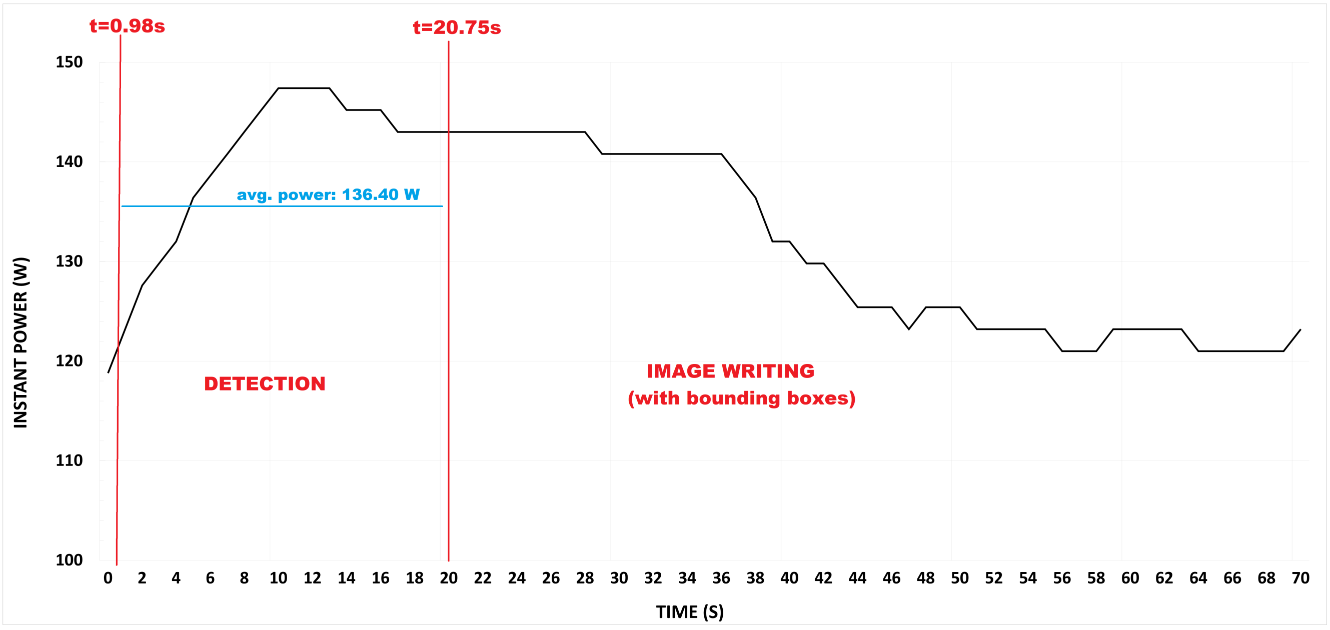


Figure 5.7: Power plot of i5-12400 CPU accelerated with RTX-3060 GPU running a detection algorithm for YOLOv3-Tiny.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

6.1.1 Choice of architecture

There are many techniques emerging that are increasingly sophisticated to increase the speed and accuracy of CNNs. People that are interested in the subject tend to incorporate new discoveries into their work. This is not necessarily a good thing, since we must always remember to select a CNN architecture that is appropriate for the dataset the network will train on. If the objective is to detect objects from the COCO dataset, a basic three-layer CNN will not yield decent results, but it will already be a suitable design for smaller datasets like MNIST or CIFAR10. The dataset's complexity should match the complexity of the architecture.

The YOLOv3 Tiny architecture can only reach 33% mAP on the COCO dataset, which has a wide range of inputs and 80 classes, but the YOLOv4 architecture can achieve 65.7%, nearly double that of the YOLOv3-Tiny. In other words, with the COCO dataset, more sophisticated architectures are required to get acceptable metrics. This is not the case for the Brakish dataset, which has a limited variety of inputs and just six classes. It is usual for simpler architectures to get high mAP levels when properly configured and used in the right scenario.

This was one of the work's conclusions: there is a temptation to constantly utilize the most recent designs with the most advanced algorithms, but we must remember that for simpler datasets, a simpler architecture can generate better results than more complicated architectures. However YOLOv4 and further versions still perform well in almost all situations, the use of older versions should be more encouraged, since we achieved nearly the same mAP with YOLOv3-Tiny as other works that focused on the advantage of using more complex architectures.

6.1.2 Underwater Image Filtering, where does it stand?

There are several reasons to exclude the filtering methodology used in this work as a pre-processing algorithms:

1. Since the filtering processing power isn't accelerated, it would take a long time to run creating a lot of latency and bringing consequences in energy consumption.

2. The filtering algorithms studied in this dissertation didn't influence positively the model as pre-processing algorithms.

3. By analysing **Table 5.4** one can conclude that the studied filtering methods don't create an advantage when detecting species with the augmented model either.

Because of these reasons, we can look at the studied filtering methods as post-processing algorithms for better human perception of the obtained results, and of course, as dataset augmentations.

If the purpose is for better human perception, this post processing could also occur outside of the edge device, after gathering the data for a cloud device to analyse.

6.1.3 Energy efficiency, did the result match the expectations?

It was said in **section 1.4** that this dissertation would be focused on prioritizing low energy consumption. However we used Vitis-AI which provides high frame rates and focuses on performance, since it only supports FPGAs designed for high performance computing, and offers limited amount of architecture configurations for the DPU. The initial objective would be relying on FINN since it offers endless hardware configurations, for example, by simply choosing different combinations of folding parameters (PE and SIMD). However, since FINN was left behind due to technical issues and difficulties, Vitis-AI was used as an amend.

Leaving this matter aside, the results obtained through Vitis-AI were acceptable since it provides better energy efficiency than using a GPU, as it is shown in **table 6.1**. Using the FPGA results in a performance drop of 71.629% in FPS, energy consumption drop of 56.364% in Joules per frame, and a mAP drop of 9.8%. These values are considered a good outcome since the frame rate and mAP are still in a good value range of 70.05 FPS and 82.6% respectively. Consuming 56.364% less energy means having 56.364% more autonomy if the device is purely focused on detection tasks, which makes a respectable difference and many underwater robotic devices for marine exploration could benefit from this study.

	Inference Performance and Efficiency			
Hardware Platform	FPGA		Computer	
	CPU	CPU+DPU	CPU	CPU+GPU
Frames per Second	1.88	70.05	16.79	246.91
Joules per Frame	3.83	0.31	8.45	0.55

Table 6.1: Summary of measurements for performance and energy efficiency for the different HW platforms.

A matter that should not be overlooked is the energy consumption in idle time. For idle times, while using PYNQ, the instant power supply was around 12.3W, and in Vitis-AI around 17.3W. PYNQ can load and unload bitfiles which can be accelerators developed by FINN or other frameworks, as long as they have the correct connections to the Zynq processor being used. This means that using PYNQ overlays would save an even greater amount of power if we take idle time power consumption into account.

6.2 Future Work

Before going on to supplementary and extra tasks, there are still improvements that may be made to this project, such as:

- Explore and compare the results from implementing the custom made architectures for underwater environments at [5] and [7], since they seem to be a better choice than YOLOv3-Tiny, using less parameters and less FLOPs.

- Use an accelerator created with the FINN framework, possibly after an updated and better version is released.

- There are still other DPU architectural configurations to be used in Vitis-AI and some tools provided by Xilinx to support projects made with Vitis-AI, such as the Vitis-AI Optimizer and Profiler. With the optimizer one can perform model pruning and optimize the compiled models, which reduces the number of operations during model inference. The profiler serves as an monitoring tool for model performance, to check where it lacks optimization. In this project we only used the compiler and quantizer. The profiler was also used, but only in order to visualize the results obtained from tracing model inference.

  It is also possible to add custom IP in the Vivado project, this raises the possibility of having dedicated IP to read images from a camera in real time.

The following interesting additions would be useful, perhaps after the project's core is more stable:

- Make use of the encoder/decoder approach to incorporate filtering into the edge device, which would use CNNs and make it simpler to parallelize the filtering procedure, since there are tools like FINN and Vitis-AI that can do this for us.

- Using two filter modes that would be selected based on the turbidity of the water (this is possible with turbidity sensors). When the water is clear, filtering algorithms such as RoWS or RGHS may produce superior results. In circumstances when the water is more turbid or visibility is reduced for other reasons, the edge device can employ the filtering developed in this dissertation (Color Restored GC_CLAHE). This opens up the prospect of investigating the undersea habitat at night, or bigger depths where the light levels are low.

Bibliography

[1] EvoLogics BOSS - Manta Ray webpage. `https://evologics.de/projects/boss`. Accessed: 2022-07-29.

[2] PYNQ webpage. `htt-p://www.pynq.io/`. Accessed: 2022-08-13.

[3] Minghao Zhao, Chengquan Hu, Fenglin Wei, Kai Wang, Chong Wang, and Yu Jiang. Real-time underwater image recognition with fpga embedded system for convolutional neural network. *Sensors*, 19(2), 2019.

[4] Liangwei Cai, Ceng Wang, and Yuan Xu. A real-time fpga accelerator based on winograd algorithm for underwater object detection. *Electronics*, 10:2889, 11 2021.

[5] Lin Wang, Xiufen Ye, Huiming Xing, Zhengyang Wang, and Peng Li. Yolo nano underwater: A fast and compact object detector for embedded device. In *Global Oceans 2020: Singapore – U.S. Gulf Coast*, pages 1–4, 2020.

[6] Minghua Zhang, Shubo Xu, Wei Song, Qi He, and Quanmiao Wei. Lightweight underwater object detection based on yolo v4 and multi-scale attentional feature fusion. *Remote Sensing*, 13:4706, 11 2021.

[7] Shijia Zhao, Jiachun Zheng, Shidan Sun, and Lei Zhang. An improved yolo algorithm for fast and accurate underwater object detection. *Symmetry*, 14(8), 2022.

[8] Michal Machura, Michal Danilowicz, and Tomasz Kryjak. Embedded object detection with custom littlenet, finn and vitis ai dcnn accelerators. *Journal of Low Power Electronics and Applications*, 12(2), 2022.

[9] Shenbagaraman Ramakrishnan. Implementation of a Deep Learning Inference Accelerator on the FPGA. Master's thesis, LUND UNIVERSITY, Sweden, 2020.

[10] Michaela Blott, Thomas Preusser, Nicholas Fraser, Giulio Gambardella, Kenneth O'Brien, and Yaman Umuroglu. Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks, 2018.

[11] Thomas B. Preuser, Giulio Gambardella, Nicholas Fraser, and Michaela Blott. Inference of quantized neural networks on heterogeneous all-programmable devices. In *2018 Design, Automation &amp Test in Europe Conference &amp Exhibition (DATE)*. IEEE, mar 2018.

[12] Colin Yu Lin, Zhenghong Jiang, Cheng Fu, Hayden Kwok-Hay So, and Haigang Yang. Fpga high-level synthesis versus overlay: Comparisons on computation kernels. *SIGARCH Comput. Archit. News*, 44(4):92–97, jan 2017.

[13] Malte Pedersen, Joakim Bruslund Haurum, Rikke Gade, Thomas B. Moeslund, and Niels Madsen. Detection of marine animals in a new underwater dataset with varying visibility. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2019.

[14] Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit Webpage. `https://www.xilinx.com/products/boards-and-kits/zcu104.html`. Accessed: 2022-07-29.

[15] Laith Alzubaidi, Jinglan Zhang, Amjad J. Humaidi, Ayad Al-Dujaili, Ye Duan, Omran Al-Shamma, J. Santamaría, Mohammed A. Fadhel, Muthana Al-Amidie, and Laith Farhan. Review of deep learning: concepts, cnn architectures, challenges, applications, future directions. *Journal of Big Data*, 8(1):53, Mar 2021.

[16] D H Hubel and T N Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *J Physiol*, 160(1):106–154, January 1962.

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

[18] Pradeep N, Sandeep Kautish, and Sheng-Lung Peng. *Demystifying Big Data, Machine Learning, and Deep Learning for Healthcare Analytics.* Academic Press, 2021.

[19] Yann LeCun, Koray Kavukcuoglu, and Clement Farabet. Convolutional networks and applications in vision. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 253–256, 2010.

[20] Ruiqi Chen, Tianyu Wu, Yuchen Zheng, and Ming Ling. Mlof: Machine learning accelerators for the low-cost fpga platforms. *Applied Sciences*, 12(1), 2022.

[21] Abien Fred Agarap. Deep learning using rectified linear units (relu). *CoRR*, abs/1803.08375, 2018.

[22] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *CoRR*, abs/1505.00853, 2015.

[23] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. Yolov7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors, 2022.

[24] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788, 2016.

[25] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation, 2013.

[26] Pedro F. Felzenszwalb, Ross B. Girshick, David McAllester, and Deva Ramanan. Object detection with discriminatively trained part-based models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(9):1627–1645, 2010.

[27] Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6517–6525, 2017.

[28] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement, 2018.

[29] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection, 2016.

[30] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection, 2020.

[31] Bo Gong, Daji Ergu, Ying Cai, and Bo Ma. A method for wheat head detection based on yolov4, 09 2020.

[32] Shu Liu, Lu Qi, Haifang Qin, Jianping Shi, and Jiaya Jia. Path aggregation network for instance segmentation, 2018.

[33] Sanghyun Woo, Jongchan Park, Joon-Young Lee, and In So Kweon. Cbam: Convolutional block attention module, 2018.

[34] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. In *Computer Vision – ECCV 2014*, pages 346–361. Springer International Publishing, 2014.

[35] Ross Girshick. Fast r-cnn. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1440–1448, 2015.

[36] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[37] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-fcn: Object detection via region-based fully convolutional networks. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

[38] Jiangmiao Pang, Kai Chen, Jianping Shi, Huajun Feng, Wanli Ouyang, and Dahua Lin. Libra r-cnn: Towards balanced learning for object detection, 2019.

[39] Hei Law and Jia Deng. Cornernet: Detecting objects as paired keypoints, 2018.

[40] Kaiwen Duan, Song Bai, Lingxi Xie, Honggang Qi, Qingming Huang, and Qi Tian. Centernet: Keypoint triplets for object detection, 2019.

[41] Abdullah Rashwan, Agastya Kalra, and Pascal Poupart. Matrix nets: A new deep architecture for object detection, 2019.

[42] Ze Yang, Shaohui Liu, Han Hu, Liwei Wang, and Stephen Lin. Reppoints: Point set representation for object detection, 2019.

[43] Joseph Chet Redmon's webpage. `https://pjreddie.com/`. Accessed: 2022-08-08.

[44] Pranav Adarsh, Pratibha Rathi, and Manoj Kumar. Yolo v3 tiny object detection and recognition using one stage improved model. pages 687–694, 03 2020.

[45] Upesh Nepal and Hossein Eslamiat. Comparing yolov3, yolov4 and yolov5 for autonomous landing spot detection in faulty uavs. *Sensors*, 22(2), 2022.

[46] Karel J. Zuiderveld. Contrast limited adaptive histogram equalization. In *Graphics Gems*, 1994.

[47] Stephen M. Pizer, E. Philip Amburn, John D. Austin, Robert Cromartie, Ari Geselowitz, Trey Greer, Bart ter Haar Romeny, John B. Zimmerman, and Karel Zuiderveld. Adaptive histogram equalization and its variations. *Computer Vision, Graphics, and Image Processing*, 39(3):355–368, 1987.

[48] S.M. Pizer, R.E. Johnston, J.P. Ericksen, B.C. Yankaskas, and K.E. Muller. Contrast-limited adaptive histogram equalization: speed and effectiveness. In *[1990] Proceedings of the First Conference on Visualization in Biomedical Computing*, pages 337–345, 1990.

[49] Shanto Rahman, Md Mostafijur Rahman, M. Abdullah-Al-Wadud, Golam Dastegir Al-Quaderi, and Mohammad Shoyaib. An adaptive gamma correction for image enhancement. *EURASIP Journal on Image and Video Processing*, 2016(1):35, Oct 2016.

[50] Wending Xiang, Ping Yang, Shuai Wang, Bing Xu, and Hui Liu. Underwater image enhancement based on red channel weighted compensation and gamma correction model. *Opto-Electronic Advances*, 1:18002401–18002409, 10 2018.

[51] Liu Chao and Meng Wang. Removal of water scattering. In *2010 2nd International Conference on Computer Engineering and Technology*, volume 2, pages V2–35–V2–39, 2010.

[52] Dongmei Huang, Yan Wang, Wei Song, Jean Sequeira, and Sébastien Mavromatis. Shallow-water image enhancement using relative global histogram stretching based on adaptive parameter acquisition. In Klaus Schoeffmann, Thanarat H. Chalidabhongse, Chong Wah Ngo, Supavadee Aramvith, Noel E. O'Connor, Yo-Sung Ho, Moncef Gabbouj, and Ahmed Elgammal, editors, *MultiMedia Modeling*, pages 453–465, Cham, 2018. Springer International Publishing.

[53] Stephen M. Pizer. Intensity mappings to linearize display devices. *Computer Graphics and Image Processing*, 17(3):262–268, 1981.

[54] David J. Ketcham. Real-Time Image Enhancement Techniques. In John C. Urbach, editor, *Image Processing*, volume 0074, pages 120 – 125. International Society for Optics and Photonics, SPIE, 1976.

[55] Robert Hummel. Image enhancement by histogram transformation. *Computer Graphics and Image Processing*, 6(2):184–195, 1977.

[56] Kaiming He, Jian Sun, and Xiaoou Tang. Single image haze removal using dark channel prior. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1956–1963, 2009.

[57] FINN webpage. `https://finn.readthedocs.io/en/latest/`. Accessed: 2022-08-13.

[58] Vladimir Rybalkin, Alessandro Pappalardo, Muhammad Ghaffar, Giulio Gambardella, Norbert Wehn, and Michaela Blott. Finn-l: Library extensions and design trade-off analysis for variable precision lstm networks on fpgas, 07 2018.

[59] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. FINN. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, feb 2017.

[60] Alessandro Pappalardo. Xilinx/brevitas, 2021.

[61] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference, 2021.

[62] Yaman Umuroglu and Magnus Jahre. Streamlined deployment for quantized neural networks, 2017.

[63] Vitis AI webpage. `https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html`. Accessed: 2022-08-22.

[64] Product Guide DPU ZCU104 and ZCU102. `https://docs.xilinx.com/r/en-US/pg338-dpu/Port-Descriptions`. Accessed: 2022-08-22.

[65] FFmpeg website. `https://ffmpeg.org`. Accessed: 2022-08-22.

[66] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *International Journal of Robotics Research (IJRR)*, 2013.

[67] Yan Wang, Wei Song, Giancarlo Fortino, Lizhe Qi, Wenqiang Zhang, and Antonio Liotta. An experimental-based review of image enhancement and image restoration methods for underwater imaging, 2019.

[68] Vitis Ai tutorials. `https://github.com/Xilinx/Vitis-AI-Tutorials`. Accessed: 2022-09-1.

[69] Multimeter webpage. `https://instruments.uni-trend.com/list_80/1238.html`. Accessed: 2022-07-29.

[70] José Pedro Araújo Azevedo. YOLO Neural Networks on Reconfigurable Logic for Vehicle Traffic Crontoll. Master's thesis, University of Coimbra, Portugal, 2022.

[71] scipy documentation. `https://docs.scipy.org/doc/scipy-0.18.1/reference/generated/scipy.integrate.simps.html`. Accessed: 2022-08-13.

Appendix A

Vitis AI IP connections

This chapter contains schematics from the DPU product guide [64], that supplement explanations issued in chapter 4.

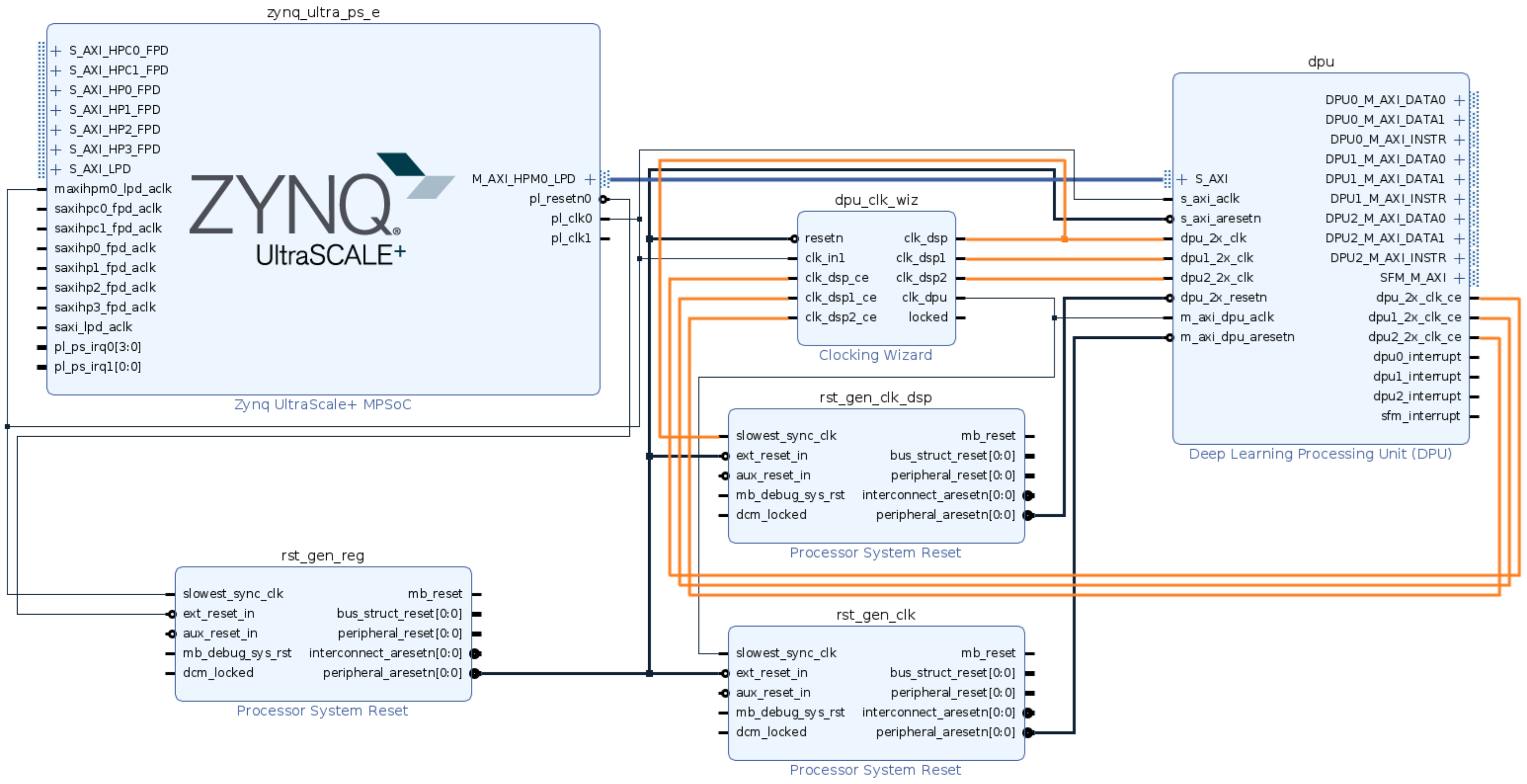


Figure A.1: Schematic from [64], demonstrating clocking and reset related connections between the DPU and CPU.

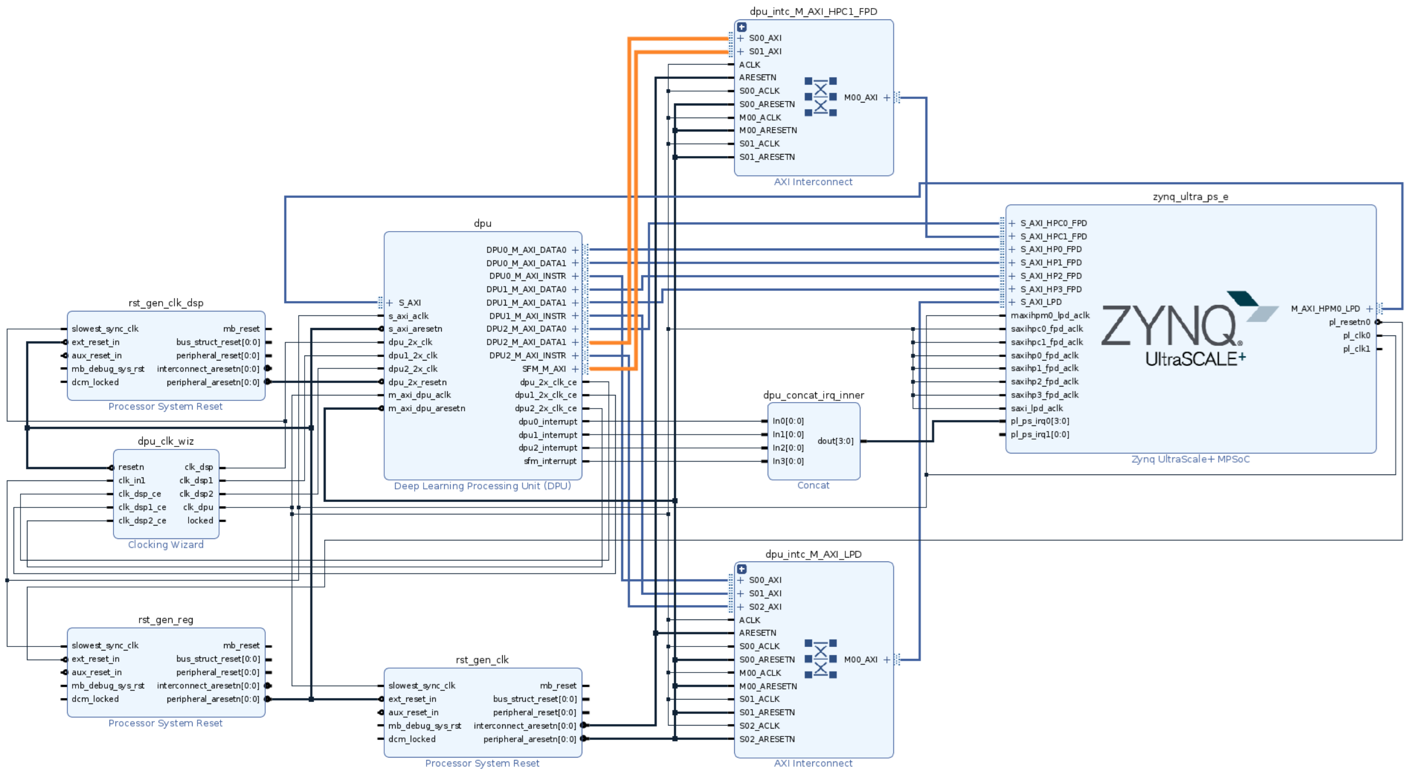


Figure A.2: Schematic from [64], demonstrating clocking, reset, interrupts, and dataflow related connections between the DPU and CPU.

Appendix B

Python Requirements

This chapter contains system requirements for the desktop computer in order to reproduce similar results from those obtained in this dissertation.

B.1 Filtering

Required packages (python v3.7.0 and pip v22.1):

```
matplotlib==3.5.2
natsort==8.1.0
numpy==1.21.6
opencv_python==4.5.5.64
scikit_image==0.19.2
scipy==1.7.3
skimage==0.0
xlwt==1.3.0
```

B.2 Training, testing and detection (desktop)

- **Python**: Python 3.7.0
- **GPU** (not a requirement, just to justify the use of torch-cuda version): Geforce RTX 3060
- **Packages:**

```
matplotlib==3.5.2
numpy==1.21.6
opencv_python==4.5.5.64
Pillow==9.2.0
pip==22.1
scikit_image==0.19.2
skimage==0.0
terminaltables==3.1.10
torch==1.8.2+cu111
torchvision==0.9.2+cu111
```

```
tqdm==4.64.0
imgaug==0.4.0
```

Appendix C

YOLOv3-Tiny Architecture - configuration file

This chapter contains the configuration file used to train the YOLOv3-Tiny model, it has detailed information about many of the used hyper parameters. If training for Vitis AI implementation, comment the maxpooling layer of stride 1.

```
[net]
# Testing
batch=1
subdivisions=1
# Training
# batch=64
# subdivisions=2
width=416
height=416
channels=3
momentum=0.9
decay=0.0005
angle=0
saturation = 1.5
exposure = 1.5
hue=.1

learning_rate=0.001
burn_in=1000
max_batches = 500200
policy=steps
steps=400000,450000
scales=.1,.1

[convolutional]
batch_normalize=1
filters=16
size=3
stride=1
pad=1
activation=leaky
```

```
[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=32
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=64
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=128
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=256
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2
```

```
[convolutional]
batch_normalize=1
filters=512
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=1

[convolutional]
batch_normalize=1
filters=1024
size=3
stride=1
pad=1
activation=leaky

###########

[convolutional]
batch_normalize=1
filters=256
size=1
stride=1
pad=1
activation=leaky

[convolutional]
batch_normalize=1
filters=512
size=3
stride=1
pad=1
activation=leaky

[convolutional]
size=1
stride=1
pad=1
filters=255
activation=linear


[yolo]
mask = 3,4,5
anchors = 10,14,  23,27,  37,58,  81,82,  135,169,  344,319
classes=80
```

```
num=6
jitter=.3
ignore_thresh = .7
truth_thresh = 1
random=1

[route]
layers = -4

[convolutional]
batch_normalize=1
filters=128
size=1
stride=1
pad=1
activation=leaky

[upsample]
stride=2

[route]
layers = -1, 8

[convolutional]
batch_normalize=1
filters=256
size=3
stride=1
pad=1
activation=leaky

[convolutional]
size=1
stride=1
pad=1
filters=255
activation=linear

[yolo]
mask = 0,1,2
anchors = 10,14,  23,27,  37,58,  81,82,  135,169,  344,319
classes=80
num=6
jitter=.3
ignore_thresh = .7
truth_thresh = 1
random=1
```

Appendix D

PYNQ packages - ZCU 104

This chapter provides a quick setup for the FPGA *Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit* when using PYNQ framework. Copy the following list of packages to a requirements.txt folder and run *$pip install -r requirements.txt* in the FPGA's console.
Python version: 3.8.2
Pynq version: 2.7.0
Python packages:

```
2ping==4.3
alabaster==0.7.12
anyio==3.1.0
argon2-cffi==20.1.0
async-generator==1.10
atomicwrites==1.1.5
attrs==19.3.0
Babel==2.9.1
backcall==0.2.0
beautifulsoup4==4.8.2
bitstring==3.1.9
bleach==3.3.0
blinker==1.4
blosc==1.7.0
brevitas==0.7.1
Brotli==1.0.9
certifi==2019.11.28
cffi==1.14.5
chardet==3.0.4
charset-normalizer==2.0.12
click==8.0.1
cloudpickle==1.3.0
CppHeaderParser==2.7.4
cryptography==2.8
cupshelpers==1.0
cycler==0.10.0
Cython==0.29.24
dash==2.0.0
dash-bootstrap-components==0.13.1
dash-core-components==2.0.0
dash-html-components==2.0.0
```

```
dash-renderer==1.9.1
dash-table==5.0.0
dask==2.8.1+dfsg
dbus-python==1.2.16
decorator==4.4.2
defer==1.0.6
defusedxml==0.7.1
deltasigma==0.2.2
dependencies==2.0.1
distro==1.4.0
distro-info==0.23ubuntu1
dnspython==1.16.0
docker-pycreds==0.4.0
docutils==0.17.1
entrypoints==0.3
et-xmlfile==1.0.1
finn-base==0.0.3
finn-dataset-loading==0.0.5
finn-examples==0.0.4
Flask==2.0.1
Flask-Compress==1.10.1
fonttools==4.33.3
fsspec==0.6.1
future-annotations==1.0.0
gitdb==4.0.9
GitPython==3.1.27
gpg==1.13.1-unknown
gTTS==2.2.3
html5lib==1.0.1
httplib2==0.14.0
idna==2.8
imageio==2.4.1
imagesize==1.2.0
imgaug==0.4.0
importlib-metadata==1.5.0
imutils==0.5.4
install==1.3.5
ipykernel==5.5.5
ipython==7.24.0
ipython_genutils==0.2.0
ipywidgets==7.6.3
itsdangerous==2.0.1
jdcal==1.0
jedi==0.17.2
Jinja2==3.0.1
joblib==1.1.0
json5==0.9.5
jsonschema==3.2.0
jupyter==1.0.0
jupyter-client==6.1.12
jupyter-console==6.4.0
```

```
jupyter-contrib-core==0.3.3
jupyter-contrib-nbextensions==0.5.1
jupyter-core==4.7.1
jupyter-highlight-selected-word==0.2.0
jupyter-latex-envs==1.4.6
jupyter-nbextensions-configurator==0.4.1
jupyter-server==1.8.0
jupyterlab==3.0.16
jupyterlab-pygments==0.1.2
jupyterlab-server==2.5.2
jupyterlab-widgets==1.0.0
jupyterplot==0.0.3
keyring==18.0.1
kiwisolver==1.0.1
language-selector==0.1
launchpadlib==1.10.13
lazr.restfulclient==0.14.2
lazr.uri==1.0.3
locket==0.2.0
lrcurve==1.1.0
lxml==4.5.0
macaroonbakery==1.3.1
Markdown==3.1.1
MarkupSafe==2.0.1
matplotlib==3.5.2
matplotlib-inline==0.1.2
mistune==0.8.4
mnist==0.2.2
more-itertools==4.2.0
mpmath==1.1.0
nbclassic==0.3.1
nbclient==0.5.3
nbconvert==6.0.7
nbformat==5.1.3
nbsphinx==0.8.7
nbwavedrom==0.2.0
nest-asyncio==1.5.1
netifaces==0.11.0
networkx==2.4
notebook==6.4.0
numexpr==2.7.1
numpy==1.22.3
oauthlib==3.1.0
olefile==0.46
opencv-python==4.5.5.64
openpyxl==3.0.3
packaging==20.3
pandas==1.4.2
pandocfilters==1.4.3
parsec==3.9
parso==0.7.1
```

```
partd==1.0.0
pathtools==0.1.2
patsy==0.5.1
pbr==5.6.0
pexpect==4.8.0
pickleshare==0.7.5
Pillow==9.1.0
pip==22.0.4
pkg_resources==0.0.0
plotly==5.1.0
pluggy==0.13.0
ply==3.11
prometheus-client==0.10.1
promise==2.3
prompt-toolkit==3.0.18
protobuf==3.20.1
psutil==5.8.0
ptyprocess==0.7.0
py==1.8.1
PyAudio==0.2.11
pybind11==2.8.0
pycairo==1.20.1
pycparser==2.19
pycrypto==2.6.1
pycups==1.9.73
pycurl==7.43.0.2
pyeda==0.28.0
Pygments==2.9.0
PyGObject==3.36.0
pygraphviz==1.5
PyJWT==1.7.1
pymacaroons==0.13.0
PyNaCl==1.3.0
pynq==2.7.0
pynq-dpu==1.4.0
pynq-peripherals==0.1.0
pyparsing==2.4.6
pyRFC3339==1.1
pyrsistent==0.17.3
pytest==4.6.9
pytest-sourceorder==0.5.1
python-apt==2.0.0
python-dateutil==2.8.2
pytz==2022.1
PyWavelets==0.5.1
PyYAML==5.3.1
pyzmq==22.1.0
qtconsole==5.1.0
QtPy==1.9.0
requests==2.27.1
requests-unixsocket==0.2.0
```

```
retrying==1.3.3
rise==5.7.1
roman==3.3
scikit-image==0.16.2
scikit-learn==1.0.2
scipy==1.8.0
seaborn==0.11.2
SecretStorage==2.3.1
Send2Trash==1.5.0
sentry-sdk==1.9.4
setproctitle==1.2.2
setuptools==44.0.0
Shapely==1.8.1.post1
shortuuid==1.0.9
simple-term-menu==1.4.1
simplegeneric==0.8.1
simplejson==3.16.0
six==1.14.0
smmap==5.0.0
sniffio==1.2.0
snowballstemmer==2.1.0
soupsieve==1.9.5
SpeechRecognition==3.8.1
Sphinx==4.2.0
sphinx-rtd-theme==1.0.0
sphinxcontrib-applehelp==1.0.2
sphinxcontrib-devhelp==1.0.2
sphinxcontrib-htmlhelp==2.0.0
sphinxcontrib-jsmath==1.0.1
sphinxcontrib-qthelp==1.0.3
sphinxcontrib-serializinghtml==1.1.5
SQLAlchemy==1.3.12
ssh-import-id==5.10
sympy==1.5.1
systemd-python==234
tables==3.6.1
tenacity==8.0.0
terminado==0.10.0
terminaltables==3.1.10
testpath==0.5.0
testresources==2.0.1
thop==0.0.31.post2005241907
threadpoolctl==3.1.0
tokenize-rt==4.2.1
toolz==0.9.0
torch==1.11.0
torchvision==0.12.0
tornado==6.1
tqdm==4.62.3
traitlets==5.0.5
transitions==0.7.2
```

```
typing_extensions==4.1.1
ubuntu-advantage-tools==20.3
unattended-upgrades==0.1
urllib3==1.26.11
uvloop==0.14.0
voila==0.2.10
voila-gridstack==0.2.0
wadllib==1.3.3
wandb==0.13.1
wcwidth==0.1.8
webencodings==0.5.1
websocket-client==1.0.1
Werkzeug==2.0.1
wheel==0.37.1
widgetsnbextension==3.5.1
wurlitzer==3.0.2
xlrd==1.1.0
xlwt==1.3.0
zipp==1.0.0
```