TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

U-NET Neural Network Analysis and Implementation using Reconfigurable Logic

Author:
Charalampos SKOUFIS

Thesis Committee:
Prof. Apostolos DOLLAS
Assoc. Prof. Michail LAGOUDAKIS
Prof. Michail ZERVAKIS

A thesis submitted in fulfillment of the requirements for the diploma of Electrical and Computer Engineer

in the

School of Electrical and Computer Engineering
Microprocessor and Hardware Laboratory

February 22, 2021

Abstract

Diploma Thesis

U-NET Neural Network Analysis and Implementation using Reconfigurable Logic

by Charalampos SKOUFIS

In recent years, neural networks are increasingly the primary tool for image analysis, providing exceptional accuracy vs. human perception. In the field of biomedicine, in particular, the misdiagnosis of magnetic resonance imaging or computed tomography (MRI / CT) scans is a significant problem in preventing and treating various health problems which are impossible to detect by the human eye. In the field of terrain pattern recognition performed by power-limited mini-satellites, a more efficient approach for both architecture and hardware equipment is required. A recent U-shaped architecture offers impressive results and methods for detecting patterns and anomalies using semantic image segmentation. This thesis work is based on this U-NET architecture and aims to analyze, model, and build the network on multiple programming levels of abstraction, including hardware. At present, there exist more mature architectures such as Convolutional Neural Networks (CNN) that have substantial support toolsets. On the other hand, U-NET architecture does not have a great level of support tools; this work will try to address this issue. The main structure and learning process (training) of this neural network will also be presented in detail, along with all the additional tools to assist this process. The code pack starts with a user-friendly Python language, where user-customizable functions and training techniques will be introduced. The Python language level is intended mostly to aid the learning process. One step further,researchers can proceed by utilizing the C language, where the prediction step has been constructed to be further analyzed and eventually reach a specific application platform. Finally, three building blocks of this network have been implemented on Field Programmable Gate Array (FPGA) and Graphics Processor Unit (GPU) platforms (on par with the entire NN), offering the acceleration of specific processes with substantial energy savings for the computation. Last, but not least, the ecosystem developed in this thesis was not available until now - with its use more researchers can efficiently employ U-NETs.

Acknowledgements

First of all, I would like to express my gratitude and appreciation to Professor Apostolos Dollas, whose support, encouragement and guidance have been invaluable throughout this study. I would also like to thank him for giving me the opportunity to become a part of the research team and inspiring me to work even harder to achieve the highest possible result. His crucial role was essential, mostly when things were at an impasse, leading the way on how to assemble, ameliorate and promote my work successfully.

Moreover, I would like to thank Assoc. Prof. Michail Lagoudakis, and Prof. Michail Zervakis, my thesis committee, for evaluating my work.

Furthermore, I would like to thank the entire TUC and ICS/FORTH CARV team for their excellent Collaboration, advising and accessibility to Xilinx Tools and Hardware equipment. The meetings and conversations were vital in inspiring me to think outside the box, from numerous viewpoints, to form an objective and comprehensive critique.

I would also like to thank ICS/FORTH researchers Dr. Gregory Tsagkatakis for the time he devoted analyzing and evaluating the model and the mathematical side of my work, Dr. Christos Kozanitis for the framework and hardware expertise and Dr. Aggelos Ioannou for providing valuable information that helped me to overcome technical oriented problems regarding Xilinx Tools.

I would like to thank Tzanis Fotakis, now a TUC graduate, for his continued support, guidance through the software features and possibilities that was not possible for an individual to uncover during a time-limited window.

Finally, I would like to express my sincere gratitude to my family and friends who have constantly been there for me through the years of my studies.

Charalampos Skoufis,
Chania, 2021

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

AI	**A**rtificial **I**ntelligence
ALU	**A**rithmetic **L**ogic **U**nit
ANN	**A**rtificial **N**eural **N**etwork
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
AXI	**A**dvanced e**X**tensible **I**nterface
BRAM	**B**lock **R**andom **A**ccess **M**emory
BSP	**B**oard **S**upport **P**ackage
CNN	**C**onvolutional **N**eural **N**etwork
CPU	**C**entral **P**rocessor **U**nit
CS	**C**omputer **S**cience
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
cuDNN	CUDA **D**eep **N**eural **N**etwork library
DDR4	**D**ouble **D**ata **R**ate type texbf4 memory
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
DNN	**D**eep **N**eural **N**etwork
DPU	**D**eep Learning **P**rocessing **U**nit
DSP	**D**igital **S**ignal **P**rocessor
FC	**F**ully **C**onnected
FF	**F**lip **F**lop
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
FORTH	**F**undation **o**f **R**esearch and **T**echnology **H**ellas
FSBL	**F**irst **S**tage **B**oot **L**oader
GDDR6	**G**raphics **D**ouble **D**ata **R**ate type **6** memory
GPU	**G**raphic **P**rocessor **U**nit
HBM	**H**igh **B**andwidth **M**emory
HDL	**H**ardware **D**escription **L**anguage
HLS	**H**igh **L**evel **S**ynthesis
HPC	**H**ight **P**erformance **C**omputing
ILA	**I**ntegrated **L**ogic **A**nalyzer
ILSVRC	**I**mageNet **L**arge **S**cale **V**isual **R**ecognition **C**hallenge
LUT	**L**ook **U**p **T**able

ML	**M**achine **L**earning
MLP	**M**ulti-**L**ayer **P**erceptron
MMIO	**M**emory-**M**apped **I/O**
MPSoC	**M**ulti **P**rocessor **S**ystem **o**n **C**hip
MXU	**M**atrix Mutliplier **U**nit
PE	**P**rocessing **E**lement
PL	**P**rogrammable **L**ogic
PS	**P**rocessing **S**ystem
QFDB	**Q**uad **F**PGA **D**aughter **B**oard
RAM	**R**andom **A**ccess **M**emory
ReLU	**Re**ctified **L**inear **U**nit
SDK	**S**oftware **D**evelopment **K**it
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
SM	**S**treaming **M**ultiprocessor
SLC	**S**econd **L**evel **C**odebook
SSE	**S**treaming **S**IMD **E**xtensions
SSD	**S**olid **S**tate **D**rive
TDP	**T**hermal **D**esign **P**ower
TPU	**T**ensor **P**rocessor **U**nit
URAM	**U**ltra **R**andom **A**ccess **M**emory
USD	**U**nited **S**tates **D**ollar

Dedicated to my family and friends...

Chapter 1

Introduction

In 1950 machine learning [1] started its first steps when the baseline was discovered, and today's backbone of Artificial Intelligence (AI) [2] was built. Many years later, around 1980, the term 'back-propagation' [3] emerged, giving the remarkable ability of 'learning' to an ordinary machine. In the last twenty years, machine learning is developing quite conspicuously. AI is continuously learning and evolving, predicting pandemics or even synthesizing advanced medicines from the ground up. The recent years' technological leaps are impressive, marking down a massive leap towards the last century. Big data [4] is a rising problem that needs to be restrained using state-of-art techniques and, of course, with AI running efficiently on perfectly designed hardware solutions. Another critical technology field is the hardware equipment, including CPUs, electric vehicles, and the chasing of unlimited 'clean' energy using fusion reactors. Central Processing Units(CPU) [5] are becoming smaller and smaller, fighting against physics's classic laws trying to enter the quantum era. The Atom-thick transistors are a fact, using super-materials like graphene or germanium, fighting to keep Moore's law alive. This kind of architecture(×86) development is already saturated, and it is abandoned by some worlds leading companies like IBM, who are looking at the quantum field. Today's CPUs have minuscule new features keeping the same size (in nanometers), thus adding more cores to 'buy' some time. Because of these reasons mentioned above, Windows and macOS, which are widely known, are shifting to the arm architectures, a mobile-based processing unit providing outstanding performance while keeping the power consumption at a minimum. Some other hot technologies include self-driving cars powered by clean energy with zeros C02 emissions and foldable phones that can be folded and put in our pockets like a notebook with an edge-to-edge screen. Machine learning accelerates all these modern advancements even more, providing valuable information on the education system, helping business planning, and generally making our everyday lives more comfortable with the built in AI tools our gadgets boast. Image recognition, image to text, and speech to text come with some demanding needs in performance, mostly when built on a mobile platform. The algorithms used to run on mobile devices simulating a small neural network need to be parallelized, optimized, and implemented on a specific embedded system that aims to solve this kind of problem as a distinct unit. Our generation must integrate such embedded systems capable of handling massive data throughputs and algorithm complexities while retaining high efficiency levels.

1.1 Motivation

AlexNet [6], VGGNet [7], or GoogleNet [8] are the most famous deep neural network image classifiers which can read an image (or multiple images) and produce an output that includes a set of probabilities regarding the subjects of the entire input image. Semantic segmentation [9] is a whole different type of image analysis when the main goal is to break down the input image into multiple classes and then reconstruct the same image by putting back all the extracted features included in the subject of interest. Color overlays/masks are also added on top of the final result, so each subject's class can be visually differentiated from the rest. Before 2015, CNN [10] based neural networks were used in order to handle these mammoth semantic segmentation tasks. The conversion from a simple vector to an image wasn't the standout characteristic of CNNs. In 2015, the paper "U-Net : Convolutional Networks for Biomedical Image Segmentation" [11] proposed a state-of-art architecture of a U shaped neural network boasting some revolutionary improvements on training, mapping, and predicting large biomedical images outperforming the classic fully connected technique of the standard CNN based neural networks. Such an analysis can take up to 1 second for the top-rated GPUs of the market in modern days.

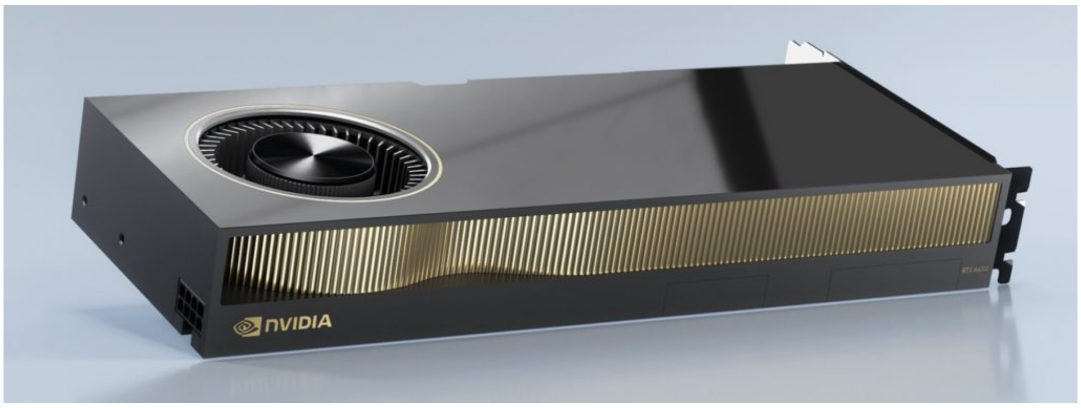


FIGURE 1.1: NVIDIA Quadro RTX A6000 - www.nvidia.com

For example, Nvidia's latest release specialized for workstations is the Quadro RTX A6000 [12] (Figure 1.1) featuring 48GB GDDR6 (ECC) of VRAM running at 768.0 GB/s, 10752 CUDA core, 336 texture mapping units, 84 ray tracing acceleration cores, 40 TFLOPS of single floating point (FP32) computations, and 300W TDP priced at $5,500.

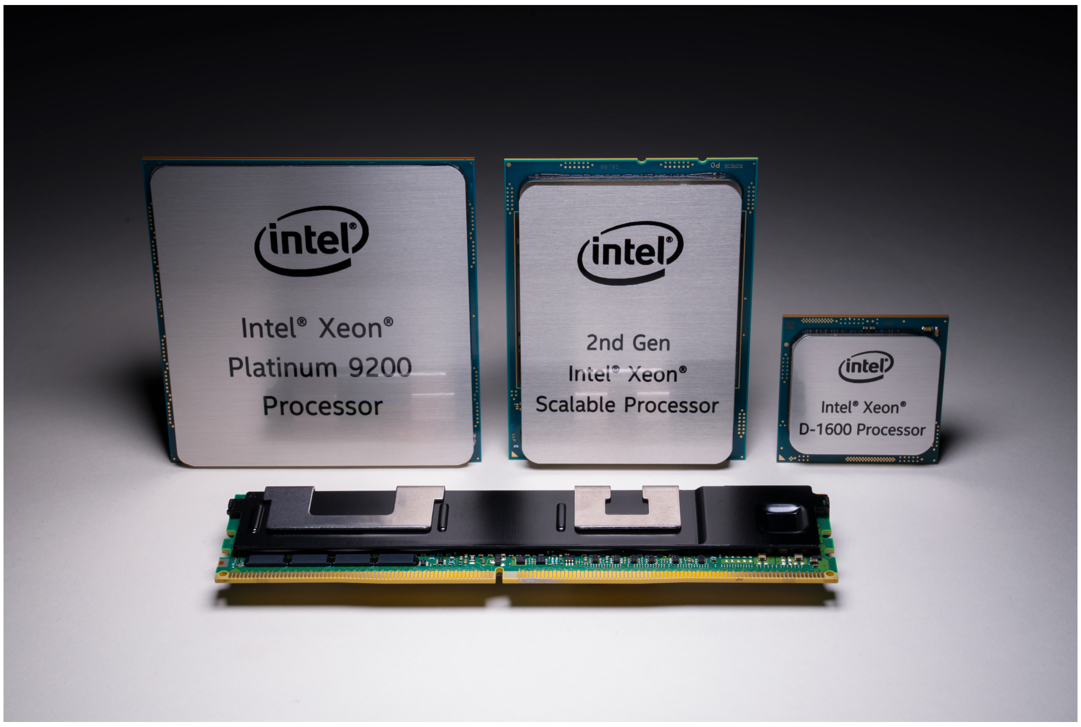


FIGURE 1.2: Intel Xeon Platinum 9200 Series Size comparison - ark.intel.com

Of course, to support this kind of raw acceleration power of the graphic / processing unit, the central processing unit must also be an advanced piece of technology. The latest Intel Xeon Platinum 9282 (Figure 1.2) takes advantage of the server / workstation space with its 56 cores, 112 threads with clock boost up to whopping 3.8 GHz, 400Watts of thermal design power, supporting AVX-512 [13], 77MB smart cache, and 14nm lithography. It retails for up to $50,000.

1.2 Scientific Contributions

What Problems will be addressed

- Lower level UNET and specific UNET functions implementation. A complete package of functions is provided as open-source for Python, C, and embedded systems.
- Organized framework equipped with multiple tools that assist its nominal functionality.
- Better power consumption/efficiency than the higher-level environments such as Matlab(CPU & GPU), and Keras (CPU).
- Basic blocks and structures which are not available as open-source knowledge at the moment.
- Publication of all the assets needed to build, model and analyze a UNET similarly with other famous architectures that already provide the 'know-how' and the functions at a lower level of abstraction that give the ability for further research and expansion.
- The proposed architecture(Version 2) can be used for any satellite running UNET for semantic segmentation, minimizing power consumption while offering similar performance with a optimized CPU/GPU.

Thesis Contribution

The scientific contribution of this work aims to provide a lower-level framework for UNET architecture, which is now available as open-source information. Tensorflow [14], [15], Parallel Computation Toolbox [16] by Matlab, MXNet [17], and other scientific teams, already have implemented all the required functions and

tools a U-shaped neural network needs to operate. In normal conditions, a newcomer/student can access a massive amount of information regarding simple neural network architectures, source code infrastructure, and training methods supplemented with all the necessary tools from High-level abstraction down to more hardware-specific programming/designing languages. The same applies for CNNs where nearly everything is available, from answered architecture based questions to already written functions for every kind of CNN's block. U-NET open-source material is limited to the high-level interfaces like Keras [18], Pytorch [19], while the source code of this architecture from the raw-python level and below is non-existent, since it is a fresh architecture that first appeared in 2015. Generally, U-NET is heavily analyzed in the mathematics field, but this kind of information is not enough for someone who wants to model and redefine architecture. Firstly, this work will make a thorough walkthrough of this architecture, providing many easy-image pre-processing, analyzing, and fine-tuning tools. Additionally, the network's training part is also built, featuring a user-friendly python environment so anyone can edit and tune the parameters adjusting it into a more specific-task network as the user desiderates. The analysis continues into C programming language where every algorithm is fully unwrapped, exposing its internals so the users can scrutinize its iterations, measure complexity, and optimize. On top of that modeling, robustness analysis, and FPGA implementation will follow. During this part, three core functions of the U-shaped architecture are being optimized and accelerated into hardware, achieving the most efficient way of U-net prediction/evaluation. FPGA's BRAM [20] is a severe restriction on that platform since it offers just a few MB of temporary(cached) fast storage. The final product consists of three IPs, the convolution, transposed convolution, and max pool IP. They are built based on the most famous U-NET architecture, as mentioned above, with the support up to 256×256 (2^n form) input image resolution. The overall BRAM usage reaches 74% of utilization, which means that these IPs can also be used in different platforms with significantly less BRAM resources. Each accelerator's core techniques have an extended presentation featuring custom line buffers, loop unrolling, and tree-structure computations for parallel computing. This thesis is a fundamental block that will provide the basic structure / ecosystem that can be used for deeper UNET development and research.
Some useful examples include the installation of the final product(embedded system) into technologies that operate with limited power source(such as mini-satellite) replacing a common CPU/GPU and reaching their performance / power efficiency with just a fraction of their total on-chip power.

1.3 Thesis Outline

- **Chapter 2 - Theoretical Background:** Machine Learning, Convolution Neural Networks, Residual/U-NET Neural Networks, basic functions, and other essential theoretical backgrounds are represented in this chapter.
- **Chapter 3 - Related Work:** Some other approaches of Semantic segmentation are being described.
- **Chapter 4 - In-Depth Theoretical Modeling & Robustness Analysis:** Classical U-NET architecture break down, prototype training part analysis with mathematics and multiple algorithm presentations.
- **Chapter 5 - FPGA Implementation:** Using ZCU102 [21], U-NET can be loaded into a large FPGA family. Illustration of the three IPs architecture.
- **Chapter 6 - Results:** Custom training function results are being posted, with visual examples plus some indicative performance results are being compared between multiple platforms and environments.
- **Chapter 7 - Conclusions and Future Work:** This chapter outlines and evaluates the work of this study. Moreover, headings forfuture work, potential expansions, and enhancements are being given.

Chapter 2

Theoretical Background

Machine Learning, Convolution Neural Networks, Residual [22]/U-NET Neural Networks, basic functions, and other essential theoretical backgrounds are represented below.

2.1 Artificial Intelligence, Machine Learning & Deep Learning

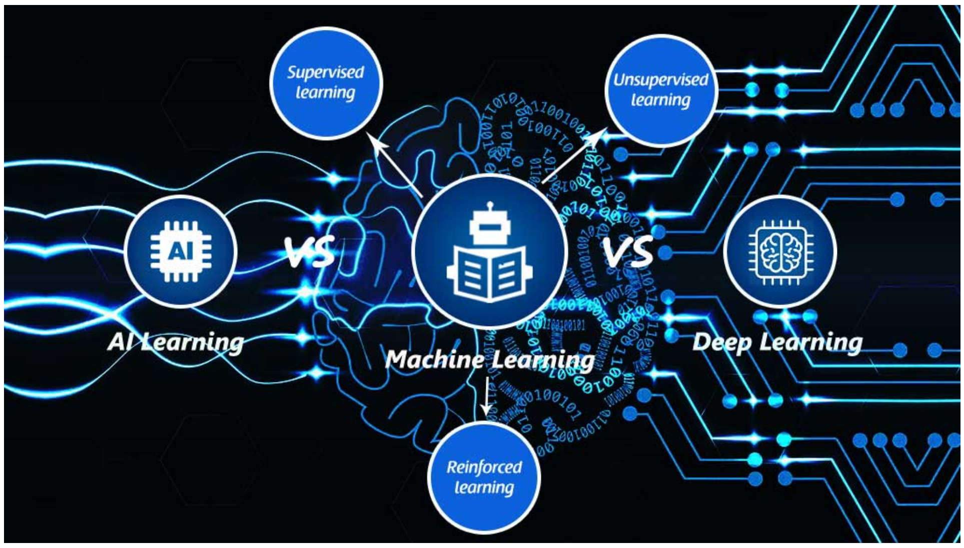


FIGURE 2.1: Artificial Intelligence, Machine Learning & Deep Learning - Western Science(UWO computer science - www.csd.uwo.ca)

Firstly, Artificial intelligence refers to a 'virtual' human-made processing system that can learn, plan, and process different kinds of data like as a physical organic brain would do. This artificial creation's primary goal is to exceed the human brain possibilities and get specialized in specific tasks and data genres with enhanced performance and accuracy compared to any other existing system. The static and predetermined way of coding does not apply in this field of machine learning(a subset of Artificial Intelligence) where everything is dynamic and unexpectable. On the other hand, deep learning is a sub-set of Machine learning, analyzing distinct factors by mimicking the human neural system. It uses a multi-layer neural network with a gradually increasing abstraction as the non-linearity of input data transformations increases.

Creating such a system is a complex task that needs a lot of expertise and specialized technique to be a successful final product with all the abilities mentioned earlier. In recent years, engineers have focused on developing such technologies that can help in any aspect of our lives, from solving drug complex algorithms for pharmacy companies to finding patterns in important data gathered for analysis.

Prediction is the main characteristic of the machine learning that is assumed to be trained on related data-set of samples to recognize patterns and features on the subject. The word training means that the system can learn via a massive data pool that is provided as input and then, according to its output, the system must be able to compare the expected outcome with its prediction and update accordingly the specific sectors of its structure, which are responsible for any possible output errors. That is why the brain is being used as the base model for artificial intelligence, utilizing the same neuron to neuron data transfer technique and the same supervised learning procedure.

Supervised learning means that someone needs to repeat -many times- "This is an apple" in order to this artificial creation, empower its corresponding neurons and finally increase the output probability to specific values that mean "It is an apple." Like a baby that learns the shape, color, and feel of an apple for the first time, parents need to repeat many times, "That is an apple," approximately a system like that can do the same.

Unsupervised learning means that there is no so-called 'teacher' user/person who instructs or provides any information to the program. The neural network needs to search and discover patterns of interest by splitting data into multiple classes by shape/type similarities. Some useful case of unsupervised learning is sales forecasting, special discounts on customers by analyzing that person's historical habits or even spotting potential risks and warnings on many different sectors.

Reinforcement Learning includes trial learning. It works by providing a positive or negative signal to learn by avoiding patterns that drive it to a disaster. Humans can also be effective in reinforcement learning. For example, a small kid can learn that the stove is dangerous by touching it for the first time and receiving the feeling of pain, which is a negative reinforcement feedback signal. Games, robots, self-driving can also work and learn effectively with the reinforcement learning technique.

2.2 Simple Neural Network

Neural Network is the function that mimics the process of how the brain works, and it is the heart of deep learning, which is useful everywhere since the hardware can learn. Therefore it can be used from self-driving cars, air-based models, AI navigation, and mapping to playing video games and all that by achieving the highest possible results. For example, having a data-set of zeros and ones dispersed on a 2-D X-Y axis(Figure 2.2), a neural network can effectively find boundaries, separate and make up some groups for these two different classes of data.

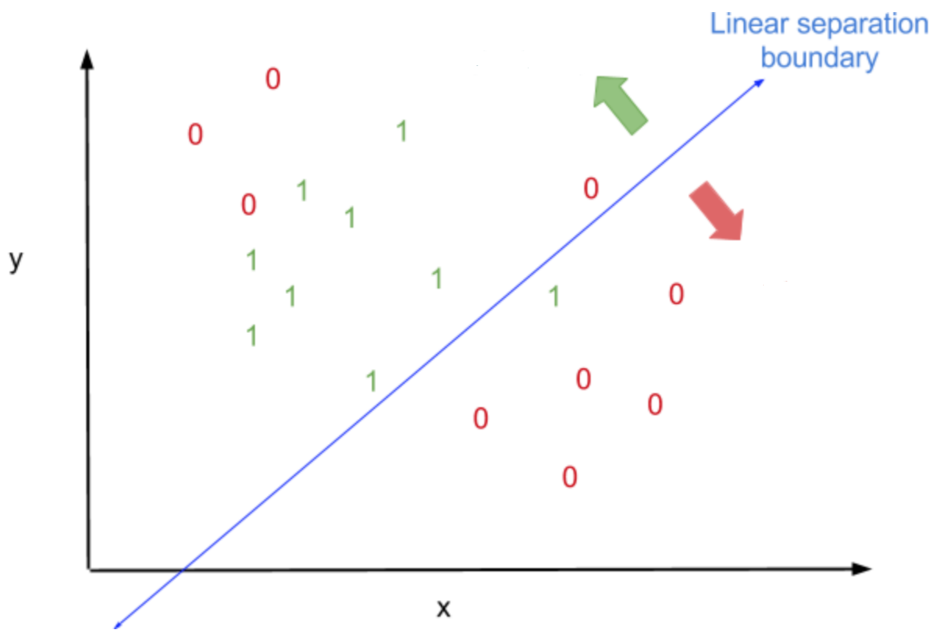


FIGURE 2.2: Classification into two different groups - Udacity Course 188

Linear Boundary Line: The drawn line above has equation:

$$a * X1 + b * X2 + C = 0. \tag{2.1}$$

With that equation is the method grouping or prediction since is able to divide into groups of data by using the equation: $Pr = Prediction(X1, X2) = aX1 + bX2 + C$, then it is possible to group each element just by following the simple rule:

$$Class = \begin{cases} 1 & Pr > 0 \\ 0 & Pr \leq 0 \end{cases} \tag{2.2}$$

The equation described above can also expressed as a linear equation:

$$Wx + b = 0, \qquad where\ \ W = (a, b) = (w1, w2)\ \ and\ \ X = (X1, X2). \tag{2.3}$$

To sum up, the main goal is to find an optimal linear boundary line that keeps most of the ones(green) above it and most of the zeros(red) below it. It can also be extended to 3-D space or even higher in order to express a more complex data system.

2.2.1 Perceptron

This algorithm, aforementioned before, can be described as a perceptron called a small neural network unit that does certain computations to detect patterns and features in the input data.

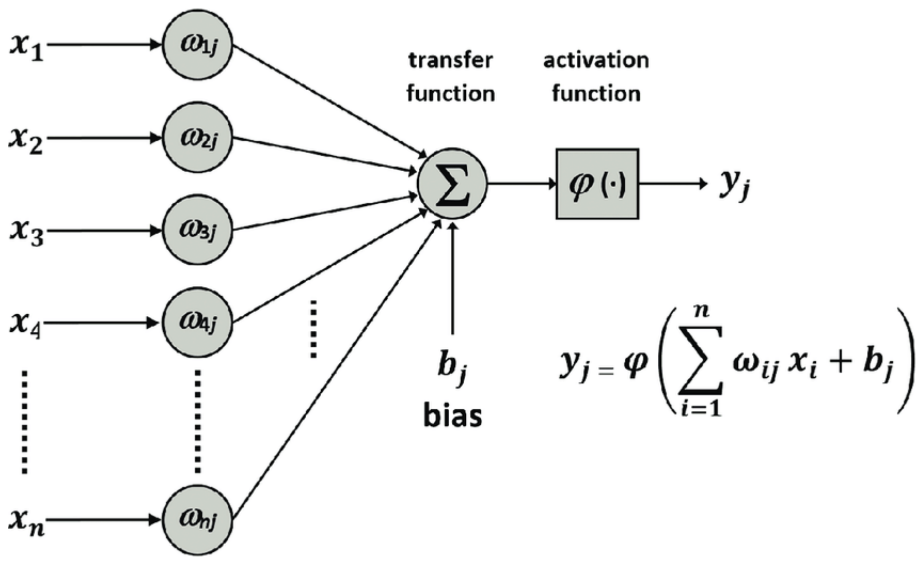


FIGURE 2.3: Scheme of a Perceptron - www.researchgate.net

There are many kinds of activation functions (or step functions) that can 'wrap' and 'fix' the result into something more rigid and desirable.

Non - Linear Regions

These regions are some areas that cannot be grouped or described from a simple linear function and need a more complex algorithm that is more generalized to other types of curves.
Given that every time the algorithm is initiated, a random line is generated on the axis system, this 'complex' line needs to be controllable from the points according to their position in axis-system space. A more "flexible" line can be possible only by defining a new type of function, which is called the error function that can tell us how far is the real solution from the target or just an excellent approximation of the solution(The most of the times it is impossible to improve the outcomes more than a specific value). So this error function can 'feel' the distance of its target and the direction it must turn/move in order to minimize the error/distance just by correcting itself (Changing $W = (w_1, w_2, ..., w_n)$ field) so its angle and shape can change to a more proper step that has lower error than the previous one.

Summarized algorithm of a Perceptron

If point i is in wrong group, then update:

$$w_i = w_i \pm (a * X_i), \quad b = b \pm a \tag{2.4}$$

where a is the rate of the whole changes that affect the final grouping ‘line’ known as learning rate.

It is also a crucial step for us to move to continuous error functions (hence from discrete predictions to continuous prediction), and that is the point where the activation functions are essential because an activation function (e.g., sigmoid that will be analyzed later), can ’break’ and normalize the results into many possible groups/classes.

Gradient Descent

The only difference with the perceptron algorithm is that here in gradient descent [23], all axis system points contribute to the final angle and shape of the ’line’ even after they are included in the final ’right’ class. To be more precise, an example can show the exact algorithm that gradient descent utilizes: Consider a 3-dimensional graph of a ’mountain,’ and the main goal is to move to the lowest area(with the lowest error) starting from the top of the mountains(high error-cost). So, the main priority is to find the path that gives the steepest descent(negative gradient) that also points the right direction, and then after some steps, decreases the cost function as quickly as possible.

Learning Rate

The learning rate is the number responsible for the descent course or simply the size of the steps that are going to be made to succeed by reaching the lowest possible value. Generally, it is recommended for the learning rate to be a tiny number since smaller and more stable steps are always a safer way to success.

Over-fitting

The over-fitting problem is a common obstacle in neural networks training, and it happens when we try to improve and push the training even more. As a result of that pushing for excellence, a phenomenon called over-fitting emerges, which means that the results of these specific predictions look great and close to zero-error outputs at first. However, it is crucial to understand that there was a mistake by training the network for some specific types of input and not generalizing the ’knowledge’ of the network.

Dropout

Dropout is a very effective way to prevent over-fitting. When a network manifests such a behavior(over-fitting), some of the neurons are very strong, meaning that they have relatively large values ($w_1, w_2, \ldots, w_n$) in comparison with some others that have close to zero weight values. As a result, the larger weights will dominate during the training procedure while other parts with much weaker neurons do not play much of a role. By adding the dropout feature to the network, the 'strong' parts of the network are deactivated, so the other sectors can also affect the results and eventually get trained. If that technique is generalized, a new system is made that can randomly deactivate (making zero) some different parts of the network for each epoch. Assuming there is a 20% chance for each node to get deactivated, it means that each node will get the same treatment after a large number of training epochs.

2.3 Convolutional Neural Networks

Convolutional neural networks make up a category of Neural Networks that have proven very effective in classification and image recognition problems. The primary purpose of a convolution is to extract essential features from a given input image. This is possible by reducing the images into a more manageable form to process without losing the spatial and temporal dependencies after many relevant filters are applied to the input image. That crucial difference that makes convolutional neural networks more powerful and efficient than dense layers is that dense layers are fully connected, meaning that every single node of one layer is connected to every node in the previous and next layer. Convolutional layers' nodes, in contrast, are connected only to a small subset of the previous/next layer. This network type also works with learnable weights and biases exactly like dense layers (that also need a random parameter initialization as a first step). A more intuitive explanation is that convolutions disassemble an image, gather all the 'useful' patterns, and then use all these features to reform the result into something familiar and known to them. A simple example is the face recognition [24], where an input image-face go through many layers that disjoint into horizontal, vertical, diagonal lines and other patterns which are pushed even deeper in the network, applying on them more and more convolutions so they can eventually form the eyes, mouth, nose. After applying the last convolutional layers, they reach a final, more recognizable shape to the network.

2.3.1 Convolutional Layer

Every layer consists of many possible kernels(filters) that will apply the algorithm 1 to the input image. Someone must understand that each convolution layer takes as input the previous convolution layer output; thus, it can discover even more patterns within the patterns.

$^{*}Note$:*Convolution and Maxpool use the following equations for calculating the output size*(OH, OW) *given the input size*(H, W), *stride*(S) *and kernel size*(KH, KW)

$$OH = \frac{H - KH}{S} + 1 \tag{2.5}$$

$$OW = \frac{W - KW}{S} + 1 \tag{2.6}$$

The equation below shows the simplest and most straightforward version of a convolution.

$$f(t) * g(t) \triangleq \int_{-\infty}^{+\infty} f(\tau) * g(t - \tau) d\tau, \tag{2.7}$$

where $g(\tau)$ at first step must be reflected to $g(-\tau)$.

In the case of multiple input channels, there is addition through all the results (of dot products procedures) per input channel(which must be equal with the kernel input channels dimension).

Convolution - Algorithm

The Algorithm 1 bellow performs a 2-D Convolution on 3-D array inputs. The parameter stride has a fixed value to 1 since every type of convolutions has been used from the neural network has a stride of 1. As described before, *input* has 3 dimensions. The first one is the number of image channels followed by height and width. The first input image is black and white, so it has only one channel. After the first convolution block, the channel dimension increases as the features are extracted and saved in each separate channel. The *kernelSize* and *padding* are the hyper-parameters of convolution layers given each time as setting up keys before the actual computation begins. And finally, *weights* and *bias* make up the network's parameters. This algorithm's output has the shape of the input image(3 dimensions) with the actual sizes per dimension calculated, as shown below.

Algorithm 1 Convolution Algorithm

```
1:  procedure CONVOLUTION(input, weights, bias, kernelSize, padding)
2:      $stride \leftarrow 1$                                                    ▷ Fixed Stride
3:      $OH \leftarrow ((input.height + 2 * padding - kernelSize)/stride) + 1$
4:      $OW \leftarrow ((input.width + 2 * padding - kernelSize)/stride) + 1$
5:      for k:=0 to (input.channels-1) do                                      ▷ Zero padding input
6:          for i:=0 to (padding-1) do                     ▷ If padding==0, It doesn't enter here
7:              for j:=0 to (input.width + 2*padding-1) do
8:                  $array(k,i,j) \leftarrow 0$
9:                  $array(k,j,i) \leftarrow 0$
10:                 $array(k, input.width + 2 * padding - 1 - i, j) \leftarrow 0$
11:                 $array(k, j, input.width + 2 * padding - 1 - i) \leftarrow 0$
                                                          ▷ Fill the center of the padded array
12:         for i:=padding to ((input.width+2*padding-1)-padding) do
13:             for j:=padding to ((input.width+2*padding-1)-padding) do
14:                 $array(k,i,j) \leftarrow input(k, i - padding, j - padding)$
15:     for oc:=0 to (weights.filters -1) do                   ▷ #Filters = #Output channels
16:         for oh:=0 to hOut-1 do
17:             for ow:=0 to wOut-1 do
18:                 $sum \leftarrow 0$
19:                 for ic:=0 to input.channels-1 do
20:                     for i:=oh to (oh-1+kernelSize) do
21:                         for j:=ow to (ow-1+kernelSize) do
22:                             $sum \leftarrow sum + array(ic,i,j) * weights(oc,ic,i,j)$
23:                 $output(oc,oh,ow) \leftarrow pixel + bias(oc)$
24:     return $output$
```

* *Note*: *Flipping the kernel is not necessary because the CNNs can adapt accordingly having learn-able weights, so technically, a flipped kernel does not change anything. In machine learning, all the applications that make use of the convolution are doing a cross-correlation. Convolution in the image processing & mathematics field means*

More options and hyper-parameters for even more in-depth customization of a convolution layer:

Strides is the number of pixels-slots the kernel sliding-windows [25] will skip. So with a value greater than one, the final result will be a reduced matrix.

Zero Padding

- *Valid:*That is the easiest and most straightforward way to convolve because there is no pre-processing on the input image. It is assumed that all dimensions are valid so that the input image fully gets covered by the filter and strides specified.
- *Same:* According to the stride setting and the input image's size, the appropriate zero paddings will be applied to produce an output precisely the same size as the input.
- *Kernel Size:* There is a heavy preference on odd$(3\times3, 5\times5, ...)$ kernel size over even$(2\times2, 4\times4, ...)$ kernel size. That is because if we choose an even kernel, the output result will suffer from aliasing problems. By choosing a kernel size to be $2n+1$ for both dimensions, we create a symmetrically shaped kernel with each anchor's side plus the center anchor pixel.

**Note:* 1×1 *kernel size is often used for channel shape corrections at the end of image processing so it can match an expected result. It is generally not considered an optimal filter size for a typical convolution layer since the result will have no information from the adjacent pixels as the features extracted would be fine-grained and local.*
**Note: Strides & padding are assumed that work in the same way for both X-Y dimensions of the input in which the convolution is applied.*

2.4 Max-pooling

Max Pooling is a simple down-sampling [26] strategy by choosing the maximum or average value(Average Pooling) within a matrix. A typical example of max pooling size is the 2×2 non-learn-able kernel with strides of 2(for each dimension). Pooling is a beneficial step after each convolutional layer since it can keep the important information per stride and reduce the size of the next layer input image to half(or greater, depending on kernel size) so the processing will be less complex thus faster.

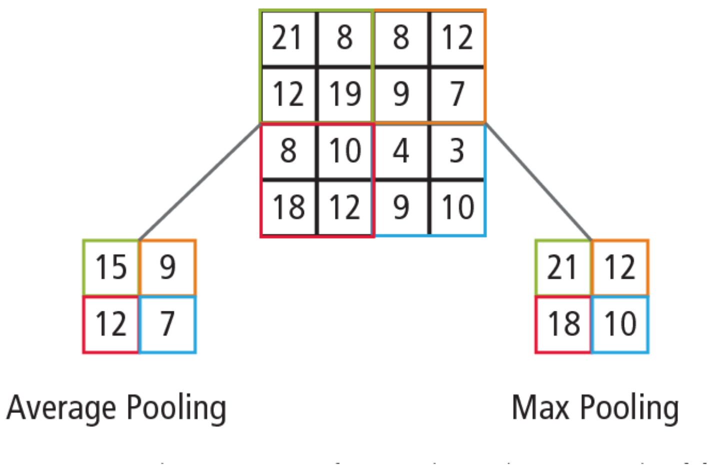


FIGURE 2.4: Max/Average Pooling - www.semanticscholar.org, Image By Hijazi Kumar

According to the 2.5,2.6, the equation for calculating maxpool, given the input, stride and kernel size, is defined as:

$$MaxPool(c,i,j) = \max_{0\le kh\le(KH-1),0\le kw\le(KW-1)} Input(c, i*stride+kh, j*stride+kw),$$
$$\text{for c = 1, 2, ..., Channels,}$$
$$\text{for i = 1, 2, ..., OH,}$$
$$\text{for j = 1, 2, ..., OW} \tag{2.8}$$

The main Algorithm of Max-pooling is described below:

Algorithm 2 MaxPool Layer

```
1: procedure MaxPool Layer(input, kernelSize, stride)
2:     $hOut \leftarrow (input.height - kernelSize)/stride + 1$
3:     $wOut \leftarrow (input.width - kernelSize)/stride + 1$
4:     for i:=0 to (input.channels-1) do
5:         for j:=0 to (hOut-1) do
6:             for k:=0 to (wOut-1) do
7:                 $max \leftarrow -\infty$
8:                 for l:=j*s to (j*s+kernelSize-1) do
9:                     for m:=k*s to (k*s+kernelSize-1) do
10:                        $curPixel \leftarrow input(i, l, m)$
11:                        if max < curPixel then
12:                            $max \leftarrow curPixel$
13:                $output(i, j, k) \leftarrow max$
14:    return $output$
```

2.5 Activation Functions

Activation Functions [27] are based on the biological neuron data firing to control the output as 'ON' or 'OFF' depending on the input. The binary form is not the only implementation of an activation function when it is possible to drive the result into the desired value window to make more clear the range of a prediction. There are some hidden steps in each convolution layer, the activation function is one of them, and it is applied after a convolution procedure is completed.

2.5.1 Sigmoid

It is used in a feedforward network, and it can drive the output between 0 to 1. Therefore, it is specialized for models where the probability is the main goal for the output prediction.

$f(z) = \sigma(z) = \frac{1}{1+e^{-z}}$

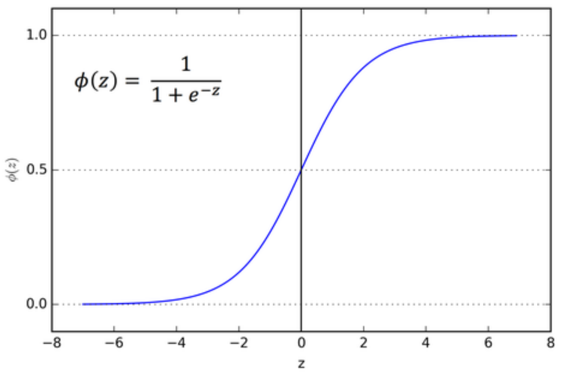
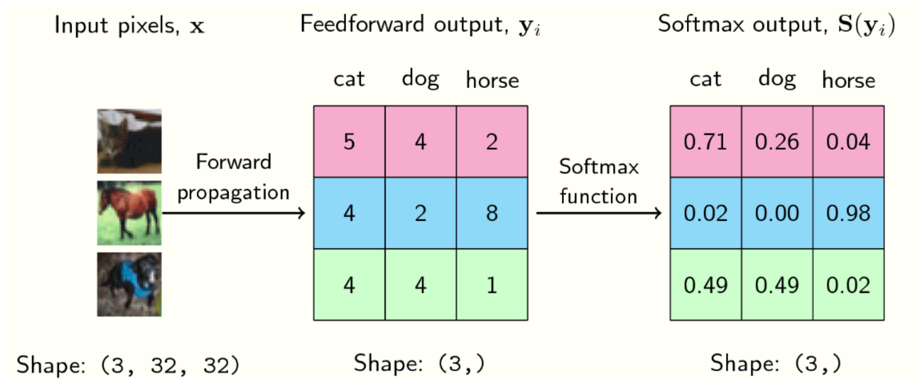


FIGURE 2.5: Sigmoid Activation Function - www.towardsdatascience.com

2.5.2 Softmax

Softmax is a more generalized logistic function [28] and it is often used for multi-class problems. It works by scaling each output between 0 and 1 with respect to the fundamental law of total probability. It is a quite popular activation function and it is often used as an activation function of the output.

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \qquad \text{for i = 1, ..., K} \quad \text{and} \quad z = (z_1, ..., z_K) \in \mathbb{R}^K$$


Input pixels, $\mathbf{x}$
Feedforward output, $\mathbf{y}_i$
Softmax output, $\mathbf{S}(\mathbf{y}_i)$
Forward propagation
Softmax function
Shape: (3, 32, 32)
Shape: (3,)
Shape: (3,)

cat	dog	horse
5	4	2
4	2	8
4	4	1

cat	dog	horse
0.71	0.26	0.04
0.02	0.00	0.98
0.49	0.49	0.02

FIGURE 2.6: Sigmoid Example - ljvmiranda921.github.io

2.5.3 TanH

Tanh is also a sigmoidal (s- shaped) and when it has a range of -1 to 1. It is used to map negative inputs close to -1 zeros at zero and positive inputs around +1.

$f(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x - e^{-x}}$

2.5.4 ReLu

ReLu is one of the most used activation functions especially in the middle/hidden area[29] of a deep neural network.

$$f(x) = \begin{cases} 0 & x < 0 \\ x & x > 0 \end{cases}$$

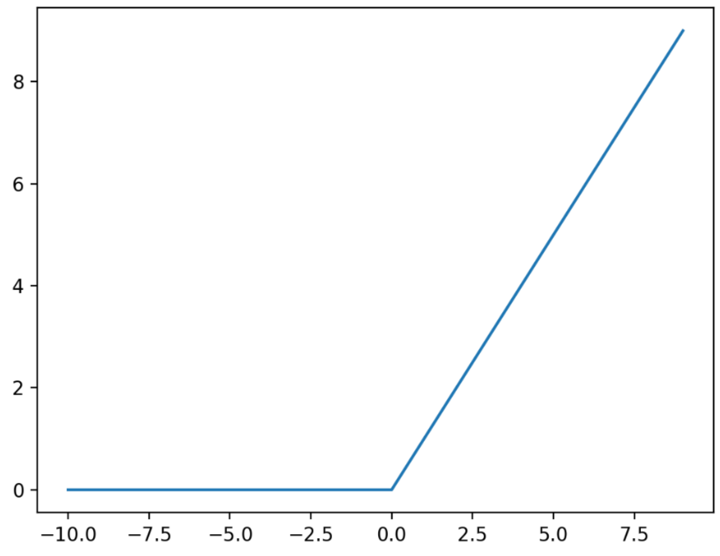


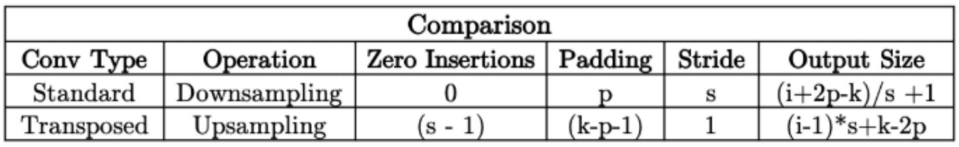
FIGURE 2.7: ReLu Function - www.towardsdatascience.com

It is a rectified linear unit that keeps as it has any positive value and makes zeros values equal to zero or less(negative). Its advantages vary from better gradient propagation, scale-invariant to high efficiency in computations because of the function's simplicity.

2.6 Transposed Convolution

Transposed Convolutions or fractionally strided convolutions or deconvolutions [30] are used for upsampling the input image to the desired size. It attempts to retrieve the previously downsampled image with a simple technique by just reversing the convolution's effect. To be more specific, a convolution with $stride = 2$(or the combination of a convolution with stride one and a max pool 2×2) will reduce the output size to half. The need for upsampling is coming from the fact that there are architectures that can reconstruct input with some added masks-overlays to emphasize the object of interest or even separate and categorize objects into different classes.

Many other resampling techniques like Nearest neighbors [31], bi-linear interpolation [32], bed of nails, and max-unpooling [33], but this thesis's main focus is around transposed convolution. Of course, transposed convolution suffers from chequered board effects with the main cause of these artifacts be the uneven overlap at some parts of the image. As described previously, the downsample convolutions often have an odd size of kernels for that exact reason of asymmetrical kernel problems. The minimum size of kernel 2×2 and a stride of 2 is recommended to reduce these problems. Below, a table shows how to calculate the zeros insertions, which are the empty slots between the input pixels and the padding around the input pixels.



Comparison					
Conv Type	Operation	Zero Insertions	Padding	Stride	Output Size
Standard	Downsampling	0	p	s	(i+2p-k)/s +1
Transposed	Upsampling	(s - 1)	(k-p-1)	1	(i-1)*s+k-2p

FIGURE 2.8: Downsampling - Upsampling Equations - www.towardsdatascience.com

The following transposed convolution algorithm is based on the paper Optimizing CNN-based Segmentation with Deeply Customized Convolutional and Deconvolutional Architectures on FPGA [34] .This algorithm is FPGA-friendly utilizing many independent additions, especially when kernel size is the same as the stride value.

Algorithm 3 Transposed Convolution

```
 1: procedure DECONVOLUTION(input, weights, bias, kernelSize, padding)
 2:     $s \leftarrow 2$                                        ▷ Fixed Stride
 3:     $OH \leftarrow 2 * input.height$
 4:     $OW \leftarrow 2 * input.width$
 5:     for oc:=0 to (weights.filters-1) do
 6:         for oh:=0 to (OH-1) do
 7:             for ow:=0 to (OW-1) do
 8:                 $output(oc, oh, ow) \leftarrow bias(oc)$
 9:         for ic:=0 to (OH-1) do
10:             for x:=0 to input.height do
11:                 for y:=0 to input.width do
12:                     for k:=0 to KernelSize do
13:                         for l:=0 to KernelSize do
14:                             $output(oc, x*s+k, y*s+l) \leftarrow output(oc, x*s+k, y*s+l) + input(ic, x, y) * weights(oc, k, l)$
15:     if padding>0 then        ▷ Remove the elements in the border of size padding
16:         for oc:=padding to output.channels-1 do
17:             for oh:=padding to output.height-1-padding do
18:                 for ow:=padding to output.width-1-padding do
19:                     $outputC(oc, oh, ow) \leftarrow output(oc, oh, ow)$
20:         return $outputC$
21:     else
22:         return $output$
```

The following Transposed Convolution Algorithm, which is using the upsampling equations of the Figure 2.8, is not recommended because with this method (which is also illustrated in Figure 4.8 in Chapter 4), there many 'empty' areas(in white) representing all the added zeros to the original input image. Recreating such an algorithm, which is CPU-oriented and optimized, onto FPGA will jeopardize the board's parallel capabilities just by executing a computation that involves a considerable amount of data dependencies between columns and rows of the output image.

Algorithm 4 Transposed Convolution 2(Not Recommended)

```
1: procedure DECONVOLUTION2(input, weights, bias, kernelSize, padding)
2:     $s \leftarrow 2$                                          ▷ Fixed Stride
3:     $OH \leftarrow s * input.height$
4:     $OW \leftarrow s * input.width$
5:     $OH_temp \leftarrow input.height * s - 1 + 2 * padding$
6:     $OW_temp \leftarrow input.width * s - 1 + 2 * padding$    ▷ Adding zeros between elements + zero padding
7:     for c:=0 to (input.channels-1) do
8:         for h:=0to (input.height-1) do
9:             for w:=0 to (input.width-1) do
10:                $temp_matrix(c, h * s + padding, w * s + padding) \leftarrow input(c, h, w)$
11:    $weights \leftarrow$ rotate180(weights, axis= (2,3))    ▷ Rotate right 2 times for each filter
12:    output = Convolution(temp_matrix, weights, bias, kernelSize, padding= 0)
13:    return $output$
```

2.7 Loss Functions

A neural network via training determines what patterns need to detect based on the loss function. The selection of a loss function between a large pool is a crucial step since it must fit the application and meet our standards to give feedback with a valuable and accurate representation of the neural network's performance so we can fairly evaluate it.
Since our main target is the reconstruction of the input image and the masking, below there are some relevant loss functions that are made for that exact reason of per-pixel comparison(label vs. prediction which both have the same dimensions) finding the element-wise deviation.

Pixel accuracy is a straightforward metric for anyone to understand the intuition behind the semantic segmentation accuracy. However, it gives poor metric results since it can return a high accuracy score like 95% when the actual representation to the human eye is wrong and possibly not even close the actual goal. For example, in Figure 2.9, the ground truth is very close to the prediction for the pixel values comparison with a score of more than 95%, but the actual prediction makes no sense and its useless.

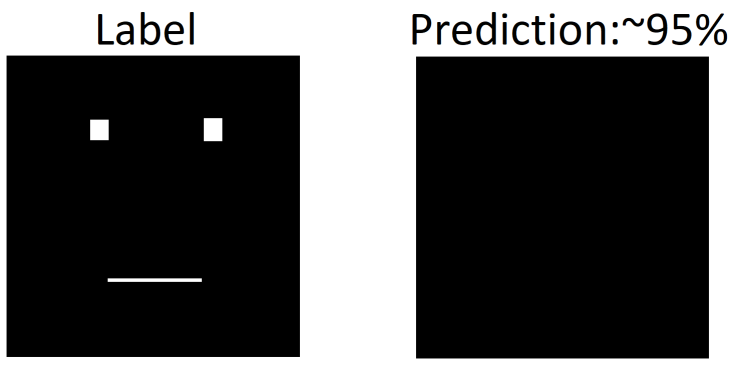


FIGURE 2.9: Pixel Accuracy - High accuracy percentage with poor real-world usage

Intersection over Union(IoU) - Balanced loss :

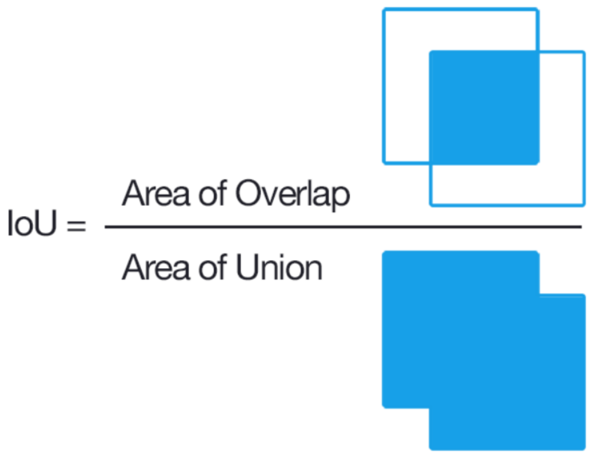


FIGURE 2.10: Intersection over Union calculation visualized - WIKI

This loss function, also known as the Jaccard Index [35], works by increasing the gradient of values/samples with high Intersection over Union(IoU) while decreasing these with lower IoU. As shown below in the figure, the amount of overlapping($A \bigcap B$) between prediction and the actual label(ground truth) over the Union of these two ($A \bigcap B$) is measured. This metric is represented in percentage, with 0% be the zero overlapping(failed prediction) and 100% the perfect match when we compare the network prediction and the ground truth area.

$$J(A,B) = \frac{|A \bigcap B|}{|A \bigcup B|} = \frac{|A \bigcap B|}{|A| + |B| - |A \bigcap B|} \tag{2.9}$$

Dice coefficient(F1 Score) [36] is a very similar loss function with the IoU represented above. The only big difference is that the numerator (area of overlap between prediction and ground truth) is multiplied by 2. So we get :

$$DC = \frac{2|A \bigcap B|}{|A|+|B|} = \frac{2TP}{2TP+FP+FN} \tag{2.10}$$

It can also be defined as a loss function(result is now an error value), by using probabilities:

$$DC(p,\hat{p}) = \frac{2p\hat{p}}{p+\hat{p}}, \tag{2.11}$$

when for $p = \hat{p} = 0$ the appropriate handling is required. A simple fix for that problem is the addition of '+1' as shown below:

$$DL(p,\hat{p}) = 1 - \frac{2p\hat{p}+1}{p+\hat{p}+1}, \quad where \quad p \in \{0,1\} \qquad and \qquad 0 \le \hat{p} \le 1 \tag{2.12}$$

The main effect of the '+1' fix is the shift of the result value window from [0, 1] to [0, 0.5] Loss function often return tensors, but with the above implementation using reducing sum over the matrix elements its possible to remove '+1'. So the scalar result is :

$$DL(p,\hat{p}) = 1 - \frac{2\sum p_{h,w}\hat{p}_{h,w}}{\sum p_{h,w} + \sum \hat{p}_{h,w}} \tag{2.13}$$

Example : Let **P** be the real Image, $\hat{\mathbf{P}}$ the prediction and **L** the result of the loss function.

$$\mathbf{P} = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}, \qquad \hat{\mathbf{P}} = \begin{bmatrix} 0.5 & 0.6 \\ 0.2 & 0.1 \end{bmatrix}$$

$$Then \quad \mathbf{L} = \begin{bmatrix} -\log(0.5) & -\log(0.6) \\ -\log(1-0.2) & -\log(1-0.1) \end{bmatrix}$$

The result is :

$$\mathbf{L} \approx \begin{bmatrix} 0.6931 & 0.5108 \\ 0.2231 & 0.1054 \end{bmatrix}$$

Binary Cross Entropy also known as *Sigmoid Cross-Entropy loss* because its a mix of the classic logistic sigmoid plus the cross entropy loss.The loss is calculated by computing the following average(between all the pixels) :

$Loss = -\frac{1}{OutputSize}\sum_{i=1}^{OutputSize} y_i \log \hat{y}_i + (1 - y_i)\log(1 - \hat{y}_i),$

where y_i is the pre-calculated scalar i^{th} (logarithm-softmax-logistic/sigmoid) output value with range from 0 to 1. The classification problem must have two classes where the binary cross entropy loss can be applied. This loss, is also equal to the average value of the categorical cross entropy on many 'binary' tasks, by meaning that each task/problem has only two possible classes.

2.8 Residual Neural Networks

This work is built on the main idea of *Residual neural networks.* This neural network category has a unique architecture that utilizes shortcuts(or skip connections) to jump over some layers. The general idea is based on the biological brain analog of having skip neurons where needed. This feature is beneficial, especially in deep neural networks where the networks' actual depth is quite large, so their ability to push forward information while maintaining the impact/weight/importance is almost impossible. The last one also applies to the back-propagation steps, causing the known problem of the vanishing gradient.

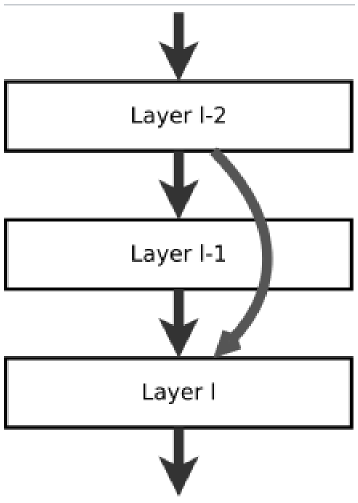
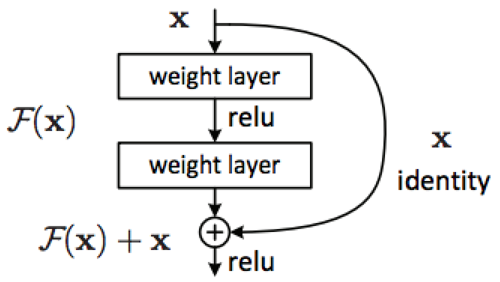


FIGURE 2.11: Normal & Skip Connections/Neurons - Wiki

With that skip connection trick, it is possible to amplify the previous' layer value and make it 'visible' to layers way more profound in the network, when for a simple deep neural network, only the weights of the closest neighbors can learn to adapt. This technique seems useful during the training process by minimizing the training time and accumulated work because of the fewer layers during the back-propagation process.


x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu
x
identity
Figure 2. Residual learning: a building block.

The block above shows that the $y = F(x) + x$ combines the information of the two previous layers pushing it through as input to the next one. Some times, there is no dimension match between the combined information, so the fix to that problem is just a small reshaping/shrinking reformation.

2.9 Computer Vision Tasks

Computer vision problems can vary with multiple levels of difficulty and depth(Figure 2.12) or from a coarse-grained down to a more fine-grained understanding.

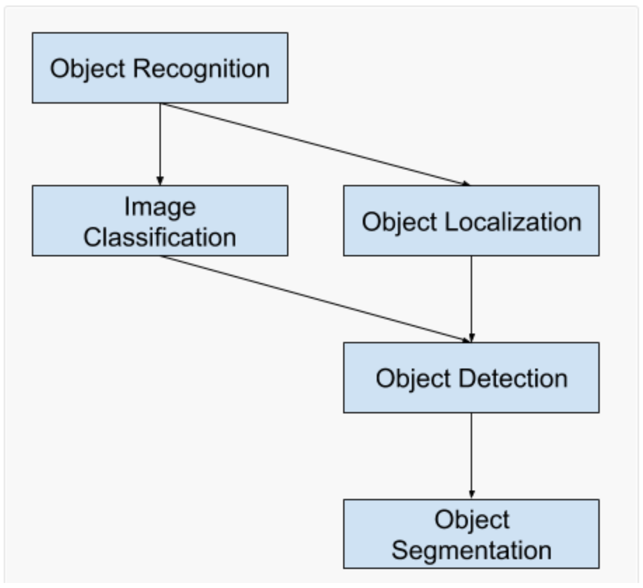


FIGURE 2.12: Computer Vision Tasks - www.machinelearningmastery.com

Image Classification is the core task in computer vision, where a simple input image is expected from the computer to label it with a specific tag. This prediction can output the type or class of the object in the input image by examining all the available features it can extract from it.

Object Localization: Locates the object of interest anywhere in the image and indicate it with a bounding box. This task includes the Image classification plus the localization of the main target in the image. It is important to note that this method can only mark and locate one object's presence per image.

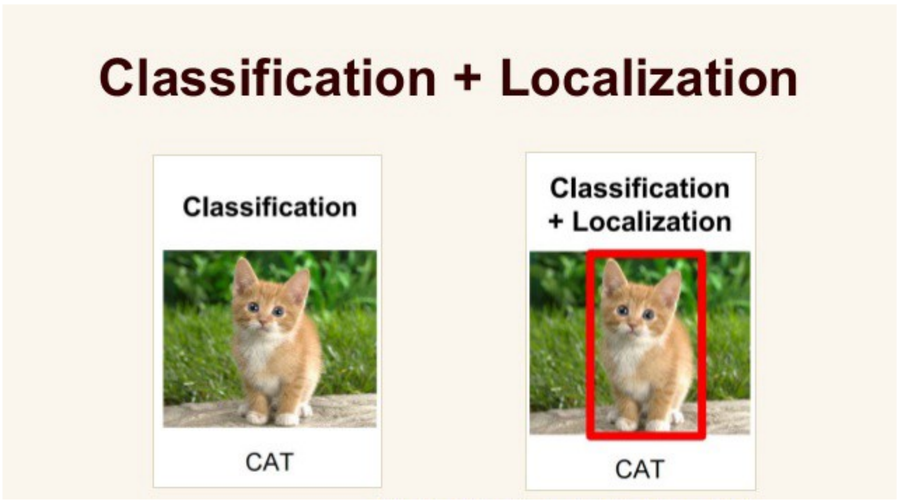


FIGURE 2.13: Classification with Localization - theaisummer.com

Object Detection extends the previous task by supporting multiple object classification in an image and the ability to localize their exact position in the image. So the output of this network is an image with more than one labeled bounding boxes showing all the available objects of interest, which can be classified in a different category/type/class.

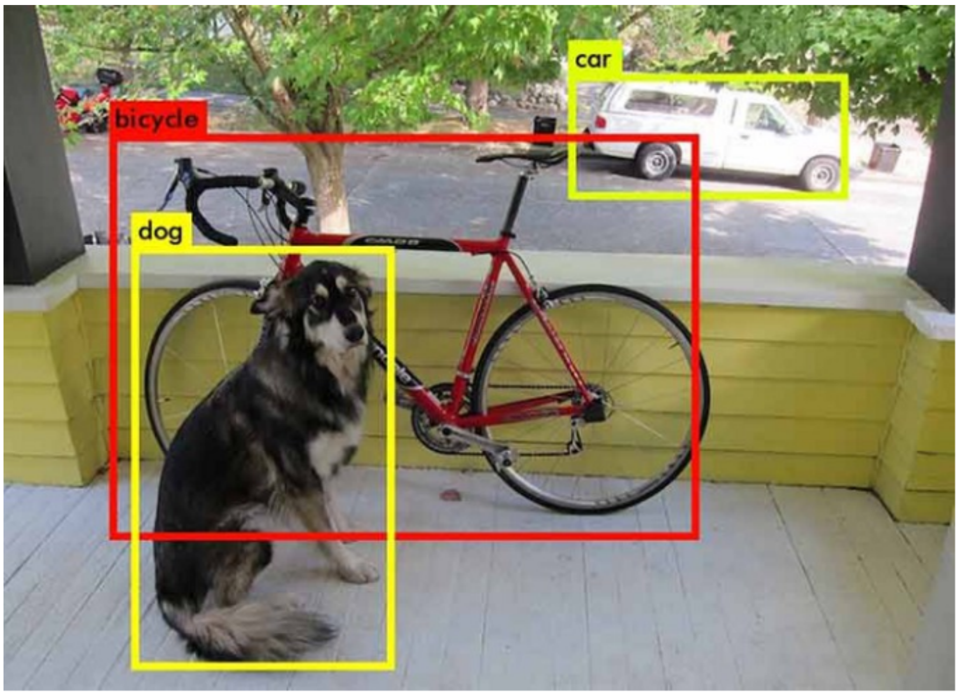


FIGURE 2.14: Object Detection - docs.nvidia.com

Semantic Segmentation is a further extension of the tasks mentioned above, and its main goal is to classify each pixel in the image to a class, so this expected output does not have any bounding boxes nor labels, just colored(classified) objects in order to distinguish them from any other located feature.

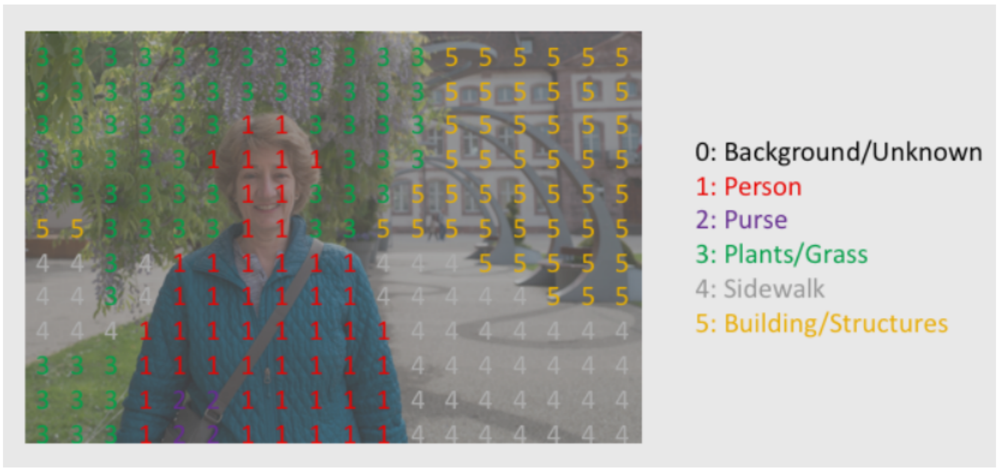


FIGURE 2.15: Semantic Segmentation Example - www.jeremyjordan.me

This work aims to model and accelerate(on FPGA) this task supporting two different classes with black and white representation. The subject contains bio-medical images gathered from MRI-CT scans and 'examined' from the neural networks for possible anomalies and disease detection so it can help humans prevent and avoid further clinical damage by this early stage pattern recognition.
In other words, semantic segmentation tasks include the support of multiple classes(more than 2) where each class is saved in a separated exclusive channel(Figure 2.16)

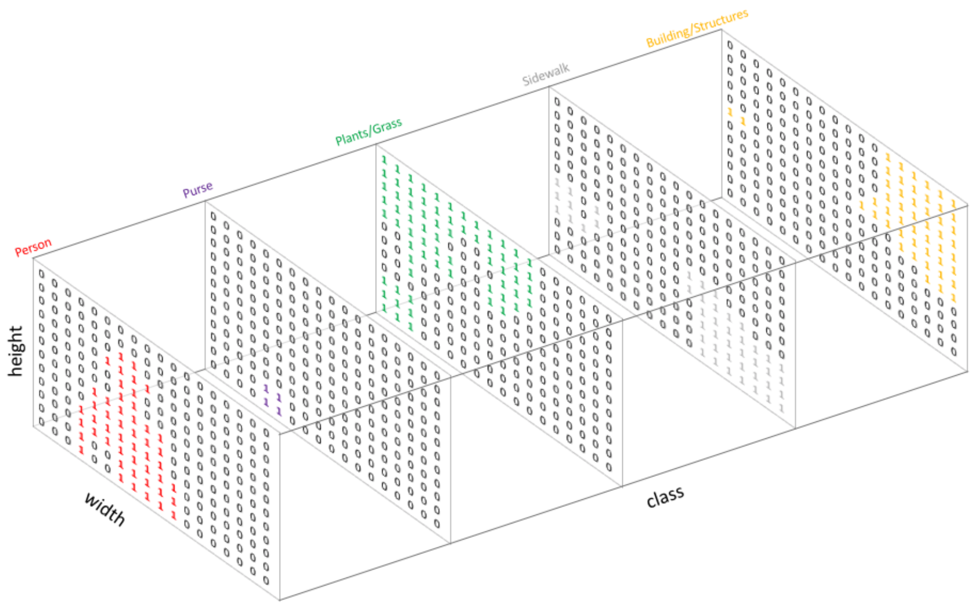


FIGURE 2.16: Class per channel - www.jeremyjordan.me

Instance Semantic Segmentation(Figure 2.17) is not treats every object of the same class as a single image entity(When simple semantic segmentation does). Instance, Segmentation treats multiple objects that belong in the same class as different/distinct instances.

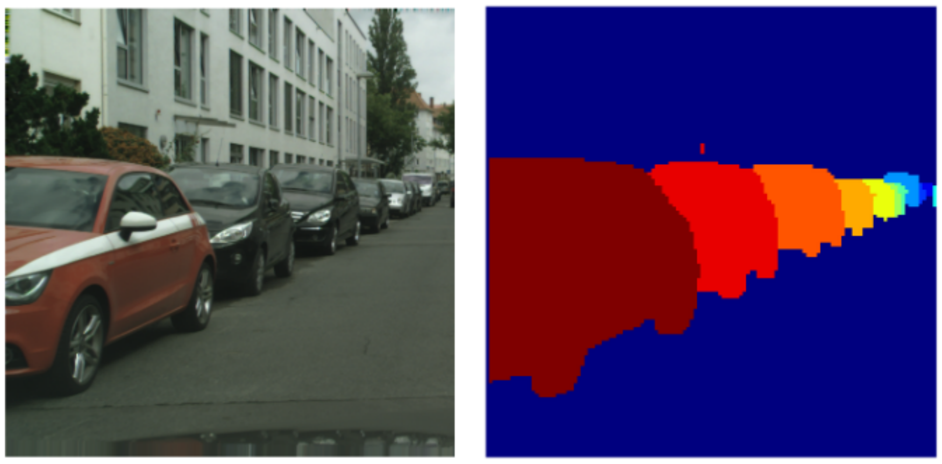


FIGURE 2.17: Instance Segmentation Example - www.arxiv-vanity.com

To achieve such a categorization of the same class object, it uses multiple channel labels(Figure) that drive and train the neural network. With more detail, for each output channel is carried a piece of specific object information like shape and locality. The output channel number must be the same as the total number of entities supported. The final fusion of the channels generates the output with multiple masked objects from different channel overlays.

Chapter 3

Related Work

In the last five years, tremendous progress on machine learning and especially on semantic segmentation has been made, using the main idea of residual networks and the famous skip connection neurons. The state-of-art architecture using the technique, as mentioned earlier, is called U-NET because of its ’U’ shape, which in this thesis will be thoroughly analyzed. Some other approaches involve high-throughput CNN accelerators adopting some unique functions and computations such as deconvolution, which resemble the classic Transposed convolution success. All these recent proposal works’ main goal is the image segmentation for bio-medical purposes and image patterns that cannot be spotted from the human eye or any other hardcoded techniques. In this field of image analysis, automation is the absolute objective and vision of the ’Artificial Intelligence’ technology. Especially CNNs are the recent trend, providing a less complicated and mature design, achieving great accuracy and performance in terms of training and prediction. The dataset for this field of semantic segmentation is very common, including some of Kaggle’s [37] famous datasets, or even smaller like CIFAR10 [38], MNIST [39], [40] that help scientists to observe the possibilities and every deficiency of the examined architecture.

3.1 Full-Resolution Residual Networks

Full-Resolution Residual Networks [41] work with a double processing stream for a pixel-level accuracy combining multi-scale context.

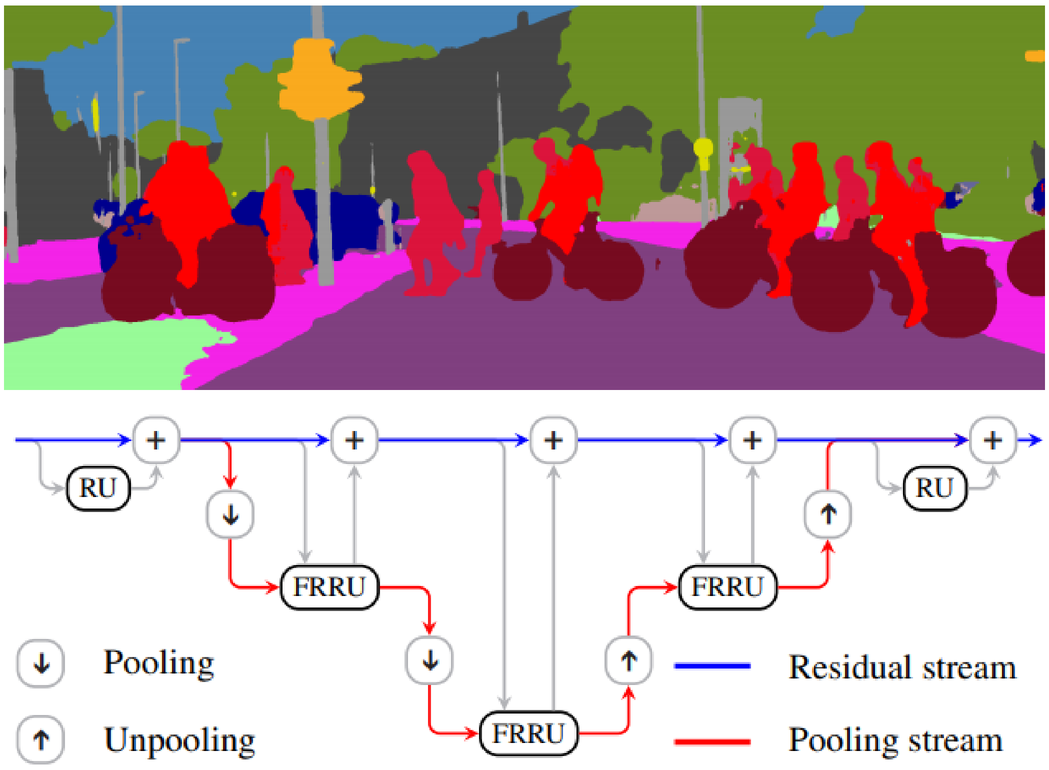


FIGURE 3.1: Full-resolution residual network & example output

The so-called residual stream (blue line) stores and transfers the image's initial input resolution through all the network to share its topological information where it is required(pooling-unpooling junctions) when the red stream undergoes each subsequent pooling and unpooling operation. The 'pairing' between blue and red streams is achieved using the full-resolution residual Units (FRRUs). In figure 3.1, each full resolution residual unit's internals is shown, splitting the unit into two boxes, the colors of which correspond to their streaming type. More specifically, the incoming residual stream is reduced via a pooling function to eventually be concatenated with the 'pooling stream' (red). The previous concatenations result is driven to a double convolution block, including batch normalization and ReLu activation functions for the feature extraction. The result of the second convolutional block has two output paths. In the first one, an un-pooling operation occurs, forming the result back to its original 'residual streaming' size (blue stream) so it can be used in the next FRRU. The other path is the output of the FRRU that will keep its destination through the network for further encoding or decoding according to its stage. This approach possible is not that efficient like U-NET because, with a first glance, the 'residual stream' (blue line in figure ..) needs to be pooled multiple times to match the FRRU conditions while keeps the 'general idea' of the image through the end while using its features.

3.2 Fully Convolutional Networks for Semantic Segmentation

Some of the latest successful Fully Convolutional Networks like AlexNet, the VGG net, and GoogLeNet can work for Semantic Segmentation tasks by transferring and fine-tuning their representations on that field. To produce accurate and detailed segmentation of the input image, the paper's proposed work [42] aims to mix the semantic information of a deep coarse-grained layer with more generalized appearance features and a shallow fine-grained layer. Translating those mentioned above to a more straightforward form, this paper [42] can efficiently train a fully convolutional network on how to make dense predictions with the only difference that each pixel output can be divided into multiple classes that eventually can be confronted as pixel-wise tasks that overall make up the semantic segmentation output.

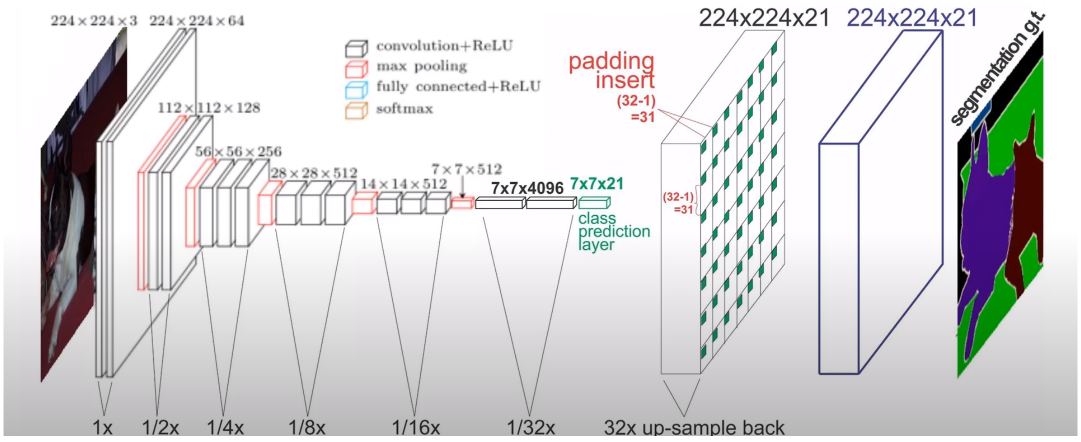


FIGURE 3.2: Fully Convolutional Networks Structure (YouTube Explanation Video)

The conversion from classification neural network into FCN that produces output maps is a trick introduced by OverFeat. Following the basic idea that higher layers correspond to specific spatial locations of the input image on which they are path-connected and called receptive fields, changing only the layer strides and filters a convolutional network can produce a result similar to the shift-and-stitch trick by OverFeat. As shown in figure 3.2, starting from the input image, many layer downsampling for further feature extraction happens as a regular fully convolution neural NatureWorks would make. The part where the trick happens is during the layer with the output of $7 \times 7 \times 4096$. At that stage, the class prediction layer uses a 1x1 convolution sliding across the $7 \times 7 \times 4096$ dimension with 21 filters(for each output class for this example). The next process consists of a learnable (not fixed) deconvolution filter (e.g., learnable bilinear upsampling), putting the results into a padded tensor of overall example size $224 \times 224 \times 21$. The purple tensor occurred by applying a standard convolution to the padded tensor. This process simulated the fractionally strided convolution with zero insertion in the input of the convolution image. As a result, the purple tensor has the final desired dimension while still storing the 21 different classes, each of them representing the confidence (in percentage) for each object. The last one can be easily converted into color(according to the class) that maps each pixel to a colored semantic segmentation output.

3.3 Mask R-CNN

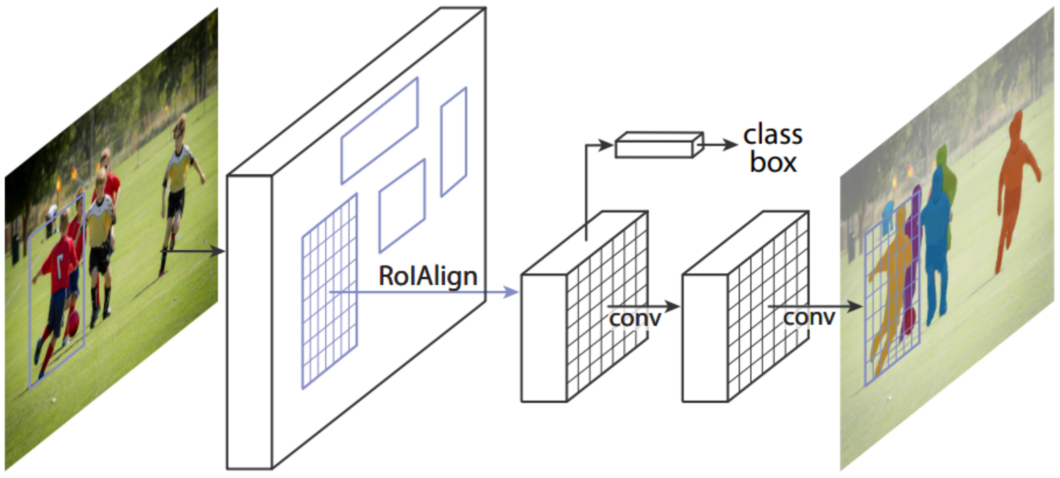
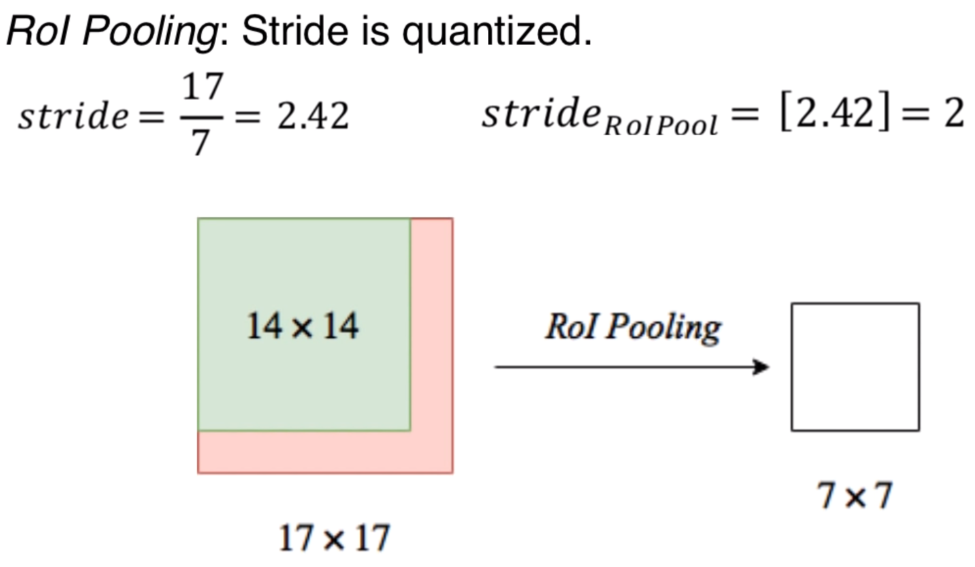


FIGURE 3.3: Mask RCNN structure

The Mask R-CNN [43] method extends the existing work of Faster R-CNN [44] that carries out the task of object detection in real-time. Faster R-CNN generates two outputs, and that is why the R-CNN detector works in a two-stage mode. During the first stage, called Region Proposal Network (RPN), the class label is produced, individualizing and classifying each object into a category. The second stage follows, assembling a bounding-box offset that is the coordinate values of the bounding box's center(simple object detection). As the Mask R-CNN adopts these functions/stages, it also extends its basic object detection and classification functionality. Faster R-CNN does not support pixel-to-pixel alignment, having some devious spatial aftereffect between network input and output images. First, during the bounding-box structuring stage, a binary mask is generated for each region of interest (RoI), while the first stage (RPN) stays intact. The mask branch encodes $m \times m$ masks (having K classes). It also essential to be noted that the traditional practice of per-pixel softmax and multinomial function of cross-entropy loss is not used(common for FCNs to semantic segmentation). In the Mask R-CNN case, a per-pixel sigmoid and an average binary function of cross-entropy loss is applied, avoiding the mask 'competition' across the K classes.

The prediction mentioned above ($m \times m$) of each RoI using FCN layers is encoding the inputs object's spatial information that can eventually be later addressed by the pixel-to-pixel relation occurred by the convolutions. Executing RoI pooling methods for small feature extraction, the results become misaligned when compared with the spatial information of the input.



RoI Pooling: Stride is quantized.
$stride = \frac{17}{7} = 2.42$
$stride_{RoIPool} = [2.42] = 2$
14 × 14
RoI Pooling
17 × 17
7 × 7

FIGURE 3.4: RoI Pooling visualization example

This alignment slip happens because the stride used by this technique is quantized. The example figure 3.4 shows that the quantized value of 2.42 has a stride of 2 for both image's height and width. Applying this RoIpool on a 17x17, the layer only considers the upper left 14x14 pixels in the 17x17 region, translated into a loss of data(of the remaining data) plus an unaligned output.

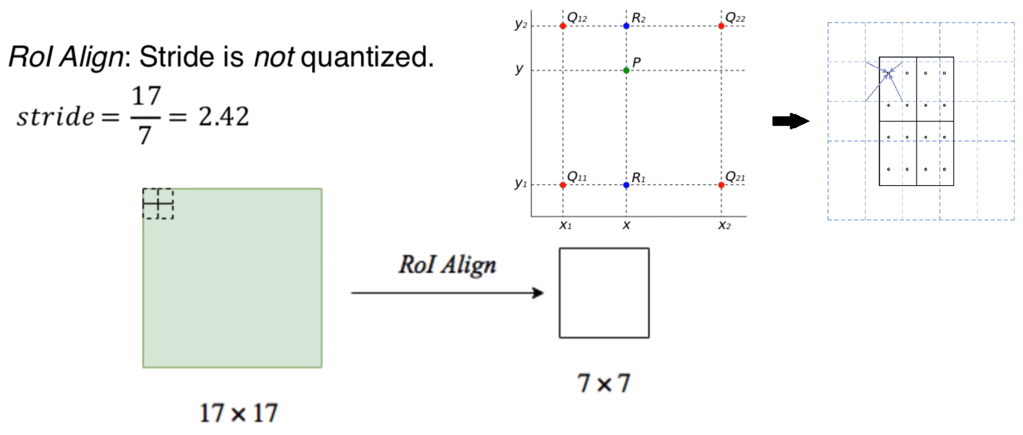


FIGURE 3.5: RoI Align visualization example (Video Explanation - Custom Image)

However, the RoIAlign (figure 3.5) layer is used to countermeasure the problem above by adequately replacing the RoIPool layer. The idea is to keep the original stride value(2.42 for the example) and then divide the generated square regions into four smaller squares. From this point, each subcell is pulled using bilinear interpolation, so the final cell value is computed either by an average or the maximum over the four sub-values.

3.4 Gated Shape CNNs for Semantic Segmentation (Gated-SCNN)

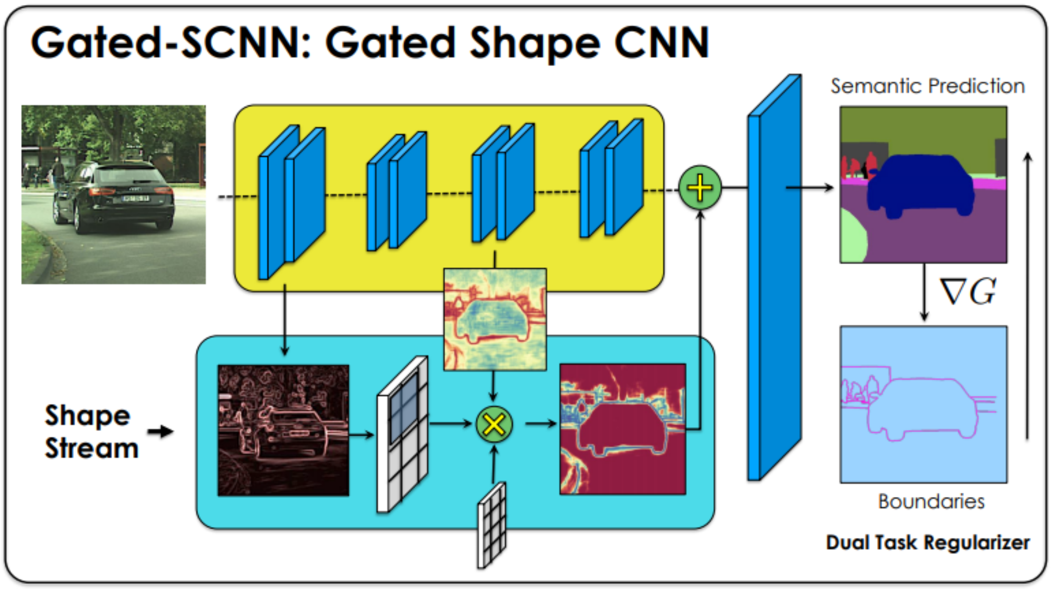


FIGURE 3.6: Gated-SCNN (GSCNN) - A two-stream CNN architecture for Semantic Segmentation

Today's Deep CNN solutions for semantic segmentation problems are processing color, texture, and shape information all at the same time inside the architecture. The proposed work of the paper Gated Shape CNNs [45], suggests a two-stream CNN path. The first path is the classic CNN for analyzing and classifying the extracted features, and it can be any feedforward, fully-convolutional network. The other path, called shape stream, works seamlessly and parallel with the classic CNN stream while processing everything about relevant boundary objects and shaping information to a specific shallow shape-stream" architecture. Results coming out from the classic stream CNN branch and shape-stream are fused at the end of processing so that the result can be accurately reconstructed according to the training knowledge of the neural network.

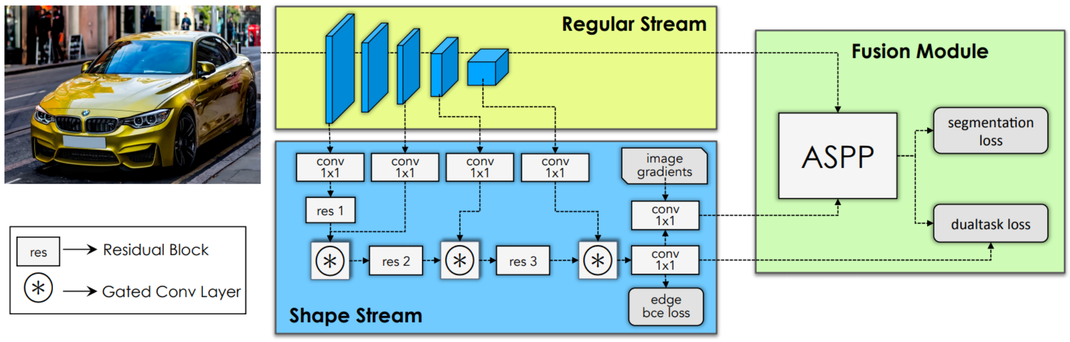


FIGURE 3.7: Example of each Gate Usage

In figure 3.7 a more detailed analysis is being presented the two-stream and fusion modules. There are some so-called gated mechanisms, so the cooperation between the regular stream and the newly added shape-stream branch can be facilitated. These gates keep the shape stream lightweight because they constantly grasp many features by the 'regular' branch.

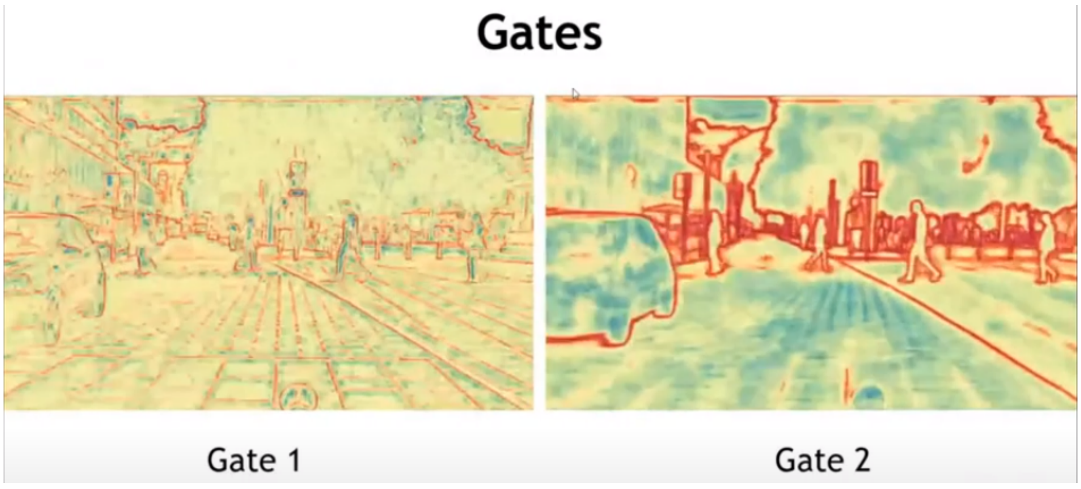


FIGURE 3.8: Example of each Gate Usage

In figure 3.8 we can see more about the functionality of the gates. Gate 1 learned to focus on specific features as the relatively low-level edges and textures, whereas gate 2 highlights the high-level object boundaries. The dual-task loss module comes into play so that the boundary outputs can be efficiently used for a better segmentation guide. The exact processing inside the dual-task module is the enforcement of consistency between these two-stream outputs. Afterward, the resulted segmentation masks can be reviewed and evaluated, which is the two-branches fusion output where dense feature representations(from the 'regular' branch) and boundary guides are combined while preserving the multi-scale contextual information.

3.5 DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs

The following work [46] is focused on Image semantic segmentation using three essential parts. The first one aims to reduce complexity providing a larger field of view without the need for excess filter parameters(atrous/Dilated Convolution [47]). A process supporting variable scales for segmenting objects of interest (atrous spatial pyramid pooling (ASPP)) is also developed. Moreover, the object boundary localization is greatly improved by combining techniques used in Deep Convolutional Neural Networks (DCNNs) and probabilistic graphical models. An object's localization effectiveness can be degraded following a path that includes actions like max-pooling or generally downsampling methods. The DeepLab paper has beaten this kind of hardship by proposing a Conditional Random Field (CRF) [48] that is fully connected.

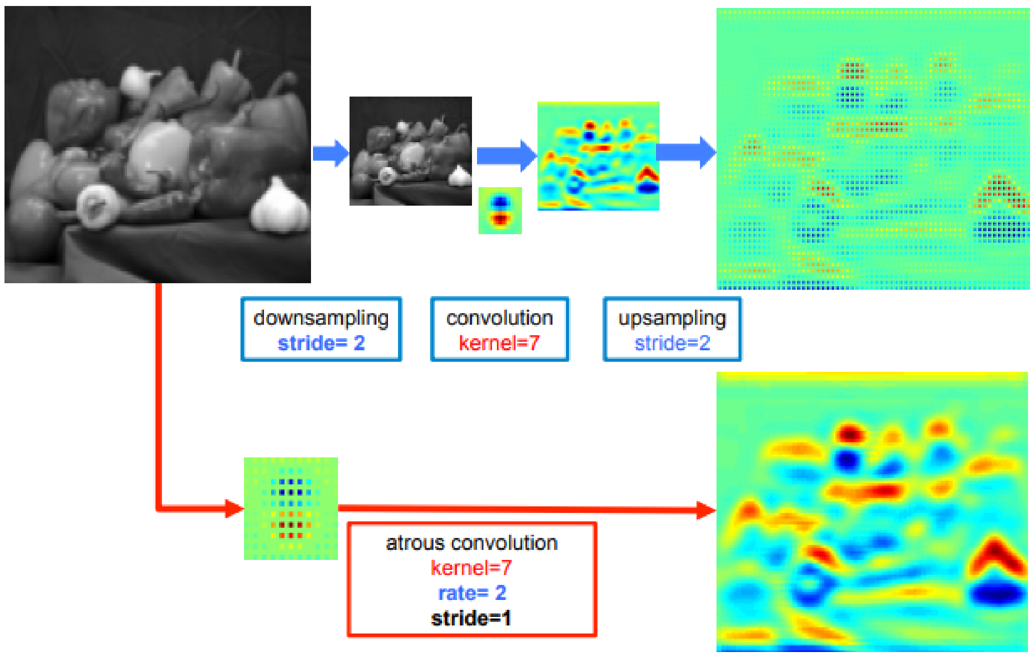


FIGURE 3.9: 2-D illustration of atrous convolution

3.5.1 Dilated/Atrous Convolution

The consecutive max-pooling and striding that occurs almost at every layer reduce the output feature maps' spatial resolution. This reduction has a factor around x32(downsampling). Figure 3.9 illustrates the 2-D including the typical architecture structure of Downsampling → Convolution → Upsampling. More specifically, the classic encoder-decoder architecture's execution means that there is a 'sparse' feature extraction. On the other hand, atrous convolution is an operation that helps to perform a dense feature extraction using a rate of two($r = 2$), which is applied to the initial high-resolution image/feature map. Essentially, a rate of two for a dilated convolution means that the kernel is used with some holes-zeros between the actual learnable values making a larger kernel and not a dense one. This sparse kernel is applied on the input image, resulting in a dense feature extraction without increasing the operation's computational expansiveness. Below, a depiction of a dilated convolution is represented on a 1-D convolution example for both sparse and dense extraction with the following respective settings: $rate = 1$, $pad = 1$, and $pad = 2$, $rate = 2$. Conditional Random Fields are generally used for interactive segmentation or simply for a foreground/background pixel classification.

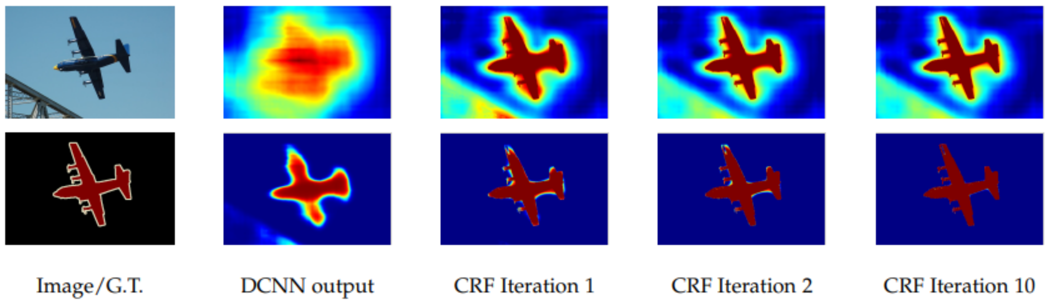


FIGURE 3.10: 1st row(scope) and 2nd row(belief) result maps after each mean field iteration

Figure 3.10 shows that applying this optimization problem, where the neural network iterates the output, establishing that all pixels agree with each other, following the smoothness assumption. At the same time, it keeps correcting(each iteration - CRF correction) to a more accurate and refined solution as the iteration depth advances.

Chapter 4

In-Depth Theoretical Modeling & Robustness Analysis

In this chapter, U-net architecture will be thoroughly analyzed with a very informative description via examples and visual representations of the connectivity and functionality to every bit of the neural network. Furthermore, a mathematical approach will help discover some paths, techniques, and solutions to many possible questions a newcomer has.

Every part of this work is made by combining a vast amount of different approaches, examples, and raw neural network mathematics(especially for back-propagation), creating a result that represents a tutorial that is currently unique since it is not possible for someone, who wants to learn the construction of U-NET, to find a guide that introduces the concept of (U-NET) forward and back-propagation in a lower level of programming language like simple python(with/without using Numpy) or C/C++.

Usually, terms like transposed convolution and skip neurons are only described with images and simple maths, leaving a big part of the actual code and architecture implementation technique in the dark. Keras, PyTorch, mxNET have already built some high-level open-source U-NET tools and libraries, providing a wide variety of architectural freedom with user-friendly functions giving the ability to construct any neural network just by following some simple steps.

Finally, a simple yet fully customizable training function will be shown side by side with the mathematical background analyzing each step with examples and figures. The list of supported settings and options will be explained so anyone can adjust and tune it. The actual purpose of this training function is to test and validate the correctness of the network. Having such a tool -which is very limited in terms of performance since it is built on python using only Numpy- can be very useful for checking the network's characteristics and mainly if the network can learn, improve and generalize the problem.

4.1 U-NET Sub Part & Block Analysis

This thesis U-net architecture can be thought of as two base network parts:

Encoder is the first half of the architecture that consists of many convolutions, activation functions, and max pooling in order to split features into many different categories/dimensions and reduce the pixel count so it can alleviate the computation complexity for the upcoming layers by keeping only the critical information.

Decoder is the second half of the architecture, and its primary goal is to up-sample using transposed convolution to re-construct the image by keeping only the features of interest and positioning each of them at the exact spot from where they have been extracted. Concatenation, followed by the regular convolutions (similar to these used in the encoder part), can result in the image's correct spatial reconstruction. An in-depth walk-through will be presented in the following subsections breaking down the network into multiple parts and blocks.

4.1.1 Analyzing U-NET Sub-Parts

Data Matrix Formats

- *Input/Output*(each layer): *(Channel Number, Height, Width)

- *Filters*: (Filter Number, Input Channel Number, Height, Width)

- *Bias*: (Filter Number, 1) ≈ (Filter Number)

*Assuming there is always a single input image

Encoder

The very first input image[1] can be in any format with the recommended dimension to be a power of two. The last one will significantly help the computation since the up-sampling procedure(later on) does not need any cropping to create

[1] ⚠ *Important note so the pre-trained weights can match the input samples: The format/dimension, pixel range value, and pre-processing of the image files must be the same as the dataset used for training. There is no additional image processing during the reading of each input.*

a result that matches the skipped connection pixel dimension so they eventually can be concatenated with respect to channels dimension.

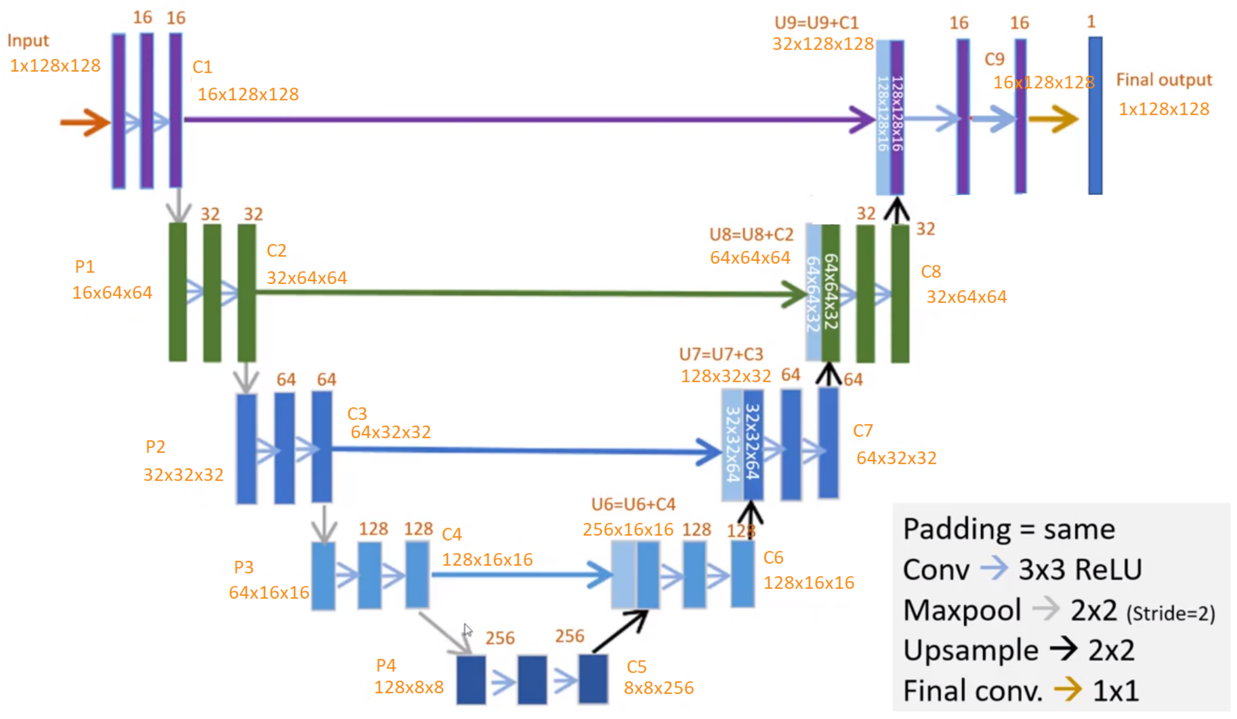


FIGURE 4.1: U-NET Analyzed Architecture - lmb.informatik.uni-freiburg.de

Starting from the 3-dimensional input image, the first step of the process block (Figure 4.2) is a convolution. The shape of the first filter is (16, 1, 3, 3). There are 16 different filters with one channel per filter and a 3x3 kernel size each. It has one channel because this dimension of the filter must concur with the first dimension of the input image, which is also 1 for the simple reason that the images are black and white and not RGB(3 Channels - 1 for each color).

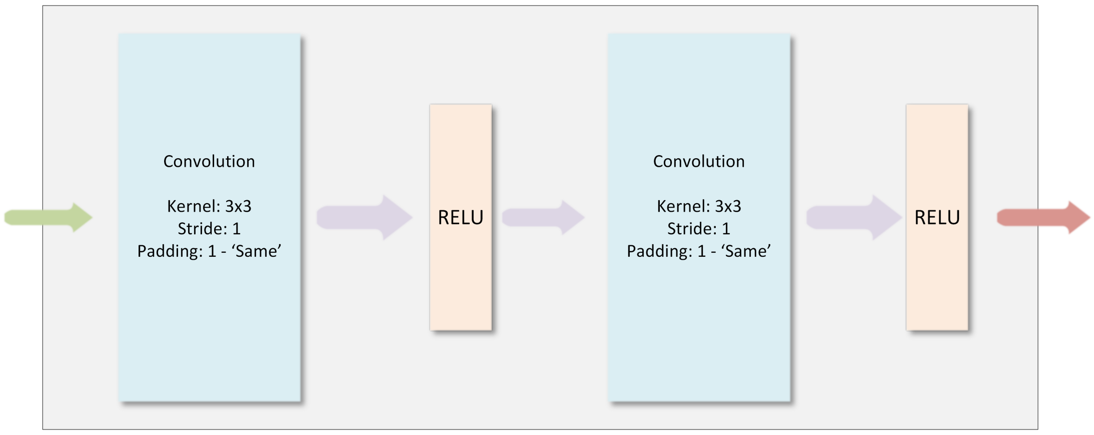


FIGURE 4.2: U-NET Convolution Block

Moreover, the size of the input image is (1, 128, 128). Since the part, as mentioned earlier, is clear, the actual convolution computation can begin. The number of different filters(16) is responsible for the output channel number, which will also be 16. More detailed, each of the 16 filters will be convoluted with the same input and the same one (input) channel to produce the result where the 16 bias values will be added, respectively. The convolution is set to 'SAME' which means that the input resolution will be the same as the output(128x128). Convolution with the 'Same' option is possible by adding a zero padding of 1 around the input image, increasing its resolution to 130x130. Then a simple convolution with the 3x3 filter and a stride of 1 will be computed, resulting in the same as input 128x128 resolution, following the equation:

Output_Size = $\frac{input\_pix - filt\_size + 2*pad}{stride} + 1 = \frac{128-3+2*1}{1} + 1 = 128$ (The same happens for both dimensions H,W).

When the convolution is completed, the output shape will be (16, 128, 128), meaning that 16 extracted features are saved in 16 unique channels, respectively, for each filter that affects the image differently. Finally, the activation function ReLu (which is zero for negative values while the positive go through) is crucial since it can only activate a subset of neurons diminishing the computation complexity.

The same procedure will happen one more time but with different matrix multiplications(using another filter). The intuition behind the subsequent convolutions(that also follow after max pooling) is that each extra convolution can synthesize higher-level features than these of the previous layer.

The concept 'Low-level' features are that the basic geometric shapes are a group of horizontal, vertical, diagonal lines, circles, edges, and corners. The previously extracted features/shapes are combined to produce even more complex features known as 'mid-level'. For example, if a human face is given as input, the mid-level features consist of the nose, mouth, and eyes, so high level includes different faces, as shown below.

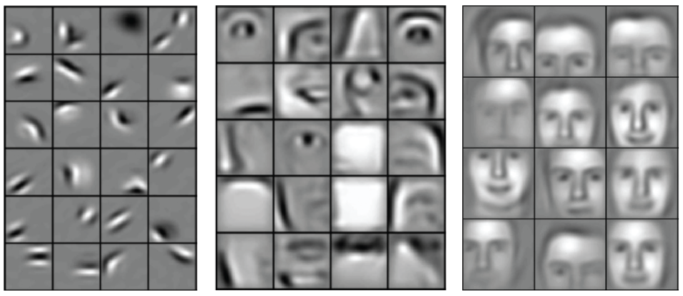


FIGURE 4.3: Convolution Stages/Levels of extraction - medium.com

The complexity is now increased since we have a filter of shape (16, 16, 3, 3), and the new input image is a shape of (16, 128, 128). With more detail, the convolution begins with the first filter, a group of 16 channels convoluted with the corresponding channel of the input, then an addition between them is needed, followed by one more addition of the first filter bias. Finally, having the first channel(out of 16) calculated, the same must be done for the rest 15 filters.
There is a representation of a similar convolution below between an image with shape
(3, 5, 5) and a filter with shape (2, 3, 3, 3)

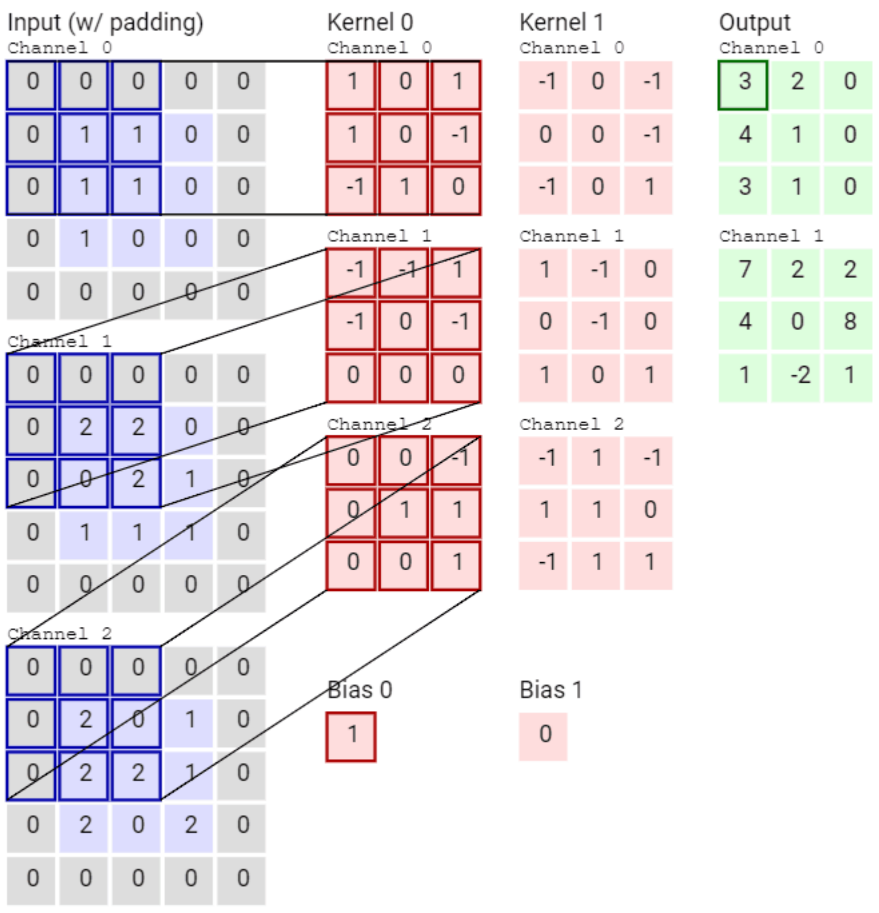


FIGURE 4.4: Convolution Visualization - www.freecodecamp.org

The convolution block concludes with an output shaped as 16 × 128 × 128 like before, followed by a ReLu. This part (like the end of the rest convolutional blocks) is a vital key spot of the architecture because that is the root of where the first skip connection begins(Figure 4.5), so the above result must be temporarily saved so it can be used later on for the concatenation with the up-sampled output(of the same resolution) on the decoder part.

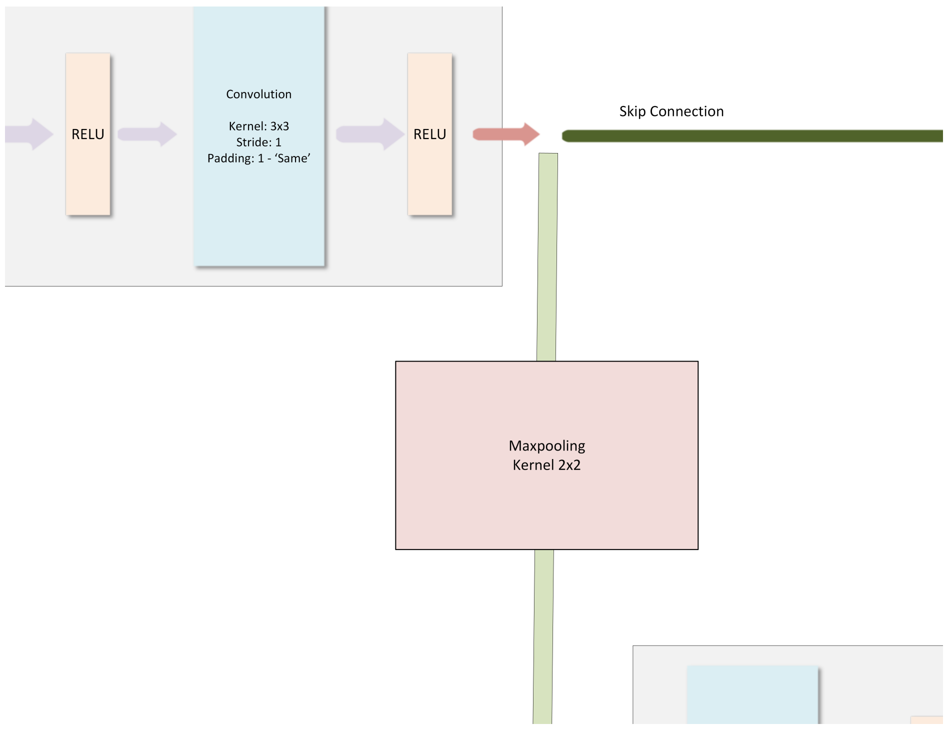
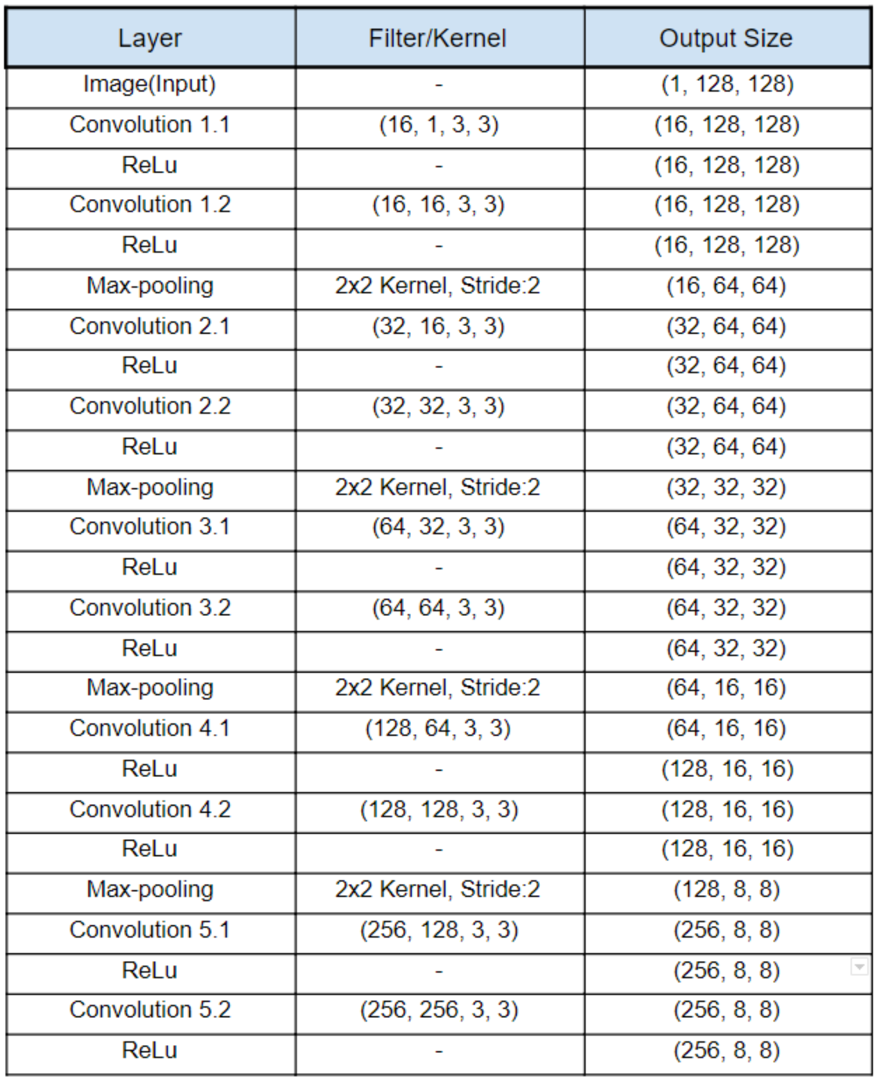


FIGURE 4.5: Convolution block output → Skip neuron & Max-pool

The next major step is to reduce the resolution so the forthcoming layers can receive a lower resolution image; hence the complexity will be less. That job will be executed from the max-pooling module, which has a sub-matrix range of 2×2 and stride of 2, meaning that can convert a 2×2 sub-matrix to just one cell just by choosing the maximum value between them, which practically is the most important one(the one that has the most severe impact). After max pooling is completed, the output image will shape $16 \times 64 \times 64$. The next blocks also will follow the same strategy as those mentioned earlier. The input/output dimension values of the **encoder** are gathered and shown below(Figure 4.6).

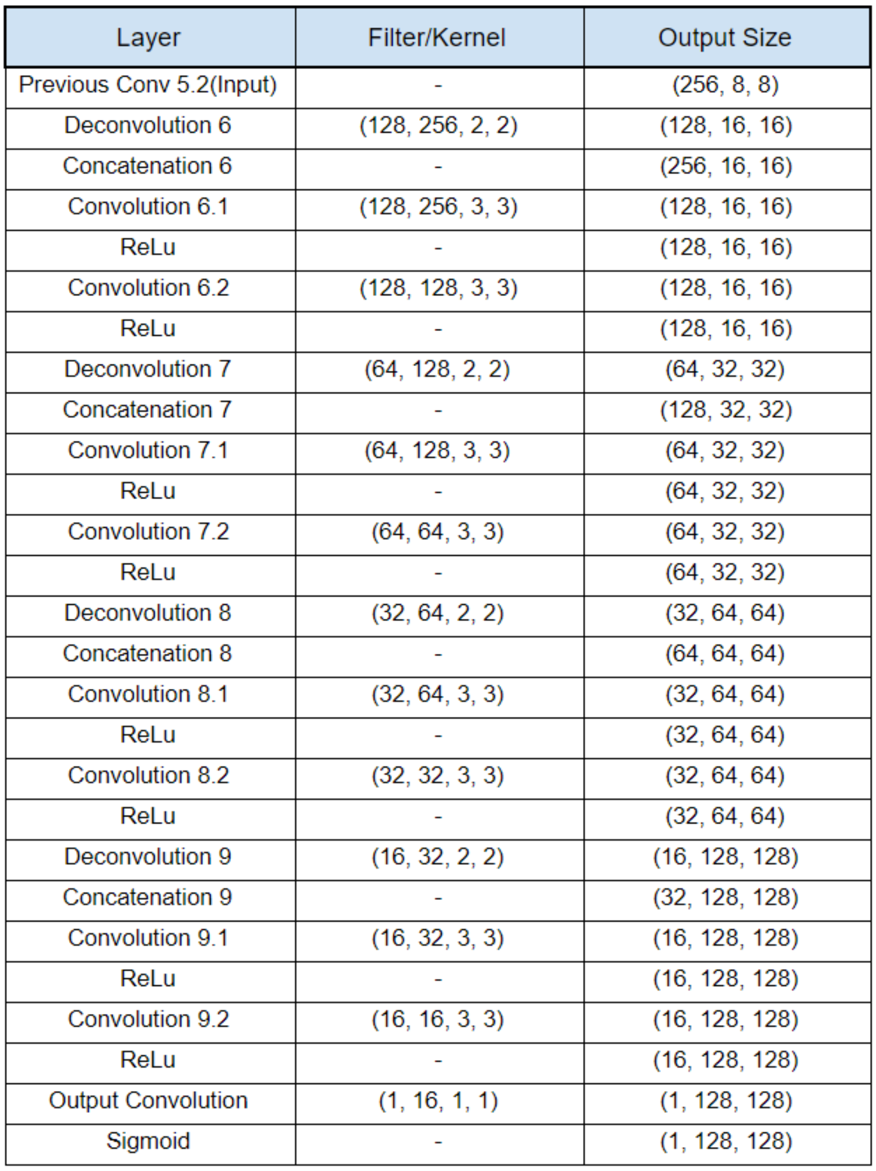


Layer	Filter/Kernel	Output Size
Image(Input)	-	(1, 128, 128)
Convolution 1.1	(16, 1, 3, 3)	(16, 128, 128)
ReLu	-	(16, 128, 128)
Convolution 1.2	(16, 16, 3, 3)	(16, 128, 128)
ReLu	-	(16, 128, 128)
Max-pooling	2x2 Kernel, Stride:2	(16, 64, 64)
Convolution 2.1	(32, 16, 3, 3)	(32, 64, 64)
ReLu	-	(32, 64, 64)
Convolution 2.2	(32, 32, 3, 3)	(32, 64, 64)
ReLu	-	(32, 64, 64)
Max-pooling	2x2 Kernel, Stride:2	(32, 32, 32)
Convolution 3.1	(64, 32, 3, 3)	(64, 32, 32)
ReLu	-	(64, 32, 32)
Convolution 3.2	(64, 64, 3, 3)	(64, 32, 32)
ReLu	-	(64, 32, 32)
Max-pooling	2x2 Kernel, Stride:2	(64, 16, 16)
Convolution 4.1	(128, 64, 3, 3)	(64, 16, 16)
ReLu	-	(128, 16, 16)
Convolution 4.2	(128, 128, 3, 3)	(128, 16, 16)
ReLu	-	(128, 16, 16)
Max-pooling	2x2 Kernel, Stride:2	(128, 8, 8)
Convolution 5.1	(256, 128, 3, 3)	(256, 8, 8)
ReLu	-	(256, 8, 8)
Convolution 5.2	(256, 256, 3, 3)	(256, 8, 8)
ReLu	-	(256, 8, 8)

FIGURE 4.6: Encoder Part

The 5^{th} block of convolutions is the last one of the first half(encoder), where the output result has a shape of (256, 8, 8). As a result, there are 256 features, each saved in a separate channel! Bear in mind that it is also possible for the initial input to be 64×64 resolution since the 5^{th} block will result in a (256,4,4) image, which is also feasible for a 4×4 resolution matrix to save a feature.
On the other side of the coin, choosing a resolution greater than 128×128 for the initial image is not recommended for the current architecture since it would be a waste for the 5^{th} block not to be able to extract the additional information. So, for a greater than 128×128 initial input image, it would be better if there was one more block-reduction to 512 features that one may help to squeeze out some extra features.

Decoder



Layer	Filter/Kernel	Output Size
Previous Conv 5.2(Input)	-	(256, 8, 8)
Deconvolution 6	(128, 256, 2, 2)	(128, 16, 16)
Concatenation 6	-	(256, 16, 16)
Convolution 6.1	(128, 256, 3, 3)	(128, 16, 16)
ReLu	-	(128, 16, 16)
Convolution 6.2	(128, 128, 3, 3)	(128, 16, 16)
ReLu	-	(128, 16, 16)
Deconvolution 7	(64, 128, 2, 2)	(64, 32, 32)
Concatenation 7	-	(128, 32, 32)
Convolution 7.1	(64, 128, 3, 3)	(64, 32, 32)
ReLu	-	(64, 32, 32)
Convolution 7.2	(64, 64, 3, 3)	(64, 32, 32)
ReLu	-	(64, 32, 32)
Deconvolution 8	(32, 64, 2, 2)	(32, 64, 64)
Concatenation 8	-	(64, 64, 64)
Convolution 8.1	(32, 64, 3, 3)	(32, 64, 64)
ReLu	-	(32, 64, 64)
Convolution 8.2	(32, 32, 3, 3)	(32, 64, 64)
ReLu	-	(32, 64, 64)
Deconvolution 9	(16, 32, 2, 2)	(16, 128, 128)
Concatenation 9	-	(32, 128, 128)
Convolution 9.1	(16, 32, 3, 3)	(16, 128, 128)
ReLu	-	(16, 128, 128)
Convolution 9.2	(16, 16, 3, 3)	(16, 128, 128)
ReLu	-	(16, 128, 128)
Output Convolution	(1, 16, 1, 1)	(1, 128, 128)
Sigmoid	-	(1, 128, 128)

FIGURE 4.7: Decoder Part

Things get more tricky since the skip connection technique and transposed convolution are not everyday occupations for a machine learning scientist. As described above, the 'input' features to the decoder part are an image with shape $(256, 8, 8)$. That is precisely the input for the first transposed convolution that aims to double the resolution(from 8×8 to 16×16) and to reduce to half the channel number. Jumping into the actual computation, a (learnable)kernel of shape(128, 256, 2, 2) means that we apply the algorithm of transposed convolution on the input features, so we get a 128 channel output of 16×16 resolution. The stride for the transposed convolution is always set to 2 for both axes.

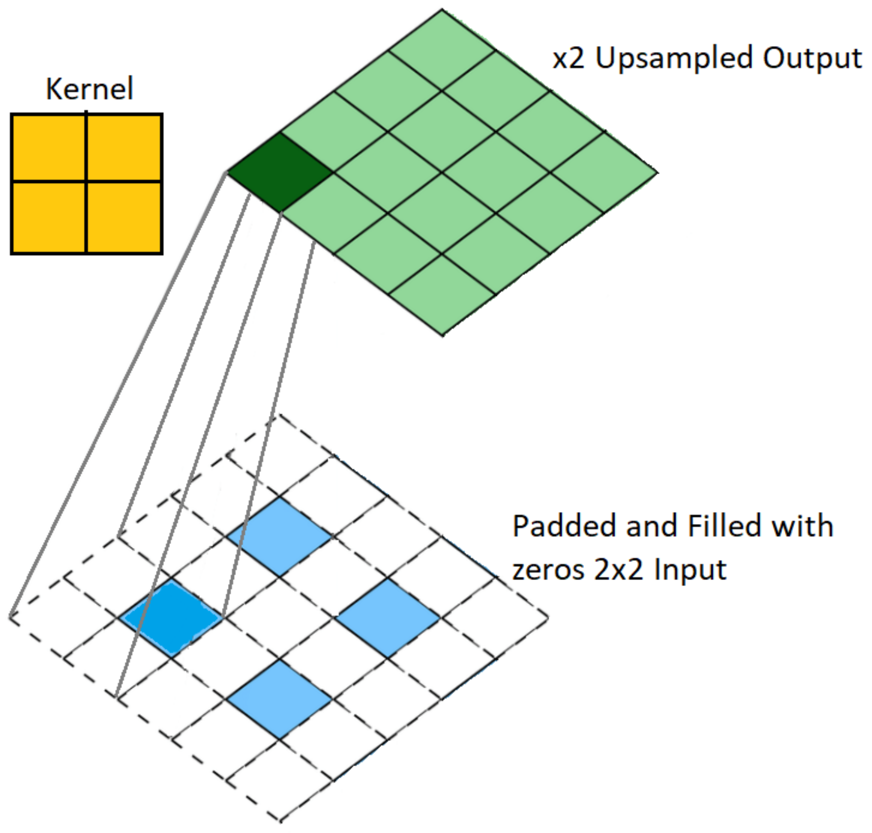


FIGURE 4.8: Transposed Convolution Algorithm 2

According to the transposed algorithms that presented above, the first way(Algorithm 4, Figure 4.8, with the size equations of Figure 2.8) to complete such a computation is to zero pad(wrap around) the input features with 'pad' (where $pad = 1$) plus $(s-1=1)$ zero fill between each input element, and then apply a simple convolution with stride of 1 with the respective kernels. By using the aforementioned algorithm we get:
Padded matrix size: $8*2-1+2*p = 17$
where $8*2-1$ is the number of rows columns after the zero filling between elements and $2*p$ the wrap around zero padding.
Then a simple Convolution is applied, with the flipped 2×2 kernel, $stride = 1$, and using the known equation(Figure 2.8) for convolution we get:
$(17-2)/1+1 = 16$ as a final result which is the double resolution in comparison with the input.

On the other hand, the main Transposed convolution algorithm(Algorithm 3) has more FPGA-friendly properties by utilizing fewer data dependencies. It begins by using each input feature element to multiply it with every kernel element, thus creating a 2×2 partial product stored on the result matrix at the corresponding sub-array slots.

The first pixel(of the first channel) will be multiplied by each kernel element of the respective channel, so this 2×2 matrix product will be saved on the result sub-array $(0 \to 1, 0 \to 1)$. Having a stride of 2 for this algorithm is very helpful since the jump of 2 does not include data dependencies from the previous 2×2 sub-arrays computation, and the new result can be calculated immediately. The output matrix where the addition between different channels happen, there are data dependencies between these per-channel partial results but that cause no problems because the time needed until whole columns are calculated is enough for the data to be computed. The prerequisite calculations for the final result (for the respective filter) include the same procedure for all the input channels.

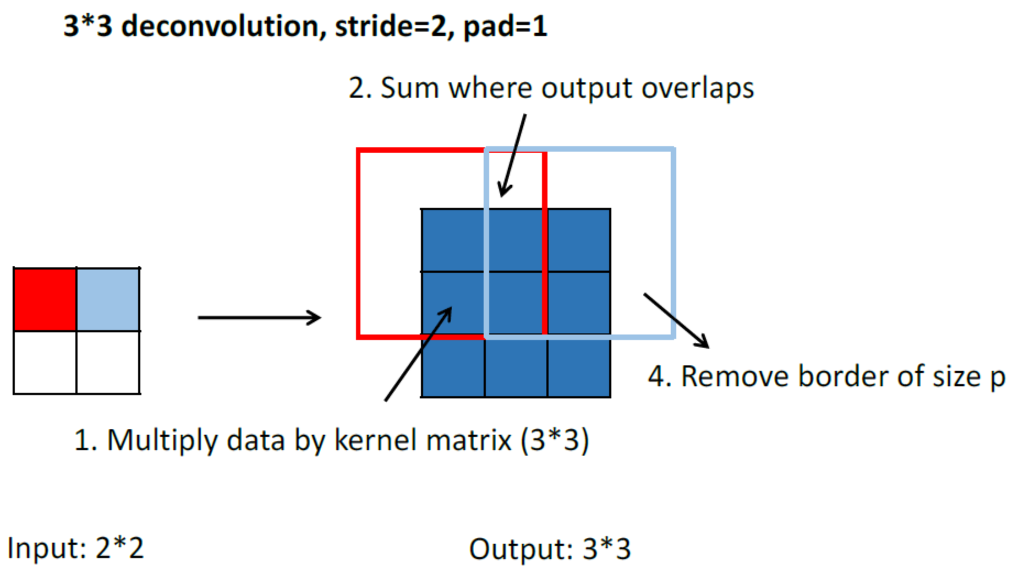

3*3 deconvolution, stride=2, pad=1
2. Sum where output overlaps
4. Remove border of size p
1. Multiply data by kernel matrix (3*3)
Input: 2*2
Output: 3*3

FIGURE 4.9: Example [34] - Recommended Deconvolution Algorithm - Left: Input Image(2×2), Right: Output Image(3×3), Kernel Size = 3, Stride = 2

A simple convolution block, which is also used in the encoder part, will take this fused(Concatenated) input to reduce the channel number back to 128 while keeping the resolution. At the end of this known convolution block(includes two simple convolutions with ReLu, same as the convolution block in the encoder part Figure 4.2), the result (128, 16, 16) will be the input to the next transposed convolution so its resolution will be doubled using the knowledge saved to the 128 channels. So, the output of this transposed convolution will be (64, 32, 32). The same fusing will happen at this point again(as shown in Figure) by sticking the 64 channels of the (encoder) skip connection coming out from the 'Maxpool 3', thus taking back a concatenated result of 128 channels and the same resolution 32×32. The above procedure needs to be completed two more times until the last concatenation is completed between the first skip connection of the encoder part and the last up-sampling result. The last one will return a result (32, 128, 128), the concatenated product. The last convolution block will decrease its channel number down to 16, producing a result (16, 128, 128). Finally, a 1x1 convolution will drive the result channel number down to 1 with the final result be (1, 128, 128), same as the Input Image shape. The final activation function is quite different from the ReLus we used earlier, after every convolution. This last activation function is called sigmoid, which drives values to 0 –>1 space to translate it to pixel values(1 white, 0 black) and compare it with the input. The difference between input and output images, as described before, is that the output includes features/information that our neural network decided to keep and put them back in their place, discarding everything else which is though to be out of the scope of our interest.

4.2 PyTorch, Keras and C/C++ implementations

Firstly, a vital step is the transposed convolution analysis, built entirely from the start without any other type of official or open-source code at any language level. Then, Pytorch tools like training (in-depth) analysis, image pre-processing, and weight encoding-decoding for easy transfer between Keras and C will also be presented. Keras is the top-level programming language where the training capabilities will be utilized, so a malleable parameter file is built and can be used for real-time problems offering comprehensive coverage and generalization. C/C++, including any supported tool built in this environment, constitutes nearly everything needed further to expand the support for an embedded system like FPGA.

4.2.1 Transposed Algorithm

The transposed algorithm 3, (including back-propagation) that developed during this work will be analyzed during training analysis later on. The simple technique was built from the ground up, with the source of mathematics, examples, and results of some already completed higher level transposed convolution functions that essentially reverse-engineered to achieve the same outcome. The source code between many languages and levels is now open source and available for anyone who wants to build a U-NET architecture and implement it on an embedded system. The availability before the release of this work was limited to excessively enhanced and obscure low-level CUDA [49] languages.

4.2.2 Python

The very first attempt of the U-NET neural network took part in a high-level python environment. Having a simple CNN as the baseline, because the lack of information around the U-net architecture was a big obstacle, every area of the architecture explored and validated by using maths, experience, and other methods that can ensure the functionality, thus the ability for the neural network to learn, improve and produce a respectful result. The main idea behind the python implementation is a more in-depth understanding of the matrix multiplications such as weight matrices multiplied with layer nodes, input/output size, how it is produced, and other more intuitive points that need to be entirely clear.

Python Code Walk-through

First of all, reading functions must be constructed in order to read the input images. These functions are built in a way that only 'read' the .PGM (recommended) format without any pre-processing or resizing. Everything is needed(like pre-processing, resizing, reformatting) is already covered by some extra custom made tools that will be presented later on.
By using OpenCV (CV2) [50], the pre-processed gray-scale image data are read for all the categories such as test images, validate images, plus their respective labels/masks. Training images and their corresponding labels can also be loaded in case someone wants to test and verify the neural network capabilities or even test and further improve some of its functions or the training itself.
As mentioned before, training is also implemented only in this high-level environment to validate its functionality.

Training analysis

In this part, the custom training function will be analyzed with a thorough presentation for both forward and backward propagation and the optimizing techniques used to speed up this process. Bear in mind that training running on one core of the CPU is not a recommended training solution since the complexity and the time will need, as the dataset increases, is countless. On the other hand, the training presentation can help the newcomers learn how calculations are executed and allow learning everything is needed to improve further, study, and implement an even more optimized design to their environment. Simple NN, CNN can already provide such an open-source presentation accessible for anyone who wants to learn and build a project. However, when it comes to semantic segmentation, the knowledge of how to implement and get started building a neural network in a lower environment (rather than Keras, Pytorch, and more) lacks supportive material and general intuition.

Train function settings:

- *Number of epochs* : It is the overall number of iterations over the whole dataset.
- *Learning Rate* : It is the actual setting that can control how fast the neural network can learn and improve stability. (Recommended values: 0.008)
- *Batch size* : Every 'batch number' of data (that completed forward and back-propagation), update the filter & bias values according to the accumulated deviations of the last batch.
- *Dropout Enable* : Enable dropout for a more robust and generalized neural network.
- *Group Normalization Enable* : GN is a custom layer normalization that ensures the protection of gradient/value explosions or vanishing over the samples. Epoch execution time increases, but there is a huge improvement of accuracy and stability that eventually decreases the maximum required epochs Filters & Bias trimming: The variance around initialization of the filter values can be set in this setting. For example, trimming = 0.1 means that, on average, values will range from -0.1 to +0.1 with greater possibility around zero(Normal distribution around zero). Fluctuation recommended values : filter_trim = 0.1 , bias_trim[2] = 5*filter_trim.

[2] ⚠ *By increasing beta multiplier, a more foggy(better handling with edge/like using high anti-aliasing values) visual result will be appeared, when using bias_trim ≪ filter_trim will result in a more austere and edgy(pixelated) visualization.*

- *GN Learning rate(Alpha)* : 20*Learning_rate
- *GN Trim* : Recommended : 0.05
- *GN Beta1, Beta2 setting* : Beta1 is the exponential decay rate from which is dependent on the estimates of the first moment when beta2 is responsible for the second-moment estimates. Beta2 is recommended to be approximately one in case of problems with sparse gradients, like the current one that includes computer vision. Default Values: beta1 = 0.92, beta2 = 0.995.
- *Smart Weight disk Saving* : By enabling this utility, when a local maximum accuracy value appears, automatically, the latest weights used to produce this result will be saved in a disk location that is provided by the user. If it is disabled, the weights will be saved temporarily in the local ram 'params' for forwarding/validation steps. Any effort to re-train will erase the local 'params.' There is also a tool that can read the disk saved weights and load them back to 'params' to be used from the validate function.
- *Accuracy Recording* : That option gives the ability to save every step/epoch data progress and eventually print them or produce a Matlab accuracy diagram related to epoch passage.
- *Per epoch/batch accuracy printing* : This is a simple 'verbose' option that gives analytical information after every iteration or epoch on how accuracy behaves on every specific time frame.

The training starts with the initialization of weights and group Normalization values. Adam optimization special values(must be reset to zero every single iteration) help calculating the new weights. More detailed, these values' shape is the same as the bias shape(Number_of_filters, 1).
Moreover, every filter has a specialized 'momentum' value and 'RMSProp' [51] value(both learn-able), so they can be combined to perform the weight update in the best possible way(Saving these values over the iteration is not supported).
Gamma and beta of GN(Group Normalization) [52] are also learn-able and can improve huge deviations between layer values that can muzzle small numbers and lose information(gradient diminishing).

Since the forward propagation is fully explained before, we jump at the end of the forward step, where the actual re-constructed image appears.
**Each final value of the resulting image is made using sigmoid as the output activation function that helped populate our more abstract values to 0 → 1 space so we can compare it with the label/mask.*

Having this product, let us call it $\hat{Y}$ (Figure 4.12), of shape that is same as input and each pixel values ranges from 0→1, we can easily subtract, Y which is the actual label(the ultimate goal/target), from $\hat{Y}$ and take back the difference between their pixel values. This difference includes both positive and negative numbers to spot where we need to increase or decrease the result.
The objective is to back-propagate the difference via the nodes and edges that caused them. This difference must be saved and used later(during weight update) to countermeasure the problem per layer.

For better understanding, the diagram below(Figure 4.10), shows the last transposed convolution followed by the double convolution block with their activation functions plus group normalization and finally the 1x1 convolution followed by the sigmoid. By analyzing this last segment of the neural network, its possible for anyone to understand everything is needed because it includes every different structural stone that used to build this network.

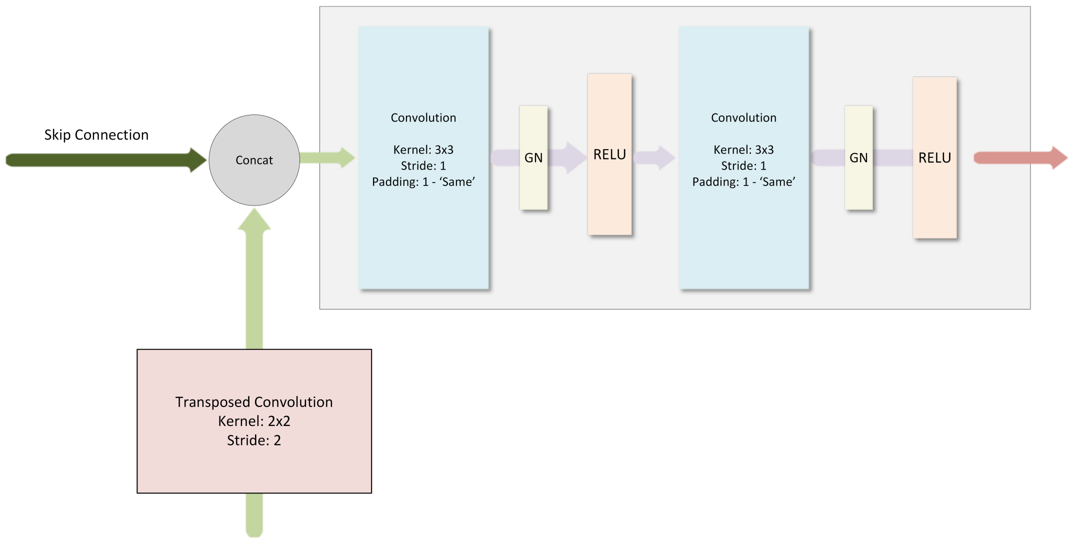


FIGURE 4.10: Training Architecture Part

**The only part that we did not breakdown its forward behavior is the GN, which will be explained through back-propagation.*

Starting from the end, we need a simple subtraction between $\hat{Y}$ and Y. Since we receive the table of differences, let us call it $d\hat{Y}$ (Figure 4.12), we want to locate the path that error went through. The last operation executed is the sigmoid activation function, precisely the first the back-propagation procedure's operation.

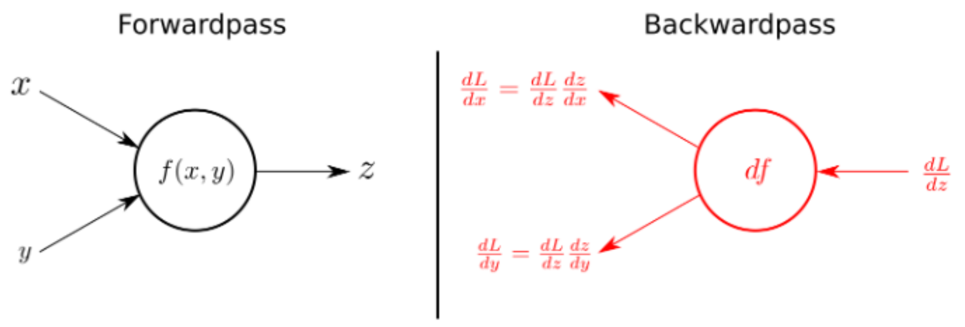

Forwardpass
Backwardpass
x
y
$f(x,y)$
z
$\frac{dL}{dx} = \frac{dL}{dz}\frac{dz}{dx}$
$\frac{dL}{dy} = \frac{dL}{dz}\frac{dz}{dy}$
df
$\frac{dL}{dz}$

FIGURE 4.11: Back-propagation Chain Rule - kratzert.github.io

This is the baseline where all the backward propagation procedure is going to lean on. It's basically a multiplication between local gradient and the gradient that cause the actual result.

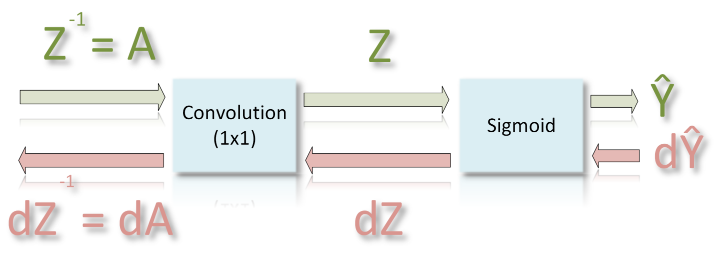


FIGURE 4.12: Forward/Back Propagation Part

We want to know which way is 'downhill' or what is the rate of change of $Y - \hat{Y}$ =Cost= J with respect to filters(**W**eights) : $\frac{\partial J}{\partial w} = (Y - \hat{Y})$, where Y is constant with respect to filters(it won't change, its just a label), so it can be assumed zero $(\frac{\partial J}{\partial w} = 0)$.
On the other hand, $\hat{Y}$ does change as the filters change, so by using the chain rule we get: $\frac{\partial J}{\partial w} = -(Y - \hat{Y}) * \frac{\partial \hat{Y}}{\partial w}$ (1), it is also known that $\hat{Y}$ is the activation function sigmoid applied on the 1×1convolution output. $\hat{Y} = f(z) = sigmoid(z) = \frac{1}{(1+e^{-z}))}$, by applying chain rule again on $\frac{\partial \hat{Y}}{\partial w}$ of (1), the result is :

$$\frac{\partial J}{\partial w} = -(Y - \hat{Y}) * \frac{\partial \hat{Y}}{\partial z} * \frac{\partial z}{\partial w}$$

Now, the rate of change of $\hat{Y}$ can be found with respect to Z(where Z is the 1x1 convolution output), the sigmoid activation function must be differentiated as follows:

$$\hat{Y} = f(z) = sigmoid(z) = 1/(1 + e^{-}z) \rightarrow \frac{\partial \hat{Y}}{\partial z} = f'(z) = \frac{e^{-z}}{(1+e^{-z})^2}(2)$$

$$(1),(2) \rightarrow \frac{\partial J}{\partial w} = -(Y - \hat{Y}) * f'(z) * \frac{\partial z}{\partial w}$$

The term $\frac{\partial z}{\partial w}$ represents the change of z with respect to the filter/weights w. Keeping the eyes on a single cell of the output matrix z is possible to discern what connects z with z^{-1}(which is the state before 1x1 convolution) is just a mix of filters that multiply the corresponding z^{-1} cell :

$$z[x, y] = \sum_{channel=1}^{16}(z^{-1}[channel, x, y]) * (f1[channel]_{1x1}).$$

Looking more precisely, there is a simple linear relationship between filter and z, when z^{-1} is the slope.
Another way to think about how the error will get propagated is that the error $Y - \hat{Y}$ must be multiplied with each filter value, so the 'synapse' that has the most significant values contributes more than the others.

Since we get the result of the function: 'sigmoid_backpropagation($\hat{Y} - Y, z$)' as described above, resulting in the dz, which includes all the differences for the output z of the same shape(1 error/diff per cell).

For the next step, the known data matrices of z, dz and filters need to be passed(in the function's parameter list) that used to produce z from A(where A is the result after ReLu as shown in the figure 4.12):
dA, df = convolutionBackward(dz, z, filters), this function will fill the dA, and df
dA : each entry of dz will try to affect the elements from which were made of.
df: is saved locally, so at the end of this iteration, the differences can be accumulated for this specific layer output and eventually update its weights accordingly.

Convolution Back-Propagation Algorithm

Algorithm 5 is considered that Stride = 1. The output shape and size are the same as the input so that algorithm can fill each cell with its respective error.

Before the algorithm begins, knowing that A is the input from the previous layer, Z the output of the convolution, dZ the matrix corresponding to the error towards the output Z, W the filters/weights, and b the bias, the generic formulas for computing dA, dW and db are given below:

$dA+=\sum_{h=0}^{n_H}\sum_{w=0}^{n_W}(W * dZ_{hw})$

This happens for every channel each time we multiply the whole filter with a pixel and then we add the result back to the output matrix. Some elements overlapped in the case of a 3x3 filter (back-prop convolution) and a stride of 1.

Python : $dA[:,h*s:h*s+filt\_size,w*s:w*s+filt\_size]+=filter[z]*dz[z,h,w]$

$dW+=\sum_{h=0}^{n_H}\sum_{w=0}^{n_W}(A * dZ_{hw})$, where A is a sub part of A.

Python: $dW[z]+=dZ[z,h,w]*A[:,h*s:h*s+filt\_size,w*s:w*s+filt\_size]$

$db=\sum_{h=0}\sum_{w=0}(dZ_{hw})$, db is computed by summing dZ

Python: $db[ch]=np.sum(dZ[ch])$

Algorithm 5 Convolution Back-propagation

```
1: procedure CONV_BACKPROP(dZ, A_in, weights, bias, kernelSize, padding)
2:     $stride \leftarrow 1$                                        ▷ Fixed Stride
3:     for oc:=0 to weights.filters-1 do
4:         for oh:=0 to dZ.height-1 do
5:             for ow:=0 to dZ.width-1 do
6:                 for ch:0 to A.channels-1 do
7:                     for i:=oh to oh+KernelSize-1 do
8:                         for j:=ow to ow+KernelSize-1 do
9:                             $df(oc,ch,i-oh,j-ow) \leftarrow df(oc,ch,i-oh,i-ow)+A(ch,i,j)*dZ(oc,oh,ow)$
10:                            $dA(ch,i,j) \leftarrow dA(ch,i,j)+dZ(oc,oh,ow)*weights(oc,ch,i-oh,j-oh)$
11:        $db(oc) \leftarrow SUM(dZ(oc))$       ▷ Sum of all the HxW values of the current channel
12:    return $dA,df,db$
```

4.2.3 Group Normalization - Forward Step

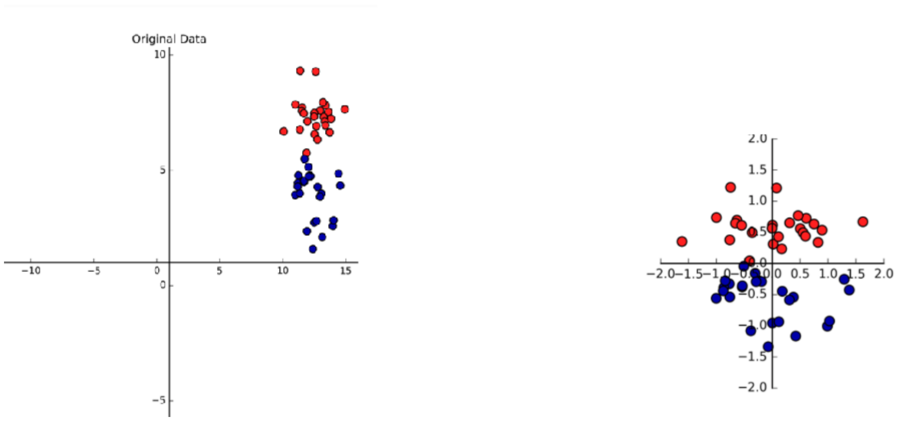


FIGURE 4.13: Scaling & shifting according to the group mean and variance Data Science - Normalization

The idea of this custom group normalization, which can also be called layer normalization, is based on the standard technique for reducing training times while increasing accuracy, Batch Normalization. Batch Normalization takes as a batch, a small group of data samples(which can be multiple inputs per run), but in the U-NET architecture where the general method is one image per neural network run, the batch normalization must be transformed into the group normalization. In the last method, the main idea is to group multiple channels(since there are not multiple input examples) and normalize data with respect to their means and variances, the scale and shift around the zero so the values explosion can be avoided. The $gamma$ and $beta$ are the parameters to be learned.

Input: Values of x over a mini-batch: $\mathbf{B} = \{x_{1...m}\}$
Output: $y_i = GN_{\gamma,\beta}(x_i)$

Mini-batch mean:

$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \tag{4.1}$$

Mini-batch Variance:

$$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \tag{4.2}$$

Normalization :

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{4.3}$$

Scale and shift :

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv GN_{\gamma,\beta}(x_i) \tag{4.4}$$

4.2.4 Group Normalization - Backward Step

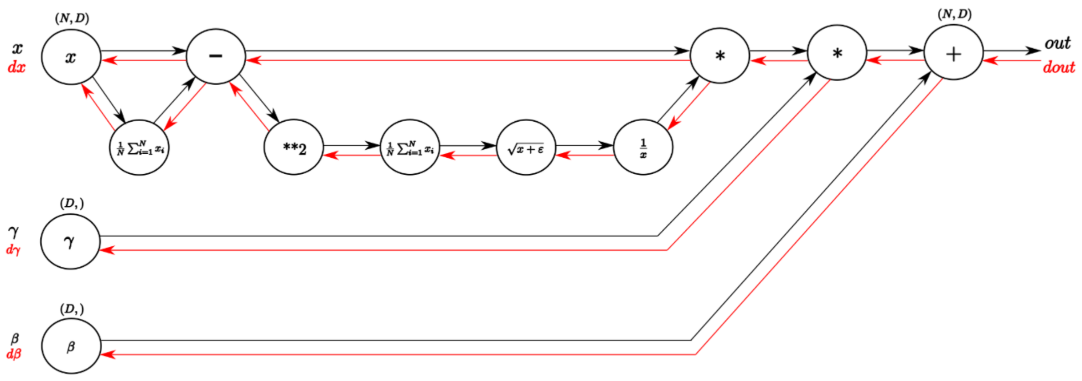


FIGURE 4.14: Forward-Backward paths of GN broken into small differentiable subfunctions kratzert.github.io

Now, the red path must be followed starting from the end $dout$ by using the chain rule:

$dout = d(\gamma\widehat{x}_i + \beta)$, the last summation becomes:

- $d\gamma\widehat{x} = 1 * dout$
- $d\beta = 1 * \sum_{i=1}^{N} dout$

$d\gamma\widehat{x} = 1 * dout$:

- $d\widehat{x} = d\gamma\widehat{x} * \gamma$
- $d\gamma = \sum_{i=1}^{N} d\gamma\widehat{x} * \widehat{x}$

continuing from $d\widehat{x}$:

- $dx\mu_1 = d\hat{x} * ivar$
- $divar = \sum_{i=1}^{N} d\hat{x} * x\mu$

$divar$:

- $dsqrtvar = divar * \frac{-1}{sqrtvar^2}$
- $dvar = \frac{1}{2} * \frac{1}{\sqrt{var+\epsilon}} * fsqrtvar$
- $dsq = \frac{1}{N} * \begin{pmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{pmatrix}^{N \times D} * dvar$
- $dx\mu_2 = 2x\mu * dsq$

From $dx\mu_1 \& dx\mu_2$ *where the subtraction happens, we get*:

- $dx_1 = 1 * (dx\mu_1 + dx\mu_2)$
- $d\mu = -1 * \sum_{i=1}^{N} (dx\mu_1 + dx\mu_2)$

$d\mu$ *back to the mean per group*:

- $dx_2 = \frac{1}{N} * \begin{pmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{pmatrix}^{N \times D} * d\mu$

From Last step is to sum up the gradients dx_1 *and* dx_2:

- $dx = dx_1 + dx_2$

dx contains the errors for the input of the GN, which is ready to back-propagate even further to the previous process(which is convolution)

4.2.5 Transposed Convolution Back-Propagation

Given that the forward pass of the transposed convolution has the algorithm 3, which has been used from the paper [34] it presents a quite FPGA-friendly way of completing a high computational task utilizing many independent additions. In our case, having a kernel of 2×2 with a stride of 2(because we want to double the input's resolution), there are no dependencies between iterations and additions. Figure... shows exactly the current case, having a kernel resolution of 2×2 and a stride of 2.

The upcoming information is critical and abstruse but, at the same time, an effortless way of solving the back-propagation step for transposed convolution. At a first glance it's easy to understand that transposed convolution does exactly the opposite computation than a convolution in combination with a Maxpool (or just a convolution with a stride>1). So, it's known that transposed convolution is actually the inverse way of convolution thus back-propagation of a convolution is the same as transposed convolution. As a result back-propagation of transposed convolutions is the same as the forward simple convolution.

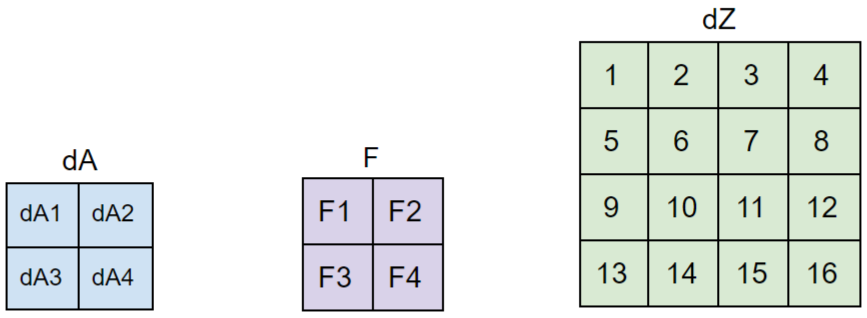


FIGURE 4.15: Example: $dA1 = 1 * F1 + 2 * F2 + 5 * F3 + 6 * F4$

It is related because we want the dz (which is the error stored in a matrix with a shape identical to the output shape of transposed convolution on this part) to be multiplied with the filter so it can propagate the error accordingly to the 'input' of the Deconvolution(same dA shape as the input A).

Algorithm 6 Transposed Convolution Backpropagation

```
1: procedure TCONV_BACKPROP(dZ, A_in, weights, bias, kernelSize, padding)
2:     $s \leftarrow 2$                                    ▷ Fixed Stride
3:     for fil:=0 to weights.filters-1 do
4:         for oh:=0 to A_in.height-1 do
5:             for ow:=0 to A_in.width-1 do
6:                 for ch:=0 to weight.channels-1 do
7:                     for i:=oh*s to oh*s+KernelSize-1 do
8:                         for j:=ow*s to ow*s+KernelSize-1 do
9:                             $df(fil,ch,i-oh,j-ow) \leftarrow df(fil,ch,i-oh,i-ow) + A(ch,oh,ow) * dZ(oc,i,j)$
10:                            $dA(ch,oh,ow) \leftarrow dA(ch,oh,ow) + dZ(fil,i,j) * weights(fil,ch,i-oh*s,j-ow*s)$
11:        $db(fil) \leftarrow SUM(dZ(fil))$      ▷ Sum of all the HxW values of the current channel
12:    return $dA, df, db$
```

Maxpool Back-propagation

Max pool is the last operation to be understood on how to back-propagate, so there is a clear view on every structural part, and eventually, by following the principles above, it is possible to back-propagate the whole neural network and update its filters. There is a more intuitive way of explaining how the back-propagation of Maxpool works. Since the data before Maxpool are available, by doing a fast search again(for the maximum value), with a window 2×2 and stride of 2, it is possible to discover each maximum's exact coordinates element in the 2×2 window. When the above process is completed, having a dz matrix two times smaller (than the one before Maxpool) that contains the errors, it is relatively easy to simply put these errors back from the exact location they came from. The rest of the slots can be filled with zeros, as their values were not the most important.

Algorithm 7 MaxPool Backward

```
1: procedure MAXPOOL_BACKPROP(dz, A_in, kernelSize, stride)
2:     $stride \leftarrow 2$                                ▷ Downsample image to half
3:     $hOut \leftarrow (input.height - kernelSize)/stride + 1$
4:     $wOut \leftarrow (input.width - kernelSize)/stride + 1$
5:     for i:=0 to (A_in.channels-1) do
6:         for j:=0 to (hOut-1) do
7:             for k:=0 to (wOut-1) do
8:                 $(a, b) \leftarrow nanargmax(A\_in, j, k, stride, KernelSize)$   ▷
   It takes a sub-matrix of A_in → $j * s : j * s + KernelSize$ and $i * s : i * s + KernelSize$, finds the max value and returns its coordinates with respect to $KernelSizexKernelSize$ window
9:                 $dA(i, j * stride + a, k * stride + b) \leftarrow dZ(i, j, k)$ ▷ The rest of the
   window values which are less than max become zeros
10:    return $dA$
```

It is essential to remember that the minus sign is vital during the filter update since the filter value must be driven in the opposite direction.
Example:

$$\hat{y} - y = 0.5 - 0.7 = -0.2 = df \rightarrow filter = filter - df * learning\_rate, \quad (4.5)$$

where $learning\_rate = 0.1 \rightarrow filter' = 0.5 + 0.2 * 0.1 = 0.52$, which is improved and provides a lower cost.

Validation Tool

The validation part includes functions like *loading weights* that can be produced either from the implemented design or from Keras training procedure. *Modes and settings* like 'smart forward step' that can predict and drive some of the values to their target point so the accuracy will be increased or even *printing methods* for visualizing the results(output vs. input vs. label and some learning curves in case the network is trained with the implemented solutions). Adam Optimization setting is implemented only for the custom training function(not for Keras), and it is recommended when using the 'Train' function.

Python Tools for image pre-processing and other utilities:

1. *Resizing Tool*: This tool can receive a path where the data are located and then resize them to the desired resolution, which must be in the form 2^n (example: 64 or 128), thus converting from RGB to gray-scale and then saving them back(replacing).
2. Easy Convert .png to .pgm and then move them into a new directory
   Example: mogrify -format pgm /data/salt/images/.png mv /data/salt/images/.pgm /data/salt/images/results
3. Keras → Python/C/FPGA weights byte file: It is used when the training is completed in Keras, and the weights must be transferred into FPGA. This procedure also produces python encoded weights that can be used from the python validation function. *Warning: the encoding is custom and can be handled only by the current implementation of functions.*
4. Python → C/FPGA weights byte file: When the training is being held in a python environment, and it is needed to be transferred into hardware.

Keras

Keras is a high-level implementation tool that gives the user a friendly environment with many useful functions to re-create easy and fast neural networks. The countless libraries written in python can help with any machine learning problem while running on top of some other APIs and accelerators like Tensorflow.

Keras was used in order to:

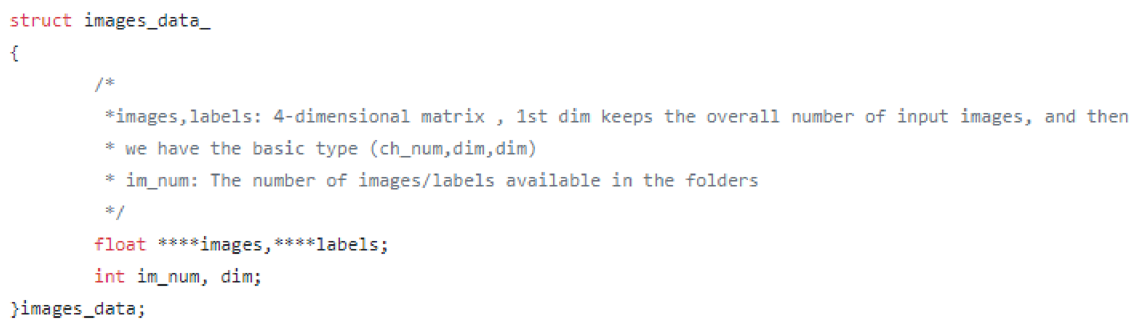
- Validate neural network functionality and the ability of learning by examining each layer output and comparing it with our thesis data.
- Train the U-Net with a faster way by reading the pre-processed image data and building/export weights that are exclusively compatible with the format, resolution, and values of the inputs
- Visualize the results, adding some useful masks on top of the images, and analyzing/converting them into any form and representation.

C/C++

Since the neural network is verified and validated, which prove its functionality, the next major step includes the design transfer from these high-level environments(like Python, Keras) into a lower-level language, like C. The source code written in C to be compatible with FPGA and hardware implementation can easily be converted and used by the Xilinx tools(HLS, Vitis). Python computations and advanced matrix multiplication or processing can be handled from the Numpy library, which eventually must be analyzed and re-written in a lower-level language to reach the core looping system and calculation. Optimization and complexity reduction techniques are essential for designing, loading, storing, and transferring data. Data types are also a significant problem because of the limited memory space and the high throughput needs during hardware sub-part communication.

C/C++ structure analysis & other tools

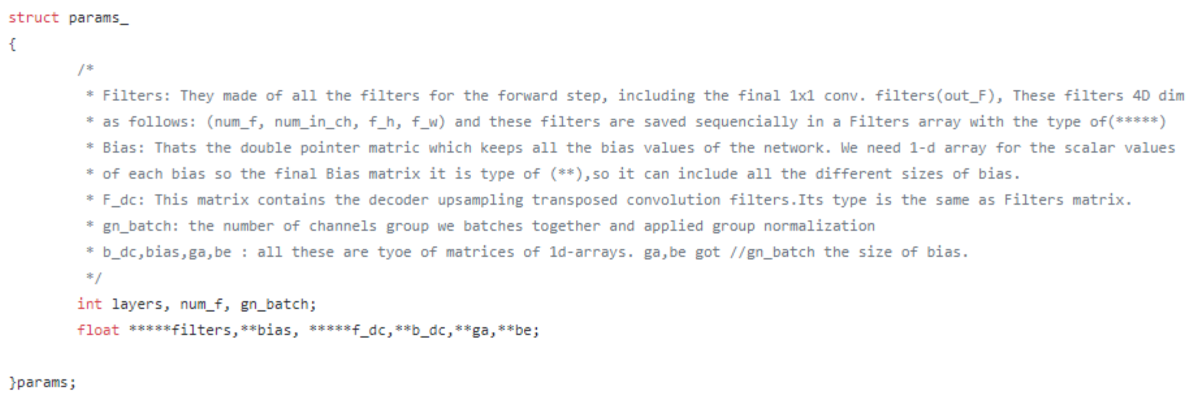
All the structures that will follow are made accordingly from a pool of dynamic space allocation functions. These functions make use of local memory allocation and can be up to 4D dimensions. The actual path of the data is needed so the reading can begin. The file Management segment can handle 8bit or 16bit .PMG pixel depth images. It can also automatically detect the data, formats and match the images with their respective labels. The image/labels data are saved into dynamic arrays, so by having the pointer and the size, we can transfer and read them from any function. Transfers utilize the structures supported by the C environment. These structures include the resolution, pointer to data arrays, and the number of data shown below.



```
struct images_data_
{
        /*
         *images,labels: 4-dimensional matrix , 1st dim keeps the overall number of input images, and then
         * we have the basic type (ch_num,dim,dim)
         * im_num: The number of images/labels available in the folders
         */
        float ****images,****labels;
        int im_num, dim;
}images_data;
```

FIGURE 4.16: Structure example of Image data pack in C

On the other hand, unpacking weights is not that easy since there is a specific encoding from the other higher-level tools and must be followed exactly to read the binary file. Saving the filter/bias data required even more deep pointers since we need a pointer to a 4D matrix so we can pass data by reference.



```
struct params_
{
        /*
         * Filters: They made of all the filters for the forward step, including the final 1x1 conv. filters(out_F), These filters 4D dim
         * as follows: (num_f, num_in_ch, f_h, f_w) and these filters are saved sequencially in a Filters array with the type of(*****)
         * Bias: Thats the double pointer matric which keeps all the bias values of the network. We need 1-d array for the scalar values
         * of each bias so the final Bias matrix it is type of (**),so it can include all the different sizes of bias.
         * F_dc: This matrix contains the decoder upsampling transposed convolution filters.Its type is the same as Filters matrix.
         * gn_batch: the number of channels group we batches together and applied group normalization
         * b_dc,bias,ga,be : all these are tyoe of matrices of 1d-arrays. ga,be got //gn_batch the size of bias.
         */
        int layers, num_f, gn_batch;
        float *****filters,**bias, *****f_dc,**b_dc,**ga,**be;

}params;
```

FIGURE 4.17: Structure example of Parameters data pack in C

With all these data been saved, the reading procedure is completed, and we can proceed into the actual calculations and transfers. Structures are also selected for this purpose, where we can save, filters, data_in, data_out, and other more specific setting values like strides, padding, resolution in/out.

Some functions can assist the computation by returning essential values that have to do with layers, shapes, and sizes, given the layer number. C/C++, Keras, and Python can now work together, share weights, and produce the same results. Each environment has its use, thus making up an essential step to a lower level analysis and implementation. Keras is the training environment since it uses GPU performance to train and deliver the weights efficiently. Python is a more experimental environment where every utility is implemented, from tools and converters to fully configurable training and validation methods. C/C++ is meant to help us approach hardware design, analyze the functions even further, and rebuilding libraries more efficient and targeted to our purpose. The next level includes designing and perfecting the data transfer, parallelizing computations, and configuring communication between an even lower level environment so we can achieve those as mentioned earlier .

Chapter 5

FPGA Implementation

5.1 Tools Used

The hardware implementation and optimization in this thesis work were feasible with the tool package Xilinx Vivado Design Suite - HL System Edition[53] supported and fully licensed from Technical University of Crete Microprocessors & Hardware Lab (MHL). This Xilinx package includes some crucial components for designing, simulating, and debugging any hardware-based creation allowing engineers to validate and precisely monitor an IP(Intellectual Property) before the actual prototype manufacturing. HL System Edition consists of the Vivado IDE, Vivado HLS [54], and Vitis [55]. These tools can work independently, but most importantly, each of these tools works as a complement for each other. The completed idea on which these professional softwares were built is to start with a high-level design using C/C++ using the HLS, which can handle and optimize the final design without a significant effort or knowledge from the creator in low-level hardware design. Since the outputs, extracted in VHDL/Verilog, can be imported to the Vivado IDE, connected with the central processing Unit (PS) of the target board. Vivado IDE, as it will be described thoroughly in this section, provides the block design making the process of connectivity between the central processing unit and programmable logic (PL) an easy task. The last step includes the Vitis, where the output design as a bitstream can be loaded into the Vitis environment where the user can use any available hardware constructed in the previous tools/steps. Controlled by C/C++, PS and PL can work together seamlessly, sharing data with the main target of a specific task's computational acceleration.

5.1.1 Vivado High-Level Synthesis (HLS)

Vivado HL System Edition includes, for free, the HLS edition that enables SystemC, C, and C++ specification to be directly targeted into any supported programmable device by Xilinx, without the need of manually creating a low-level register transfer level(RTL) design. Moreover, abstraction during C/C++ scripting is one of the essential characteristics of this tool that helps the algorithmic description and implementation that it would need countless hours for an Hardware Description Language(HDL) to describe it. Simulation of the C/C++ source code is also supported, extending the current design and adding any additional blocks, protocols, and libraries required to replicate a functional hardware system utilizing on-chip memories, DSP elements, and Flip-Flops. It can also help calculate approximately clock speeds, possible violations, hardware resources used, and provide a visual analysis of each compilation and its timing and data dependencies in hardware representation, assisting users' perception to ameliorate the performance even further. This last visual analysis can significantly help engineers improve the implementation by reducing latency, loop unrolling, and parallelizing the design. Xilinx HLS produces the IP block that can be imported as an official block in the Vivado IDE to be used as a part of a larger-scale implementation by acquiring communication with the FPGA main CPU and having access to other board hardware like DDR memory.

HLS Directives

The user-specified optimization directives are optional, and they can drive the synthesis process to a specific behavior. An implementation can work without these directives, but it is not recommended since they are crucial in assigning hierarchy, roles, and structural optimizations that can significantly impact the performance.

Below, various directive optimizations applied to the desired performance and area goals can be satisfied. A large set of directives can be applied directly in the source code by using pragmas readable by the pre-processor. The other method for using directives is the TCL based method, where its possible to make multiple solutions, then optimize by testing different directives for each one. The management of the different solutions and their corresponding directives can be easily configured within the GUI or even via the TCL based flow. Below, a set of important directives is presented:

- *Allocation:* Reduces the number of operations, cores, and the more general hardware resources with a possible negative impact on latency due to hardware sharing.

- *Array Map:*Fusing smaller arrays into larger with the main goal the resource minimizing of the block ram (BRAM) [20].
- *Array Partition:* Reduces the read/write bottlenecks that appear on large arrays by breaking them down into smaller ones, which can be accessed separately. Below(Figure 5.8), the 3 modes are represented. For block and cyclic, a 'factor' parameter can define the size of chunks that will be created.

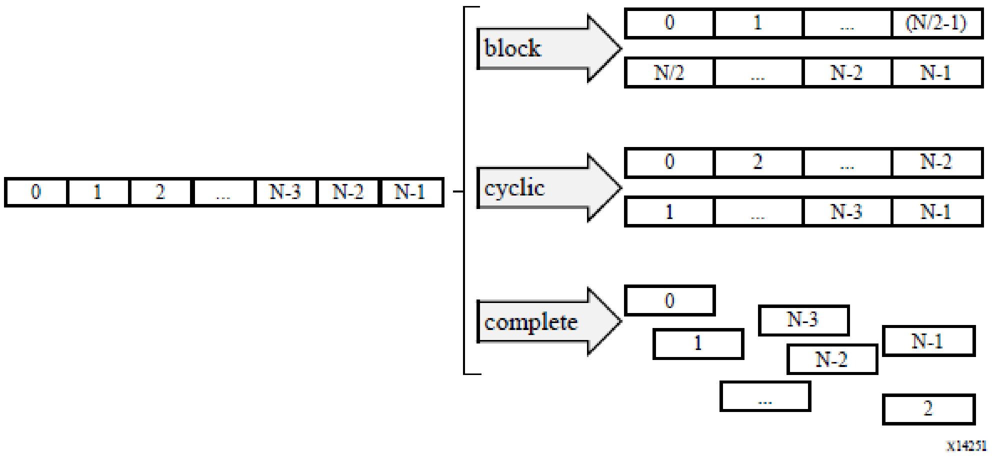


FIGURE 5.1: Array Partitions : UG902-HLS

- *Array Reshape:* Increases block ram access speed without using more block ram, just by reshaping the data from multiple small(width) to fewer and wider words.
- *Data Pack:* Creates a wider width word that packs many data fields of a struct, which is preferred than many smaller transfers(Following the same principle as Reshape).
- *Dataflow:* Allows for parallel execution of tasks, function, and loop with the primary goal, the throughput/latency improvements.
- *Dependence:* HLS tools are very preservative, so they might avoid possible risks by following safer but slower paths. According to some carrying dependencies, these directives let users inform the compiler with more specific information to utilize and succeed better pipelining.
- *Inline:* Function boundaries are removed, so the source code can seem like a united problem and removes hierarchies along with the functions' calling overheads.

- *Interface:* One of the most common directives that specifies the protocols and communication between In/Out ports of the IP with the rest system.
- *Loop Flatten:* Flattens loops in order to succeed better dependency management and apply tree calculations.
- *Loop TripCount:* It is used for a simple latency estimation when loop boundaries/iterations are unknown.
- *Pipeline:* Allows the overlapped execution while a given initiation interval(II) can be set as a parameter that pinpoints the number of cycles needed before new inputs can be accepted.
- *Stream:* During dataflow optimization, the selected array will be implemented as FIFO or RAM. By default, every function's arguments are not implemented as FIFO, and in case of sequential transfers, streaming is preferable.
- *Unroll:* This directive assists the technique of loop unrolling by creating multiple loop instances, so every operation can be placed in the right point for maximizing the computational overlapping, without disrupting the data dependencies.

5.1.2 Vivado IDE

Vivado Integrated Design Environment (IDE) was introduced in April 2012, while it is the base environment where every Xilinx tool was developed. Using high-level programming languages such as Verilog, VHDL, C/C++, and the ability to connect many different parts of a system using an oversimplified block design offers a new approach for designing, compiling, synthesizing, placing and routing and monitoring any FPGA design. In other words, Vivado IDE is the GUI for the Vivado Design Suite. Native Tcl interface is where all the Vivado design Suite tool were written, making possible for the user to access and work with GUI or even directly through Tcl commands via the provided Tcl Console or the Vivado Design Suite Tcl shell. Everything starts by creating the block design, which can also be saved as Tcl files, and it can be re-generated at any time, making easy its transfer from computer to computer. Each design consists of basic modules like the central processing unit (Zynq, MicroBlaze), the clocks, external DDR memory modules, the custom IPs, and some other important interconnects, Direct Memory Access Modules (DMAs) [56], and tools that are required for the successful communication between each different entity. The address manager pane provides the ability to edit the base address, IP, and DDR/BRAM ports that can be assigned automatically, like the I/O connectivity between the modules. Bear in mind that the IP can be further configured and programmed via the IP Integrator's hardware manager to edit the VHDL/Verilog generated files. The last step of the Vivado Design Suite design process is the validation part, where the block design goes through an inspection for possible errors and address violations. With the design validated, synthesis can take place where the RTL design can be converted into a logic gate schematic. Then, place and route can be initiated, which eventually will construct the final 'print' plan that will be downloaded into the FPGA hardware as a bitstream. Moreover, power analysis, design warnings, timings, overall hardware utilization, and temperature analysis are some of the most valuable output logs. Vivado IDE can also run the programmed FPGA with firmware currently running in the ARM processors developed in Vitis. The FPGA runs as a standalone hardware device is also available with the Vivado Design Suite.

5.1.3 Xilinx Vitis IDE

Vivado can use the Vitis Integrated Design Environment (Vitis IDE) for creating applications based on the Zynq - 7000 series SoCs, Zynq UltraScale+ MPSoC [57], and the MicroBlaze [58]. Vitis is an eclipse extension that supports C/C++ code editors, libraries, compilers, and debugging tools. Directly after the bitstream production in the Vivado Design Suite and since the exported hardware is generated based on that bitstream, Vitis can load the .xsa files with all the required drivers. In the C/C++ environment, both PS and PL can be configured, programmed, and scheduled accordingly to the user's purpose. The first thing that needs to occur is the initialization of the PL modules like DMAs, IPs, and everything else different from the ARM CPU(for example, timers, SD Mounting). When the C/C++ programming procedure is completed, the FPGA must be connected with a PC via the JTAG [59] port to program the hardware while the data sending/receiving can be monitored and debugged via the UART [60] port. Every time a new change happens on the software side(how and what the ARM central processing unit runs), then the ability to compile and program only the PS part is possible. The final solution can reduce programming time since the PL part is not reset and does not need to be reprogrammed from scratch. Another helpful feature is that Vitis can program the hardware using remote access to a server PC that runs the Xilinx Tools(with the respective Xilinx version).

5.2 FPGA Platform

The targeted platform of this Thesis is the ZCU102. The size of the IPs in terms of hardware resources is moderate, which means the usability of these modules can be extended even further for a wider Xilinx family accelerator. This section will present the primary hardware specifications of the targeted FPGA.

5.2.1 Xilinx Zynq UltraScale+ MPSoC ZCU102

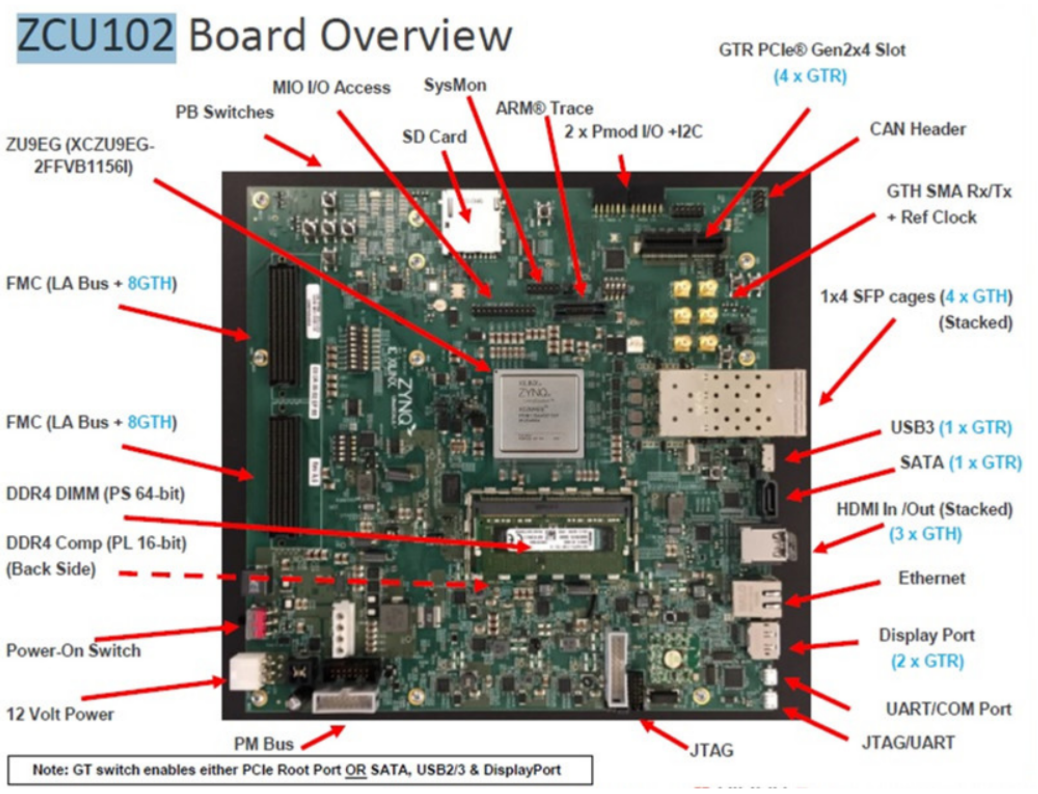


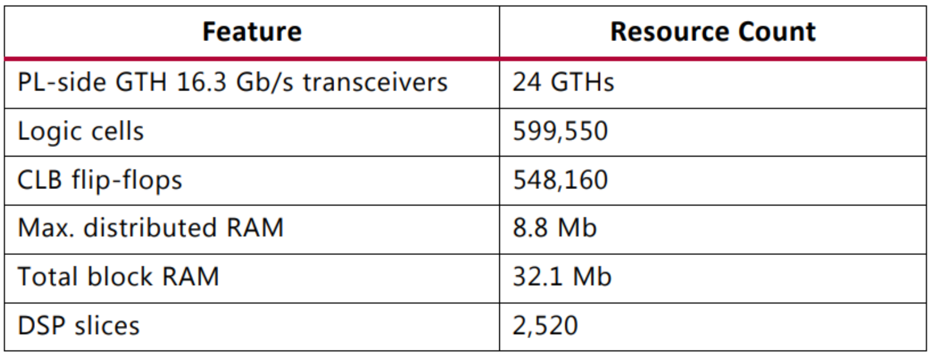
FIGURE 5.2: Xilinx ZCU102 Evaluation Board overview: www.xilinx.com - ZCU102

The Xilinx ZCU102, with the code name Zynq UltraScale+ XCZU9EG-2FFVB1156E MPSoC, is an evaluation board with the following specifications:

- Quad-core 64bit ARM v8-A Cortex-A53 with L1/L2 cache.
- Dual-core 32bit ARM v7-R Cortex-R5 with L1 cache.
- ARM Mali-400 MP2 graphic processing unit with 64 KB L2 cache.
- 256KB on-chip ECC memory.
- 4GB 64bit ECC DDR4 SODIMM RAM, 260-pin(For PS)

- 512MB DDR4(For PL)
- PCIe Gen 2x4 slots
- Supports: USB3.0, UART, JTAG, Display Port, HDMI(IN/OUT), Ethernet and SATA

The specs of the Xilinx 16nm FinFet+ programmable logic fabric are shown below:



Feature	Resource Count
PL-side GTH 16.3 Gb/s transceivers	24 GTHs
Logic cells	599,550
CLB flip-flops	548,160
Max. distributed RAM	8.8 Mb
Total block RAM	32.1 Mb
DSP slices	2,520

FIGURE 5.3: PL Fabric resources - ZCU102 User Guide

The top-level block diagram is presented below(Figure 5.4):

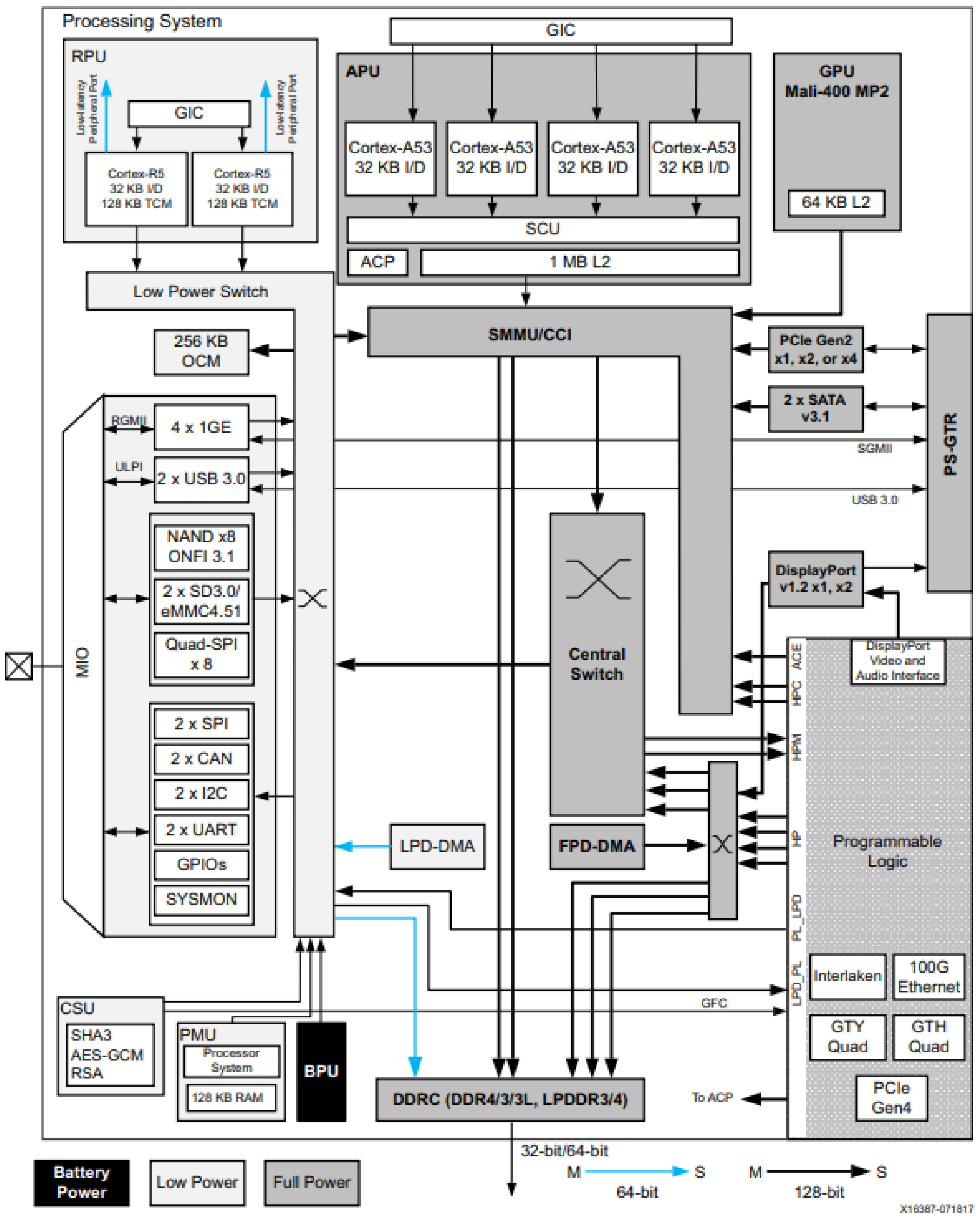


Figure 3-1: **Zynq UltraScale+ MPSoC Top-Level Block Diagram**

FIGURE 5.4: ZCU102 Top-Level Block Diagram - ZCU102 User Guide

There are also many ways of data managing and linking CPU, DDR, BRAM, and PL. AXI-4 full/Lite protocol [61] is used for streaming data(single data or burst of data) between different devices. The capability of transferring large chunks of data is possible by this protocol without repeating instructions and address mapping but just the start address and size of the burst. Clock deviation between different modules can be handled using large FIFO for buffering the data, which are restricted due to the low throughput a module possible has. A direct memory address module can serve such a functionality; thus, the central ARM processor does not need to 'feed' the specific module with data to focus on certain and more essential tasks. Axi-Lite is a lightweight interface built for small data transfers. Generally is used for task scheduling instruction from ARM to DMAs or even small setup information needed for an IP to begin computations.

On the other hand, being aware of the low throughput and latency of the main memory, an ingenious strategy is to use the BRAM that can be configured in multiple data structures, sizes, and setups, optimizing throughput and latency. BRAM's unfavorable characteristic limits the data can be stored because this kind of memory represents a cache like structure with cache like speeds. So BRAM can be a few MBs in size, providing tremendous bandwidth. BRAM is located in the PL part, and that is why the shape and size can be adjusted to be task-specific by having multiple ports, banks, or even construct multiple different BRAMs that can even split into multiple arrays as described above using the corresponding directives.

5.3 FPGA implemented Design Overview

In Figure 5.5, the overall FPGA design is presented, showing 4 Key parts. The first and most important is the central processing unit that is the module from which every instruction, initialization, and data transfer will be managed. The accelerators make up the second group of modules focused on boosting computation performance in task-specific sectors of the neural network, which cannot be handled by CPU. Such tasks are the classic convolution, the Max-pooling, and finally, the Transposed Convolution(or Deconvolution). The 3^{rd} group consists of the DMAs used for data transferring to/from IPs. Each IP channel stream got its own DMA so the data transfers can be overlapped. The last group in this design is the Axi interconnects, and more specific, the AXI interconnect(AXI-Lite) for programming all the DMAs and IPs, but also the AXIsmartConnect built as an IP output(AXI-Stream) driver to the High-performance slave of the Zynq that is connected to the DDR module.

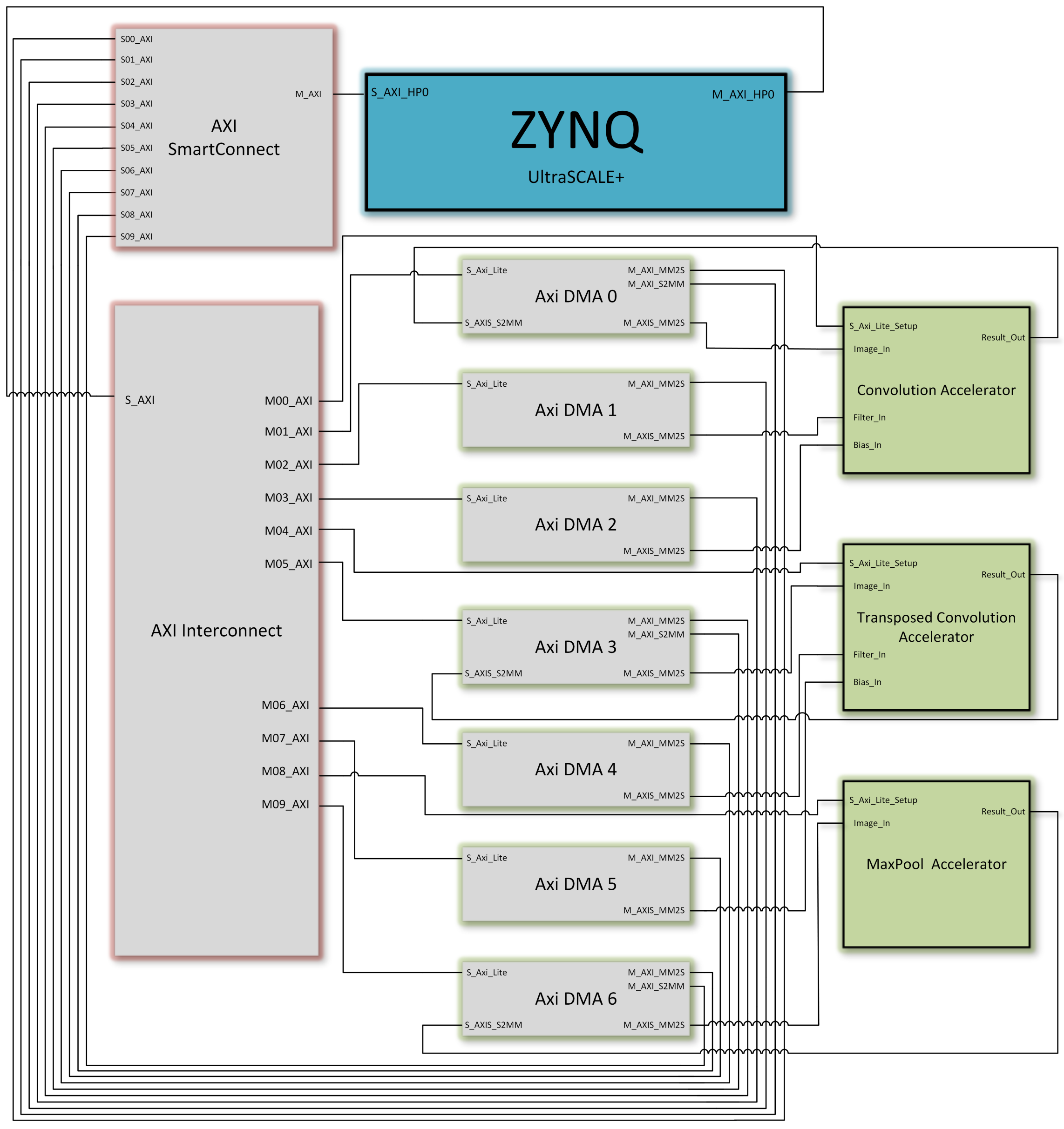


FIGURE 5.5: Main Neural Network Block Design

5.3.1 Platform Accelerator Architectures

As presented above, each accelerator works independently having its own dma channels and integrating techniques like pre-fetching, input line buffering, pipelining tasks and advanced BRAM data strategies in order to maximize throughput avoiding read/write bottlenecks.

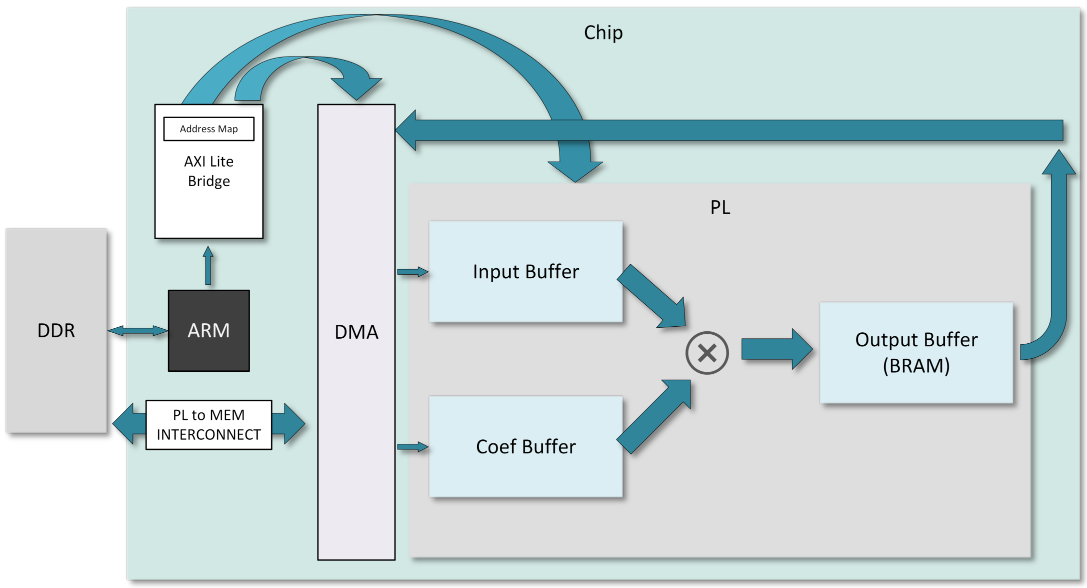


FIGURE 5.6: Accelerator-System connectivity schematic

U-NET architecture was able to be implemented and accelerated in FPGA using three custom IPs[1] that accelerate certain computations giving a massive advantage in performance compared with any other type of simple CPU running this neural network in a more user-friendly, high-level programming language such as python. A noticeable boost in performance also appears when even a lower-level programming language such as C/C++ is compared with the FPGA's low power and efficient solution.

Convolution Accelerator

Computing the convolution is by far the most complex operation in the neural network as its slowest routine takes up to 390 ms, handling a convolution between an image of size:(256, 16, 16) and a filter with the size of (128, 256, 3, 3). From Figure 5.6, the data flow starts from DDR directed by the specific DMA to the accelerator's input streams. When these data reach the custom module, a series of activities start happening.

[1] ⚠ *The maximum Input Image Resolution supported is* 256×256

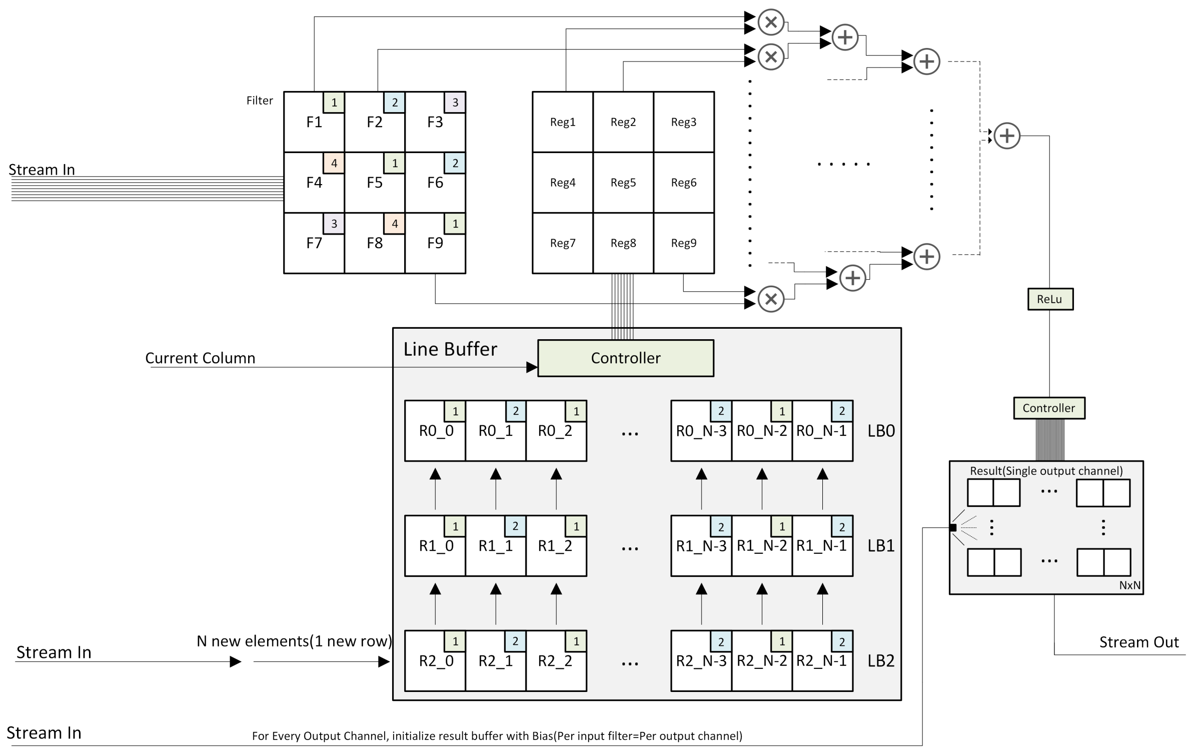


FIGURE 5.7: Architecture of the Hardware used for Convolution

Figure 5.7 demonstrates the architecture of the classic convolution by breaking down any necessary modules that are used to construct this accelerator.
First of all, this design's data structures differ because each data type(image, filters, bias) needs a special treatment according to the frequency, repeatability of data, read/write patterns, and size. The BRAM utilization of this module it's about 7% of the overall Block RAM available to the ZCU102. Three separate buffers were used, so three rows of the image can be stored. The input controller loads N elements each time a new output row production is initiated, pushing these N elements into the line buffer number 2, which always holds the newest input image row. Every time that push happens to the LB2, two shift-up actions have receded, making sure the old data of (Line Buffer 2)$LB2$ are shifted to the $LB1$ and respectively, the $LB1$'s data are shifted up to the $LB0$.

There are 3 Line buffers because the kernel size is 3x3, so the intuition is that having three lines of N elements each, it is possible to process a significant amount of calculations (for each input channel) for these three lines, producing a part of the output channel which has a shape of N elements. Each line buffer is also implemented as block Ram, which only has a maximum number of 2 data ports. On top of that, each line buffer is divided into two smaller BRAM modules(the colors/numbers at the top right side of each cell can show on which module is distributed 5.7). With banking, which is the partitioning in 'cyclic' setup, each memory bank stores only a fraction of the total data. For example, each line buffer can support two different actions: writing to an odd cell position(address) and writing to an even cell position. To sum up, three different buffers, each with a native 2 data ports, are further split into two smaller 'arrays' doubling the number of ports.

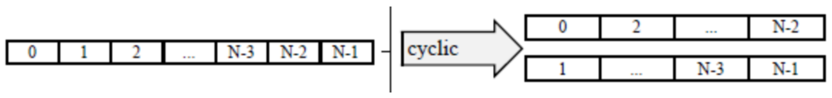


FIGURE 5.8: Cyclic Partition : UG902-HLS

Of course, the last actions result even more in the BRAM utilization burden. After the shift-up ($LB2 \rightarrow LB1, LB1 \rightarrow LB0$), which happens every time the output row changes, the DMA controller will fill the LN2 with a quick burst of N elements. Starting with the bias data, each of the bias value covers one complete filter. Each filter is going to create a different output channel. With that in mind, for each filter, the first action needed is to initiate the output buffer 'Result' with the bias values for each cell. Thus, bias is already stored in the output by the initialization, which is more efficient than filling with zeros. For the bias, no local buffer is used, just a register for the transfer to the output result, which means that it can be loaded directly from the stream provided by the DMA. Simultaneously, a different DMA channel is programmed by the central processing unit to transfer each input line of the image in a local private BRAM storage handled by the IP. The IP accepts and saves the image line as described before. The same happens for the filter input stream, where each time the input channel changes, a new 3x3 filter is loaded via stream. The local filter buffer is split into four different 'arrays,' so the maximum throughput is achieved. Multiplications between the sub-matrix(window) where the filter is sliding on and the actual filter are organized to a tree -structure logic in order to utilize to the maximum the pipelining starting every single addition and multiplication the earliest possible. Having such large size floating-point variables, ZCU102 is capable of producing output after four cycles of adding two float variables and two cycles for the multiplication. This latency of 4 cycles for the addition between floating-point variables can be improved if the variable type has been reduced to 16 bits or less. The ReLu module, which is embedded in the convolution core, does not add any additional delays.

Below, a pseudo-code, thoroughly analyzed, represents all the backbones of the altered/optimized algorithm and some crucial parts that improve the accelerator's performance, such as pre-fetching, pre-compile instruction swapping(which works better than allowing the software to do this job), and loop unrolling.

```
////////////////////////// PSEUDO-CODE //////////////////////////
    //Directives
1:  #pragma HLS ARRAY_PARTITION variable=image_row_0  cyclic factor=2 dim=1
2:  #pragma HLS ARRAY_PARTITION variable=image_row_1  cyclic factor=2 dim=1
3:  #pragma HLS ARRAY_PARTITION variable=image_row_2  cyclic factor=2 dim=1
4:  #pragma HLS ARRAY_PARTITION variable=filt cyclic factor=4 dim=1

    //Streaming Interfaces
5:  #pragma HLS INTERFACE axis register both port=image
6:  #pragma HLS INTERFACE axis register both port=filter
7:  #pragma HLS INTERFACE axis register both port=bias
8:  #pragma HLS INTERFACE axis register both port=result


     /*Starting by zero padding the ZB0, then fill the 1st and (N-1)th (last)
     cell of the ZB1,2 with zero(between them the first data of the current
     channel will be put).*/
9:   for(int y = 0 ; y < (img_width) ; y++)
10:     img_t0[y] = 0; /*Line buffer 0, all zeros every time the input
                         channel is changed*/
11:  img_t1[0] = 0;        //  1st element of LB1
12:  img_t1[dim-1] = 0;    //  Last element of LB1
13:  img_t2[0] = 0;        //  1st element of LB2
14:  img_t2[dim-1] = 0;    //  Last element of LB2

     //Filter loop begins
15:  for (int i = 0 ; i < filter_num ; i++)
16:  {
17:     float bias_t = bias.read();  /*Read the respective bias for the
                                       current filter which*/
    //Next up, output buffer will be initialized with the bias value.
    /*
    By doing this step here, it will be finished when
    the main computation start
    */
19:    for(int x = 0 ; x < output_dim ; x++)
20:    {
```

```
21:        for(int y = 0 ; y < output_dim ; y++)
22:            res[x][y] = bias_t;
23:    }

    //Channel Loop begins
24:    for(int j=0; j < channels ; j++)
25:    {
            /*
            Every time a new channel starts, a very first data read
            intended for the LB1 must be executed. That means, when
            a new channel begins, we need to put the 1st row of the input
            image into the LB1(wrapped by padding) when the LB0 is zero.
            The LB2 will be handled later.
            */
26:        for(int z = 1 ; z < (dim-1) ; z++)
27:            img_t1[z] = image.read();/*Assume, the stream sends
                                    the data in the correct order*/
        //Filter loading
28:        for(int z = 0 ; z < KernelSize ; z++)
29:            filt[z] = filter.read();

            //Output row loop begin
30:        for(int x=0; x<(output_dim-1); x++)
            // The 1 less iteration will be explained later
31:        {
              //Reading the new data line from the input buffer
              //directly to the LB2
32:            for(int z = 1 ; z < (dim-1); z++)
33:                img_t2[z] = image.read();

                /*
                pre-calculation of some important offsets, so the extra
                computations for the addressing can be avoided during
                the complex part.
                */
34:            int offset1,offset2;
35:            offset1 = 1;
36:            offset2 = 2;

                 //Output column loop begins
37:            for(int y = 0 ; y < output_dim ; y++)
38:            {
                    //Loop filter has been fully unrolled
                    /*
                    Below, the element-wise multiplication takes place
                    between the line buffers current window and filter,
                    followed by summing the results each other in a
                    tree-like scheduling technique can take advantage
                    of the pipeline directive. Keep in mind that:
                    32bit float summation Latency: 4 cycles
                    32bit float multiplication Latency: 2 cycles
                    */

                   float reg0 = img_t0[y]*filt[0];
                   float reg1 = img_t0[offset1]*filt[1];
                   float reg01 = reg0+reg1;
                   float reg2 = img_t0[offset2]*filt[2];
                   float reg3 = img_t1[y]*filt[3];
                   float reg02= reg2+reg3;
                   float reg4 = img_t1[offset1]*filt[4];
                   float reg5 = img_t1[offset2]*filt[5];
                   reg01 = reg01 + reg02;
                   float reg03= reg4+reg5;
                   float reg6 = img_t2[y]*filt[6];
                   float reg7 = img_t2[offset1]*filt[7];
                   float reg04= reg6+reg7;
                    //offset pre-calculation for the next iteration
                   offset1 = y+2;
                   float reg8 = img_t2[offset2]*filt[8];
                    //offset pre-calculation for the next iteration
                   offset2 = y+ 3;
                   reg01 = reg01+reg03;
                   res[x][y] += reg04+reg8 + reg01;
               }
                /*
                Shifting up one time, so the next iteration can
                update/refresh the LB2, which shifted up its
                data(like LB1 → LB0)
                */
               for(int y = 1 ; y < (dim-1) ; y+=2) //unrolled 1 time
               {
               img_t0[y]   = img_t1[y];
               img_t1[y]   = img_t2[y];
               img_t0[y+1] = img_t1[y+1];
               img_t1[y+1] = img_t2[y+1];
               }
           }//End of the output row loop
            /*
            Following the same way, the last iteration for the output
            row will be completed. The reason of this split is that
            an if statement is avoided. (Because the last row of the line
            buffer, when the last row of input is reached,
            needs to be filled with zeros)
            */
//////////////////////////////////////////////////////////////////
//Same procedure as above but for the last output row is omitted//
//////////////////////////////////////////////////////////////////
            /*
            pre-loading LB0 with zeros, because the next
            iteration(Loop of input channels) needs to fresh
            start with that zero LB0 and the first input
            row to the LB1.
            */
           for(int y = 1 ; y< (dim-1) ; y++)
               img_t0[y] = 0;   //presetting LB0 to all zeros
       }
        /*End of input channels loop that means an input filter
        with all of its channels has completed the convolutions
        with the respective image channels.
        */
        //Writing to stream the completed output (single)channel
       for(int x=0; x<output_dim; x++)
           for(int y=0; y<output_dim; y++)
               result.write(res[x][y]);
     }// End of filter loop
////////////////////// End of pseudo-code  //////////////////////
```

***Note:** For the most of the loops the directive' #pragma HLS pipeline' was used

Transposed Convolution Accelerator

The transposed convolution accelerator uses the main algorithm idea from the paper [34], when many parts have been altered and adapted to this U-NET architecture with some standard required specifications like the power of 2 resolution of the input/output image, a stride of 1 for the convolutions followed by the stride of 2 max-pooling(reducing size to half) and the standard ×2 up-sampling during the deconvolution(transposed convolution). This work approach utilizes an average to a low amount of BRAM resources, just 26% for ZCU102. That means the transposed convolution IP is relatively lightweight, thus compatible with a wide range of FPGAs. The fixed stride of 2 and the kernel size of two is a fixable constraint and does not affect the main algorithm and speedup compared to a standard processing unit.

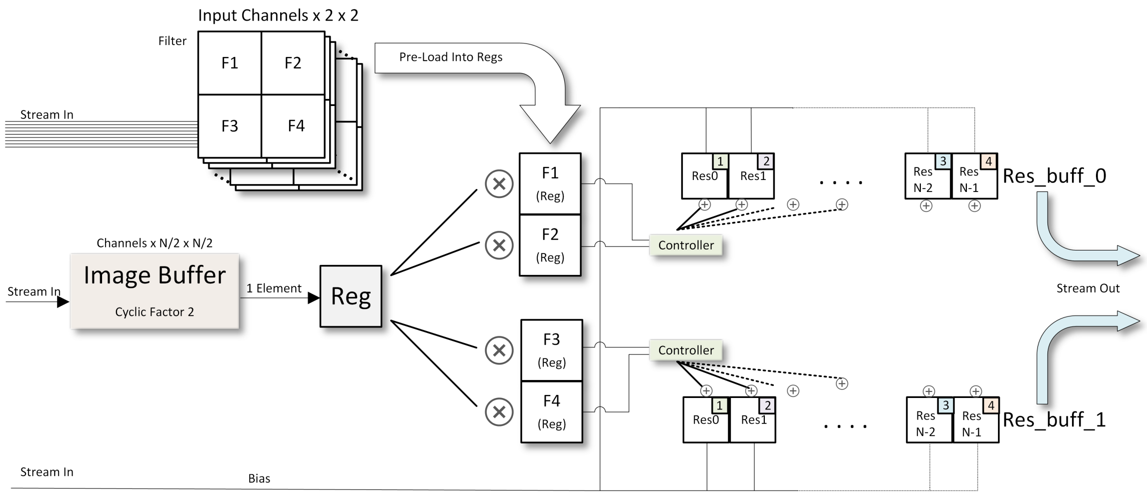


FIGURE 5.9: Architecture of the Hardware used for Transposed Convolution

Figure 5.9 describes the architecture, data structures, and information flow to/from the IP via stream -that is handled from a dedicated for deconvolution DMA- and some smaller modules like controllers, adder, and multiplier units that work in parallel in most of the computations.

Starting from the data structures, the local buffer, which is the smallest one in BRAM usage, is the so-called 'result buffers.' As the figure represents, there are two result buffers that, at first, hold a partial $2 \times output\_Columns$, which as the calculation proceeds to deeper channels, is completed and then pushed out via streaming. This approach is way different from the simple convolution since every output row is getting wholly calculated before streaming it out and carrying on for the next output row. These buffers are used quite frequently so the above can be feasible; hence the array split(banking) applied on these buffers has a factor of 4, meaning that Each of these two buffers is divided into four equally sized blocks interleaving their corresponding main buffer elements.

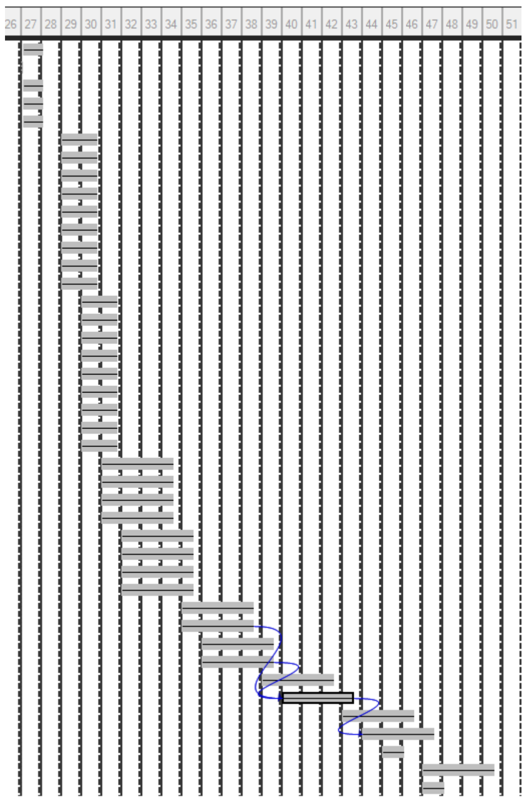


FIGURE 5.10: Transposed Convolution IP parallelism

Next up, according to its size, the filter occupies 1024(256 × 2 × 2) cells of the BRAM, supporting up to 256 channels of filters that can be stored locally. This filter's usage frequency is low for the channels change, but huge concerning the running filter since every element of the filter must be multiplied with the same input pixel. For that reason, four extra registers were created that refreshed(pre-loaded) every input channel with the new filter, and they are connected directly with the current input pixel via fixed multipliers(center of the figure). The above can produce the 2 × 2 result in 2 cycles since everything works in parallel.

Most of the BRAM usage is spent on the local image buffer that holds up to 32 × 128 × 128 = 524288 elements. That means the whole image can be store locally just like convolution, but this time the largest image is only the ¼ of the largest convolution image. The size analogies are not proportional to the speedup because transposed convolution can give up to ×250 in comparison with a standard CPU running the deconvolution on a high-level user-friendly programming language like python. This local image buffer also uses a factor 2 cyclic structure increasing its read/write ports by dividing the primary buffers into smaller ones with the same size. As the figure (pipeline HLS analysis) shows, the calculations taking two cycles involve multiplication, the one cycle load/stores, and the four cycles the additions to the output result buffer. Everything works in parallel since there is no data dependency between them. One important thing to be noted is that because of the loop unrolling, in the analysis figure, there are not just four(fully unrolled filter) calculations as expected, but eight because of the column loop it is also unrolled($\times 2$).

A brief description of how the IPs works starts with the input streaming of image(which is the largest one) loaded with the programmed DMA's help that transfers the data in bursts from DDR4 directly to IP. The IP is the one who halts and continuing the streaming flow according to its needs. That is why the BRAM is not public and accessible by the DMA but private and configurable from IP. This strategy also reduces data usage by storing only the upcoming required data and not all the whole batch. The next step is to load bias that is only required one time per output channel(on input filter). It is important to be mentioned that bias occupies just a local register that initiates the output result buffers with its values exactly in the same way as happened during the convolution part.

For each input channel running, a new filter(respectively to the input channel) is loaded to the four registers that eventually multiplied with each input pixel. When every input channel is completed, and the end of an input row is reached, it means that the two result buffers carry the final $2 \times output\_Columns$ data, which are eventually transferred back to the DDR4 with the help of the DMA controller(via AXI4-Full stream). The same happens for the rest of the input pixels until the transposed convolution computation has been finished.

Below, a pseudo-code is fully analyzed to make it easier for the algorithm affected by some small tweaks and optimization techniques like loop unrolling, pre-fetching, and the use of many different types of HLS directives.

```
///////////////////////// PSEUDO-CODE /////////////////////////

    //AXI4-STREAM
1: #pragma HLS INTERFACE axis register both port=result
2: #pragma HLS INTERFACE axis register both port=bias
```

```
   #pragma HLS INTERFACE axis register both port=filter
   #pragma HLS INTERFACE axis register both port=image

     //Array Partitioning
   #pragma HLS ARRAY_PARTITION variable=res_1 cyclic factor=4 dim=1
   #pragma HLS ARRAY_PARTITION variable=res_0 cyclic factor=4 dim=1
   #pragma HLS ARRAY_PARTITION variable=img cyclic factor=2 dim=1




//Starting with the whole image loading to the local BRAM buffer
     for(int c = 0; c < ch ; c++)
       for(int i = 0 ;i < dim ; i++)
          for(int j = 0 ; j < dim ; j++)
              image.read(img[c*dim*dim + i*dim + j]);


     float bias_t;  //just a register for bias
     /*
     Now we can start the transposed convolution(calculate every
     the channel then add up to the result buffers to receive the
     filt_num==out_channel respective result)
     */
     //number of filters == output_channels
     for (int i = 0; i < filt_num; i++)
     {
        /*
        read bias and initialize result buffer with these values for
        each filter==output_channel
        */
       bias.read(bias_t);

        //load all channel kernels for the current filter
       int ch_offset=0;
       for (int c = 0; c < filt_channels ; c++)
            for(int x = 0; x < F_DIM ; x++)
                for(int y = 0; y<F_DIM ; y++)
                    filter.read(filt[c*F_DIM*F_DIM + x*F_DIM + y]);

       //Begin the input row loop
       for(int x=0; x<height; x++)
       {
        //initialization of 2 first result rows with bias
           for(int j = 0 ; j < o_dim ; j++)
               res_0[j] = bias_t;
           for(int j = 0 ; j < o_dim ; j++)
               res_1[j] = bias_t;
```

```
      for(int j = 0; j < channels ; j++)
      {
           //preload some address offset useful for later use.
          int filt_offset1 = j*F_DIM*F_DIM ;
          int filt_offset2 = j*F_DIM*F_DIM + F_DIM;
          int img_offset = j*height*width + x*width;

           //Begin the input column loop
          for(int y = 0; y < width; y += 2)
          {
           //mult 1 pixel of image with the whole kernel
           //for each element in col calculate res.
            res_0[y*s]      += img[img_offset]*filt[filt_offset1];
            res_1[y*s]      += img[img_offset]*filt[ filt_offset2 ];
            res_0[y*s+1]    += img[img_offset]*filt[ filt_offset1 + 1 ];
            res_1[y*s+1]    += img[img_offset]*filt[ filt_offset2+ 1];
            img_offset      += 1;  //preload for the next iteration
            res_0[(y+1)*s]  += img[img_offset ]*filt[filt_offset1];
            res_1[(y+1)*s]  += img[img_offset]*filt[ filt_offset2 ];
            res_0[(y+1)*s+1]+= img[img_offset]*filt[ filt_offset1 + 1 ];
            res_1[(y+1)*s+1]+= img[img_offset]*filt[ filt_offset2+ 1];
            img_offset      += 1; //preload for the next iteration
          }
      }//end of channels loop

       //now 2 output lines are ready to stream them back
       //the current output_channel is completed
      for(int j=0;j<out_width;j++)
          result.write(res_0[j]);
      for(int j=0;j<out_width;j++)
          result.write(res_1[j]);

  }//end of input column loop

}//end of input row loop

////////////////////// End of pseudo-code  //////////////////////
```

MaxPool Accelerator

The Maxpool accelerator IP is the most simple between the custom IPs since it can downscale the image according to the kernel size and the stride. In this work, the max pool IP's settings are pre-defined. Given that the neural networks work only with images with the same height and width size, this size must be a number of the form 2^n (where n is an integer) with a maximum resolution of 256×256. Max pool accelerator IP is down-sampling the input to the half resolution to diminish the computation complexity while the image's subject is preserved, and the output size is maintained in the form of 2^n.

The minuscule BRAM usage(less than 1% on ZCU102) makes this IP suitable for any platform. The ability to support more massive inputs without any noticeable BRAM increase(after some source code tweaks) makes this IP unique, flexible, and fast at the same time.
The main idea of the line buffer is also applied for this module. More detailed, the line buffer number/size is selected according to the Kernel size. Following the same strategy as before, the N lines that will be saved locally in the IP must have the same height size with the Kernel Height. In this thesis work, a kernel of size 2×2 is used(with a stride of 2), so the number of IP's Line buffers is two. Having one line buffer for the image inputs with 'width' size includes many benefits from DDR to BRAM transfer time overlapping with the previous computation times, to fast cell access. Again, a window of Kernel size makes the comparisons between the $Kernelsize \times Kernelsize$ elements finding the one with the largest value meaning that it has the greatest impact and actual weight significance for the image. So the greatest value among the sub-matrix is selected and directly is streamed out.

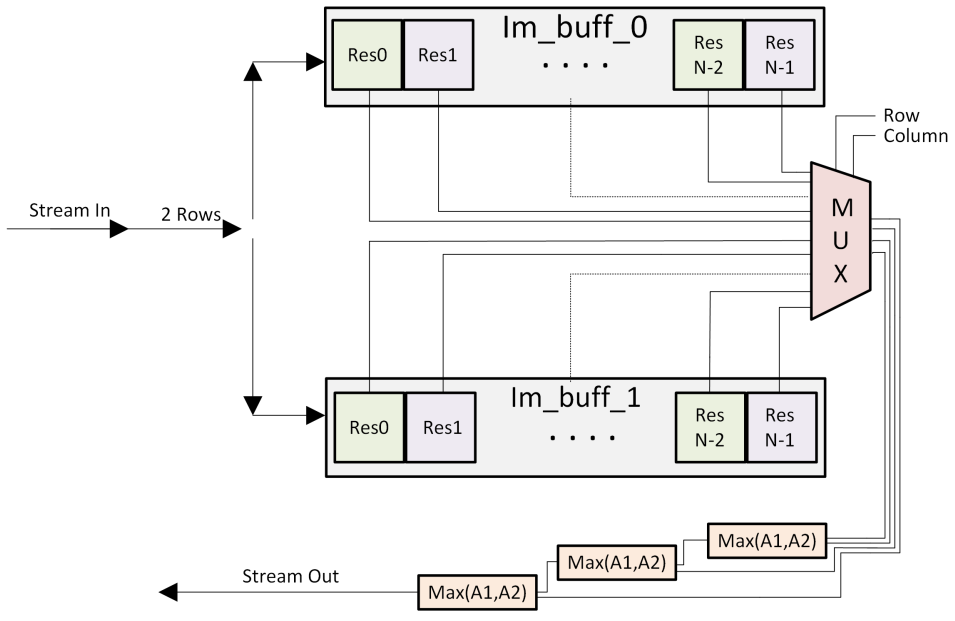


FIGURE 5.11: Architecture of the hardware used, for MaxPool

In terms of the hardware architecture, the final IP consists of two main parts:

the line buffers and the comparators connected with a multiplexer's help. AXI4-stream is the streaming solution used for the input/output. The input inserts the module in batches of KernelSize rows and Image width columns. For this purpose, the local buffer is the basic BRAM with two ports, one for reading and one for writing. Furthermore, each buffer is split into two smaller equal-sized arrays doubling its ports and increasing the throughput. For this work, where the kernel size is two and the stride is also 2, the main buffers' division into two smaller is enough since the actual computational core needs are four elements, two from each buffer. The multiplier role, which is controlled from the current running column of the process, chooses the respective sub-matrix from the buffer to drive it to the comparators section. The logic behind the selected sub-matrix follows the function $curr\_output\_column * stride \rightarrow curr\_output\_column * stride + kernelsize$ for both buffers.
The comparators work in a tree structure comparing two elements each time until there is only one final value, the greatest of all.

```
////////////////////// PSEUDO-CODE //////////////////////

    //AXI4-Stream
1:  #pragma HLS INTERFACE axis register both port=image
2:  #pragma HLS INTERFACE axis register both port=result

    //Line buffers array partition
3:  #pragma HLS ARRAY_PARTITION variable=img_t0 cyclic factor=2 dim=1
4:  #pragma HLS ARRAY_PARTITION variable=img_t1 cyclic factor=2 dim=1


5:  float max = -100000; //Initiate 'max' register with -inf

    //Channel loop starts
6:  for(int i = 0 ; i < in_channels ; i++)
7:  {
        //Output row loop start
8:     for(int x = 0; x < output_dim ; x++)
9:     {
            /*
            Filling up the 2 line buffers with width size
            elements(2 new rows per fill)
            */
10:        for (int z = 0 ; z < width;  z++)
11:            image.read(img_t0[z]);
12:        for (int z = 0 ; z < width; z++)
13:            image.read(img_t1[z]);

            //comparators Part - Tree structure
14:        for (int y = 0 ; y < output_dim;  y++)
```

```
        {
            if(img_t0[stride*y] > max)
                max = img_t0[stride*y];
            if(img_t0[stride*y+1] > max)
                max = img_t0[stride*y + 1];
            if(img_t1[stride*y] > max)
                max = img_t1[stride*y];
            if(img_t1[stride*y + 1] > max)
                max = img_t1[s*y + 1];
            result.write(max);
            max = -100000;  //reset max variable with the -inf
        }

    }\\End of row loop

}//End of Channel loop

////////////////////// End of pseudo-code //////////////////////
```

5.4 Improved Architecture Version 2.0

In this section, every key difference will be presented and analyzed compared with the previous version. Version 2.0 of the UNET Implementation is based on the original design described before and aims to reduce latency while increasing power/energy efficiency by taking advantage of the parallelization achieved on FPGA. Applying a slight tweak to the high customizable template provided as an initiate design makes it possible to accomplish a 10-20x gain in performance.

5.4.1 Accelerators Design Updates - Analysis

The simplest and indicative form of the architecture is made for extreme tweaking and adjusting as the user desires. The Convolution Accelerator carries the most of the changes compared to the Transposed Convolution, while Maxpool IP has undergone some minor changes to its data types. Some critical elements of the design consist of the data stream of a big data-packed vector(using multiple DMAs), line buffers, parallel multiplications, the computation 'window,' and the BRAM fragmentation (break down into smaller arrays for less local read/store limitations). This slightly altered version of the design includes changes in BRAM partitioning, data form, and the computation window. In the following two sections, every single deviation from the original design will be presented and analyzed.

A significant change from naive architecture and affects all the IPs has to do with the data type. From the floating-point that was used initially, this version moves to fixed-point data types. This dramatically helps loading / storing times and computation complexity. The latency of the computations is sliced in half compared with the initial data type. FPGA platforms are generally known for their advantage on floating-point arithmetic using a specialized accelerator part for this purpose. The fixed-point's overall bit size is the same signed 32-bit long, providing a total of 20 bits for the integer part(including the sign bit) and the rest 12 bits for the fractional part.

Convolution Accelerator

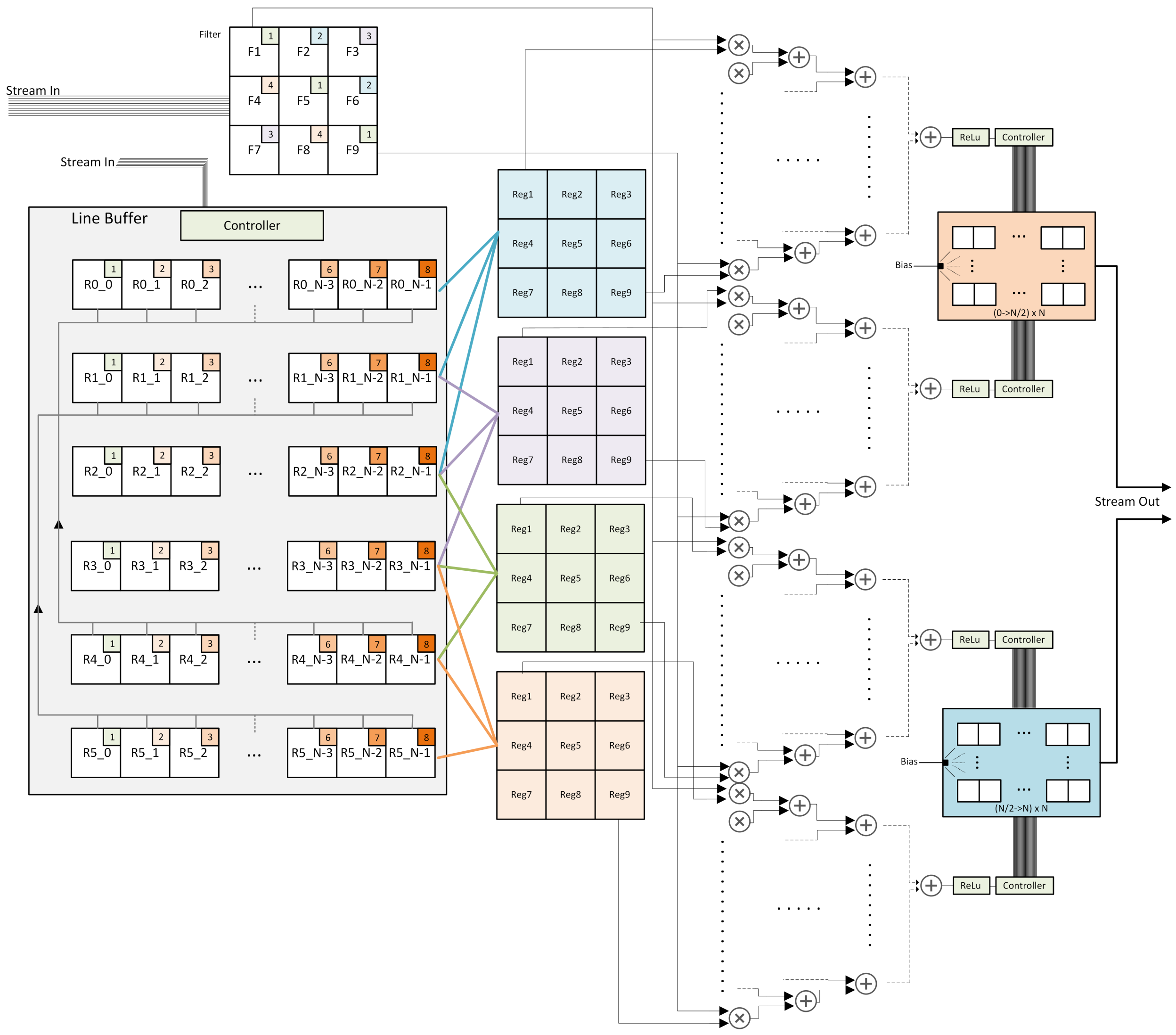


FIGURE 5.12: Architecture(v2) of the Hardware used for Convolution

Figure 5.12 demonstrates the improved convolution architecture by breaking down any necessary modules that are used to construct this accelerator.
First of all, this design's data structures differ because each data type(image, filters, bias) needs a special treatment according to the frequency, repeatability of data, read/write patterns, and size. This module's BRAM utilization is about 10% of the overall Block RAM available to the ZCU102.

The main extension from the basic version of this IP took place in the processing window. In the new design, the line buffers increased from three to six. As a result, six rows of the image can be stored, and a total of four parallel calculations can be executed. The same strategy followed for the row axis was also applied in the column axis producing eight output results per row. More detailed, the input controller loads $2 \times N$ (during initialization) to the Line Buffer(LB) zero and one. Each time the algorithm proceeds to a new group of output rows(four), $4 \times N$ lines are saved to the following $LB2, 3, 4, 5$ buffers. $LB4$ and $LB5$ are connected(for shift up) with the $LB0$ and $LB1$ respectively. The reason for that connection is that, in this window of six-line buffers, only four convolutions can be executed per group of three lines because the kernel has a size of three. These groups are:
$(LB0, LB1, LB2)$, $(LB1, LB2, LB3)$, $(LB2, LB3, LB4)$ and $(LB3, LB4, LB5)$.
Every time a push happens to the last four line buffers, two shift-up actions have receded, making sure the old data of (Line Buffer 4)$LB4$ are shifted to the $LB0$ and respectively, the $LB5$'s data are shifted up to the $LB1$, so they are ready for the next iterations.

The intuition behind these big windows is to produce a 4×8 output per algorithm iteration, reducing the loop overheads, increasing the parallelization while taking advantage of the data bursts that work better with larger chunks of data. The previous design was requesting N elements (one Line) for every iteration compared to the new design, which can read a four times larger size and produce even more dense output results in a single iteration.

Each line buffer is implemented as block Ram, which only has a maximum number of 2 data ports. On top of that, each line buffer is divided into eight smaller BRAM modules to support the eight parallel calculation for each line buffer(the colors/numbers at the top right side of each cell can show on which module is distributed 5.12). With banking, which is the partitioning in a 'cyclic' setup, each memory bank stores only a fraction of the total data. For example, each line buffer can support eight different actions. To sum up, six different buffers, each with a native eight data ports, are further split into eight smaller 'arrays'.

Of course, the last actions result even more in the BRAM utilization burden. The bias values are already stored in the output buffers by the initialization, which is more efficient than filling with zeros(same as the original design). For the bias, no local buffer is used, just a register for the transfer to the output result, which means that it can be loaded directly from the stream provided by the DMA. The output buffer is also doubled like the line buffers. The new design has two output buffers, increasing the ports since the outputs are way more populated than before. The partition strategy applied to these output buffers follows a partition of eight(for each one).

Simultaneously, a different DMA channel is programmed by the central processing unit to transfer one channel of the input image each time to a local private BRAM storage handled by the IP. The IP accepts and saves the image before the actual computation part even starts. The same happens for the filter input stream, where each time the input channel changes, a new 3x3 filter is loaded via stream. The local filter buffer is split into four different 'arrays,' so the maximum throughput is achieved. Multiplications between the sub-matrix(window), where the filter is sliding on, and the actual filter are organized to a tree-structure logic to utilize the pipelining capabilities starting every single addition and multiplication the earliest possible.

Transposed Convolution Accelerator

Figure 5.13 describes the improved architecture, data structures, and information flow to/from the IP via stream -that is handled from a dedicated for deconvolution DMA- and some smaller modules like controllers, adder, and multiplier units that work in parallel in most of the computations.

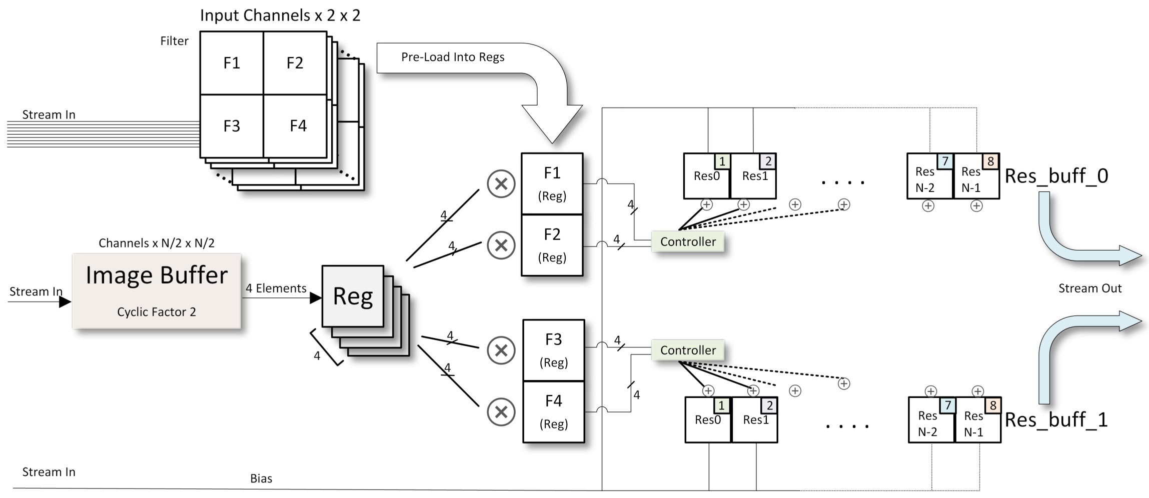


FIGURE 5.13: Architecture(v2) of the Hardware used for Transposed Convolution

Starting from the data structures, most of the local buffers are untouched in this new architecture version. The output buffers, which are two separate blocks of local Ram, are used quite frequently; hence the array split(banking) applied on these buffers has a factor of 8, compared with the factor of 4 used in the initial design. Each of these two buffers is divided into eight equally sized blocks interleaving their corresponding main buffer elements.

According to its size, the filter occupies 1024(256 × 2 × 2) cells of the BRAM, supporting up to 256 channels of filters that can be stored locally, at the same way as before. Four extra registers were created that are being refreshed(pre-loaded) every input channel with the new filter, and they are connected directly with a group of 4 input pixels via fixed multipliers(center of the figure) that creates a four-times more parallelized multiplication than the previous version(with just one input pixel). Each of the input pixels will be multiplied precisely in the same way as before(with all four filter registers, producing a 4×4 output). That also happens for the rest three of the input pixels in parallel. The output for each iteration becomes 2x8 compared with the naive output of 2×2 per iteration. Again, most of the BRAM usage is spent on the local image buffer that holds up to $32 \times 128 \times 128 = 524288$ elements. That means the whole image can be stored locally. This local image buffer also uses a factor 2 cyclic structure increasing its read/write ports by dividing the primary buffers into smaller ones of the same size. As the figure (pipeline HLS analysis) shows, the calculations are significantly reduced in latency, similar to the convolution accelerator due to the fixed-point data type. Everything works in parallel since there is no data dependency between them.

A brief description of how the IPs works starts with the input streaming of an image(which is the largest one) loaded with the programmed DMA's help that transfers the data in bursts from DDR4 directly to IP. The IP is the one who halts and continuing the streaming flow according to its needs. That is why the BRAM is not public and accessible by the DMA but private and configurable from IP. This strategy also reduces data usage by storing only the upcoming required data and not all the whole batch. The next step is to load bias that is only required one time per output channel(on input filter). It is important to be mentioned that bias occupies just a local register that initiates the output result buffers with its values exactly in the same way as happened during the convolution part.

For each input channel running, a new filter(respectively to the input channel) is loaded to the four registers that eventually multiplied with the group of four input pixels saved in registers. The above has the same functionality compared to the single input pixel (in the initial design. In this version, the amount of computation and output has been quadrupled. When every input channel is completed, and the end of an input row is reached, it means that the two result buffers carry the final $2 \times output\_Columns$ data, which are eventually transferred back to the DDR4 with the help of the DMA controller(via AXI4-Full stream). The same happens for the rest of the input pixels until the transposed convolution computation has been finished.

MaxPool Accelerator

Maxpool accelerator remains the same in terms of design, except that the data type of input/output and local computation has been updated to the new type variable of fixed-point. That significantly reduced the read/store, multiplication, and addition activities to half the latency compared to the initial logic. As a result, maxpool IP is now more powerful than GPU(Matlab PCT) in execution time and more power/energy-efficient, producing more results per joule.

Chapter 6

Results

The Results chapter focuses on providing details and benchmarks on many different aspects of performance and functionality outputs. Some simple experiments executed on the custom training tools show the capabilities of this entry-level mechanism. Furthermore, power analysis, throughput, latency, and energy consumption are the key categories where the platforms will be later compared.

In the section ('training'), the training performance and accuracy will be described. Considering that the neural network's training part is developed to run on simple python using the Numpy library only, it provides a profound and unique -inside 'open source' world- demonstration on a fully customizable back-propagation process that verifies the structure. In the section 'ZCU platform', the ZCU platform is being described and then compared (section) with some alternative platforms/processing units where the UNET neural network is implemented.

6.1 Training

The training session is tested in a limited data-set to inspect the neural network's learning capabilities and its rigid functionality. The tiny data-set consists of 4 different images from the TGS Salt identification challenge by Kaggle [62] that aims to discover salt in the sea by analyzing images taken from a satellite orbiting the earth at low altitude. Some sparse examples of brain tumors also included(Dataset of 4) as well as some rat muscle cell on microscopy by Boston University - Biomedical Image Library [63], showing the abstract capabilities, wide range, and robustness of the training implementation. The train function is initialized with the following recommended setup (which is the same for all the subjects and categories):

- *Dropout:* Disabled, Group Normalization: Enabled(batch= 2)
- *Binary classifier:* Dice Coefficient

- *Weight Initialization strategy:* Normal distribution with weight scale: 0.1
- *Learning rate:* 0.008
- *Adam Optimization:* Enabled
- *Image Resolution:* 64x64

**Note = 64x64 is the maximum resolution recommended for the training part because of the low training speed running on a single CPU core while using only the basic library Numpy.*

The figure below illustrates some of the training results, which are interestingly accurate, confirming the U-net functionality. The rough edges in some parts of the predicted outputs can be smoothed over(according to application/subject) by tweaking the bias trim value.

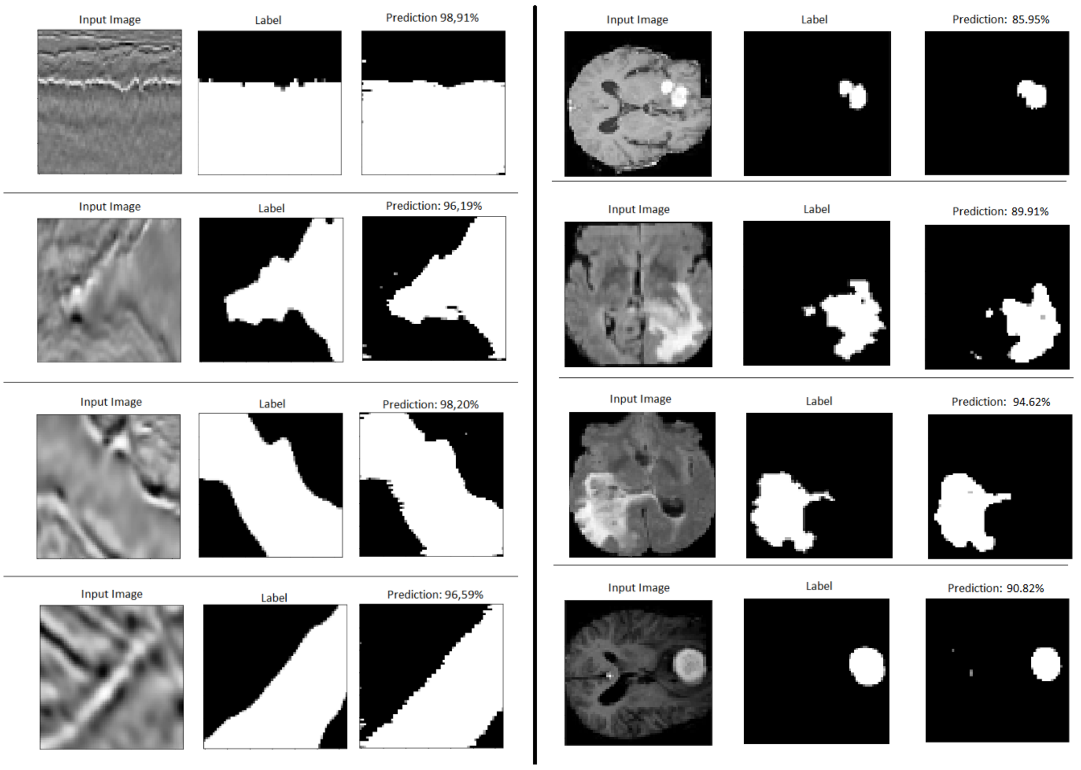


FIGURE 6.1: Training Results

Moreover, the average training(example: TGS Salt) accuracy on par with the training epochs is presented in figure 6.2. The time needed for the passage of one forward/backward propagation, for the settings and image resolution mentioned earlier, is approximately 20 seconds, which means 50 epochs need about 17 minutes to complete. Again, these numbers are not the main focus of this work, but the construction of this low-level training function, which for the moment is not available as open-source knowledge. Eventually, this work can seem very useful for the developers who want to implement such a process on any other platform/device written in a specialized environment that provides optimizations and colossal performance boost (like CUDA for GPU acceleration, VHDL for FPGA IP accelerators).

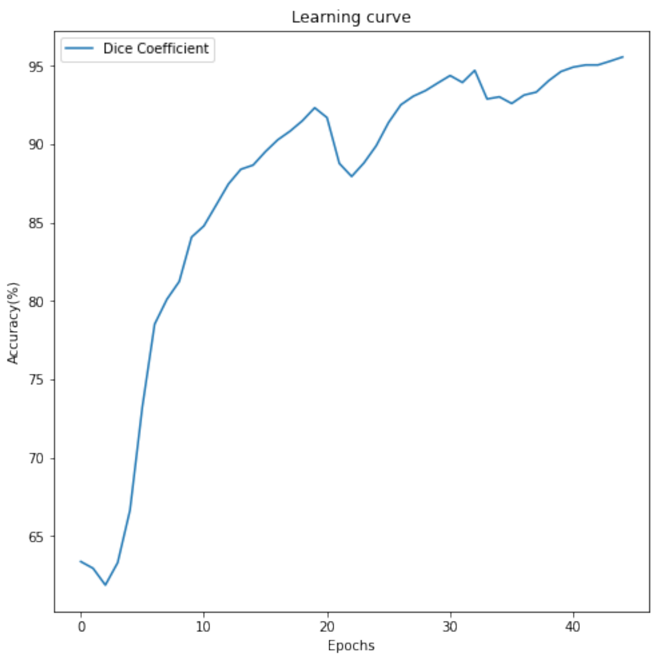


FIGURE 6.2: Training Accuracy over epochs

6.2 Specifications of the Compared Platforms and abstraction Levels

The following observation, for the forward step, includes an abstract pool of devices/environments -i5 3210M [64], NVIDIA GTX 750Ti [65], Python, C, Matlab Parallel Computing Toolbox [16], Keras- since there are no other published results regarding the standard UNET architecture running on FPGA or any other embedded system/Custom Accelerator.

6.2.1 ZCU102

ZC102 was selected for this work, providing enough BRAM resources for the testing phase, where the whole U-NET architecture can be implemented, boosting three processes: Maxpooling, Convolution, and Transposed Convolution with the main focus to be on the less known Deconvolution(transposed Convolution). This approach's variable type is the default 32-bit floating-point, which can easily be modified to 16 or even an 8-bit fixed point domain for the maximum Zynq Ultra-Scale+ MPSoC utilization. There are multiple proposed strategies in chapter 7 that can help the abrupt performance soaring.

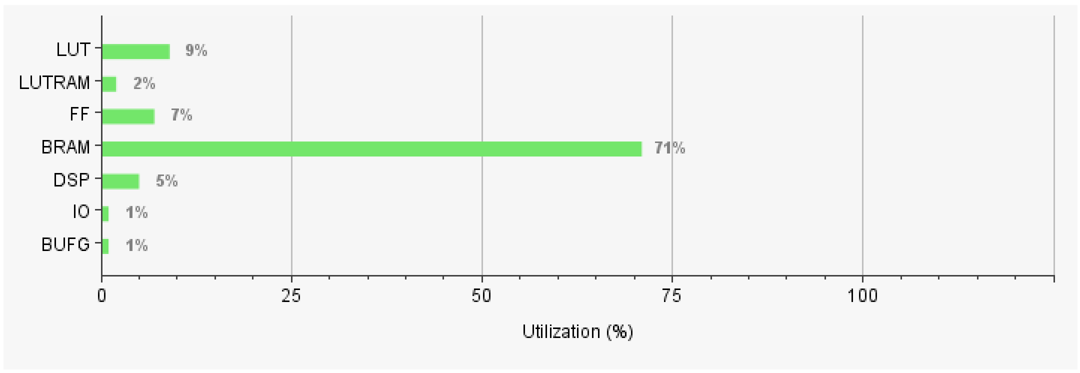


FIGURE 6.3: Overall Zynq UltraScale+ MPSoC Utilization

6.2.2 Intel i5 3210M

TABLE 6.1: Intel i5 3210M - Specifications Table

Lithography	22nm
Cores/Threads	2/4
Clock Frequency (Max)	3.1GHz
Cache (L3)	3MB
TDP	35W
Max Memory Bandwidth	25.6GB/s

6.2.3 NVIDIA GTX 750Ti

TABLE 6.2: GTX 750Ti Gold Sample(Palit) - Specifications Table

Lithography	22nm
Shading Units	640
Clock Frequency (Max)	1281MHz
TDP	60W
Memory Clock	1502 MHz
Max Memory Bandwidth	96.13GB/s
Compute Capability	5.0
FP32 Performance	1.640 TFLOPS

6.2.4 Parallel Computing Toolbox - Matlab

Parallel Computing Toolbox [16] utilizes many acceleration methods both for CPU and GPU. Loop unrolling, creation of parallel loops, parameters sweeps, multi-core distribution of the workload(with/without batch grouping), and other vital functions are used to achieve the highest result for the given hardware. CUDA support also enables the users to write the source code with a few specialized functions which translate the arrays into useful GPU data(GPUarray). The performance tuning is not an obstacle during programming since everything is handled directly by the Matlab toolbox so that the end-user can be focused on real application testing.

6.2.5 Software Prototype implementations: Jupyter Notebook(Numpy) & Eclipse CDT

The use of Jupyter Notebook can be perceived as a friendly tool where anyone can use, test, and tune a simple UNET architecture while involving some essential parts that expose and depict the architecture's inner side. There are no particular functions and libraries for deep-learning acceleration (such as TensorFlow). Matrix-to-matrix calculations are held by NumPy that reduces the complexity of the source code with the compact macros without the need for a detailed computation.

Eclipse CDT [66] is an Integrated Development Environment based on Eclipse specialized for C/C++ development. Since the higher levels are functional, the source code from the latest step(Python - Jupyter Notebook - NumPy) needs to be decomposed even further into smaller pieces. The libraries used in Python, including matrix multiplications, rotations, n-dimension array creations, image preprocessing, equations, image loading tools, data transferring, optimized struct types, and initialization methods, must be reproduced into C level. For implementing an embedded system, the libraries mentioned above and algorithms' analysis is a vital step. This way, it is possible for the researcher to locate intense computational sections and data-hungry functions that need to be elevated in performance.

6.3 Overall/Sub-part Latency Comparisons

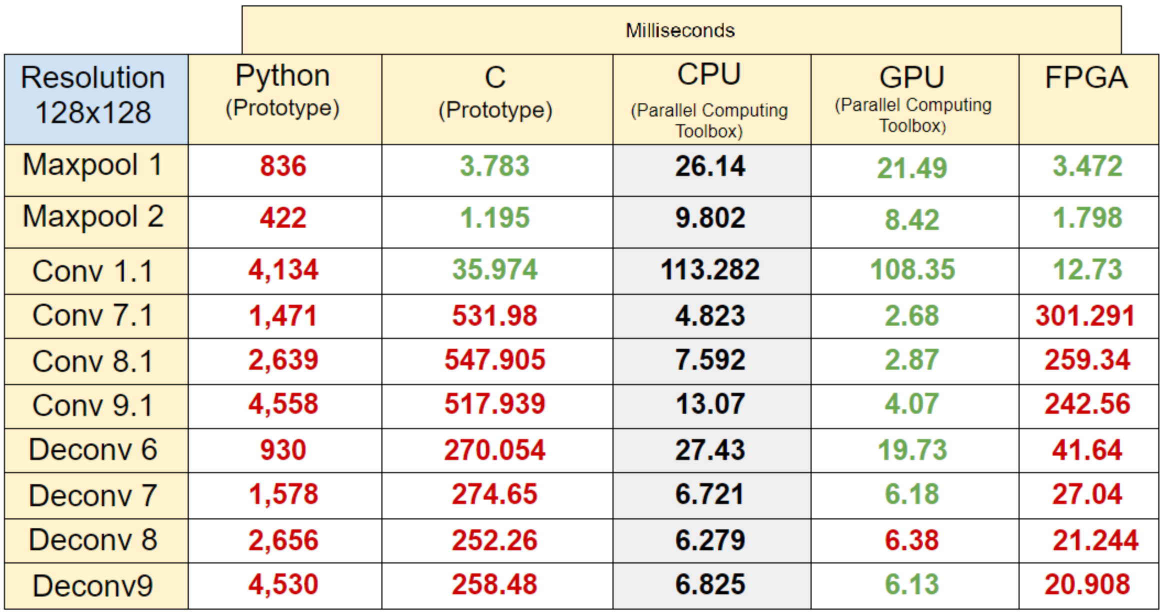
In this short section, some of the supported platforms will be used to demonstrate some latency results per layer of the U-NET. Custom Python(Numpy), C, Matlab's Parallel Computing Toolbox(CPU/GPU), and ZCU102 are eligible for this per layer measurement. This initial architecture is only the basic block -built for evaluation and proof of concept- that can be further extended, tweaked and optimized to reach the desired performance. The more advanced version 2 of the architecture is presented in the following section(6.5).

6.3.1 Latency

Latency is the delay window between two points on the time axis. Often less latency is preferred since a system's performance can be measured in latency or time of execution until a single result is generated.

Below (Figure 6.4), some layers of each category(max pool, convolution, transposed convolution) are presented. The red values show a slower performance compared to the reference platform(gray column) while green values indicate a faster result.

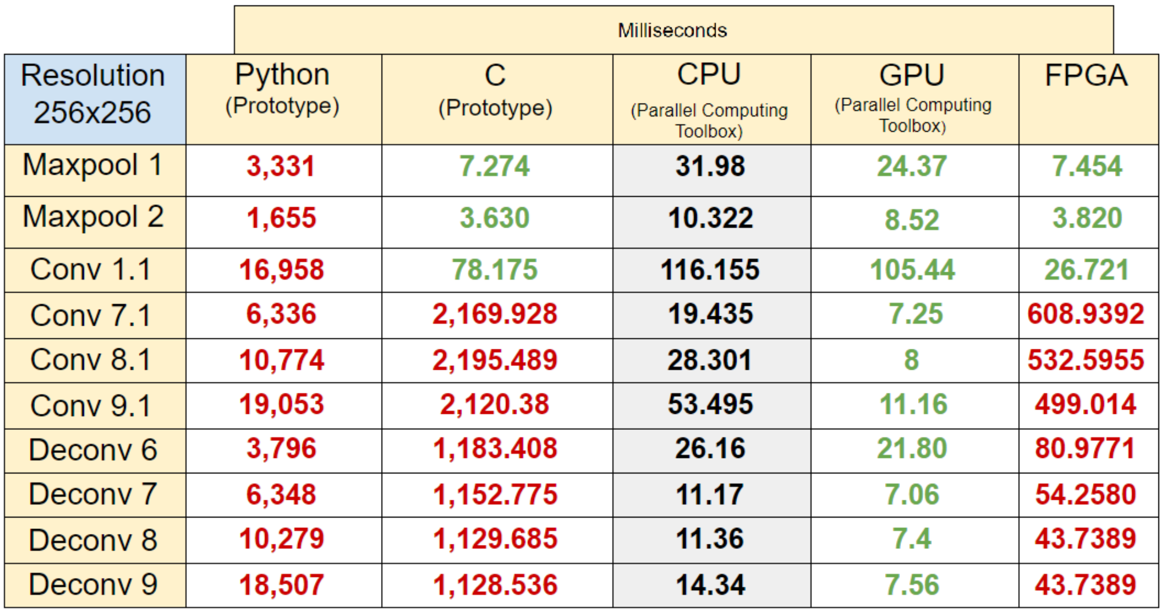
The values are calculated after multiple simulations/executions representing the average performance of each system. A batch of one was used for all the following examples 6.4, 6.5.



	Milliseconds				
Resolution 128x128	Python (Prototype)	C (Prototype)	CPU (Parallel Computing Toolbox)	GPU (Parallel Computing Toolbox)	FPGA
Maxpool 1	836	3.783	26.14	21.49	3.472
Maxpool 2	422	1.195	9.802	8.42	1.798
Conv 1.1	4,134	35.974	113.282	108.35	12.73
Conv 7.1	1,471	531.98	4.823	2.68	301.291
Conv 8.1	2,639	547.905	7.592	2.87	259.34
Conv 9.1	4,558	517.939	13.07	4.07	242.56
Deconv 6	930	270.054	27.43	19.73	41.64
Deconv 7	1,578	274.65	6.721	6.18	27.04
Deconv 8	2,656	252.26	6.279	6.38	21.244
Deconv9	4,530	258.48	6.825	6.13	20.908

FIGURE 6.4: Latency measurements for 128 × 128 resolution in milliseconds

**Note: The native resolution for the current architecture is* 128 × 128



	Milliseconds				
Resolution 256x256	Python (Prototype)	C (Prototype)	CPU (Parallel Computing Toolbox)	GPU (Parallel Computing Toolbox)	FPGA
Maxpool 1	3,331	7.274	31.98	24.37	7.454
Maxpool 2	1,655	3.630	10.322	8.52	3.820
Conv 1.1	16,958	78.175	116.155	105.44	26.721
Conv 7.1	6,336	2,169.928	19.435	7.25	608.9392
Conv 8.1	10,774	2,195.489	28.301	8	532.5955
Conv 9.1	19,053	2,120.38	53.495	11.16	499.014
Deconv 6	3,796	1,183.408	26.16	21.80	80.9771
Deconv 7	6,348	1,152.775	11.17	7.06	54.2580
Deconv 8	10,279	1,129.685	11.36	7.4	43.7389
Deconv 9	18,507	1,128.536	14.34	7.56	43.7389

FIGURE 6.5: Latency measurements for 256 × 256 resolution in milliseconds

The overall Latency of each platform for both 128 × 128 and 256 × 256 are shown below:

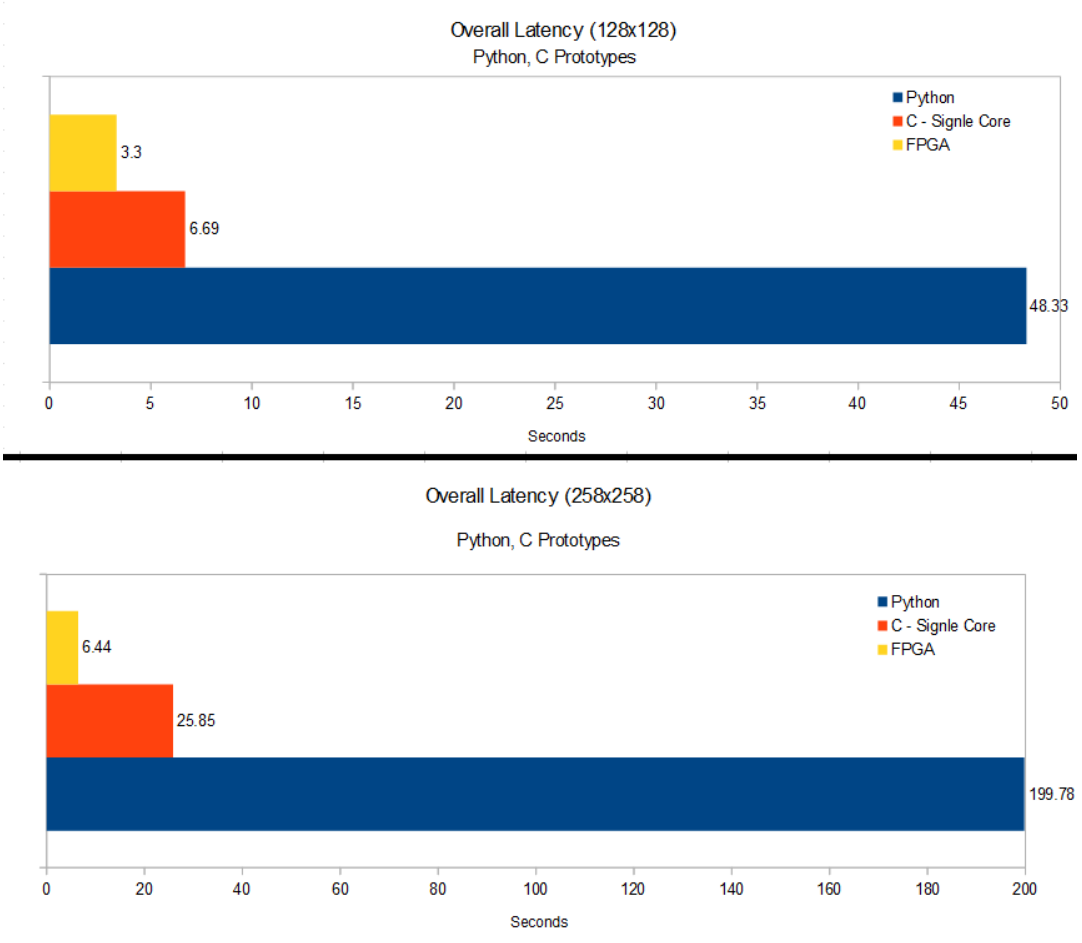


FIGURE 6.6: Overall Latency for: (128×128) & (256×256)

The prototypes' (Pythoc, C) performance is not optimal and lacks behind the other platforms. The prototype solutions do not exploit all the available hardware resources resulting in a poor scaling pattern as the input images' resolution increases. The parallel computing toolbox gives greater latency values, resulting in higher rates of throughput. ZCU102 with the current light optimization techniques, cannot compete with any high-level environment or professional hardware (which is not the purpose of this work but to provide a prototype ecosystem of UNET and the assisting tools in every level of abstraction having in scope every researcher aiming for an embedded approach/endeavor). Although, transposed convolution IP offers some interesting results in terms of latency-power.

6.4 Final Indicative Performance

6.4.1 Throughput

Throughput, in general terms, is the number of units that goes through a process per unit time or more simple, the rate of production or flow rate at which a task is completed. Generally, higher throughput values means better results in terms of hardware performance.

R: Throughput: Number of tasks that go through a process per unit time
I: The number of tasks that are currently under a process. It is measured in number of units
T: This is the time that the tasks spend from the beginning to the end of a process

$$R = \frac{I}{T} \tag{6.1}$$

6.4.2 Power Consumption

Power consumption is the amount of a natural resource(or watt in this case) consumption that is used for the functionality of a device per unit time. Energy efficiency of a system can be improved by diminishing the average Power consumption(Wh) by simplifying and using a smaller architecture for a design.

6.4.3 Energy Consumption

A task requires a specific amount of energy in a given time in order to completely compute a task assignment, which called energy consumption. This metric values are preferred to stay a the lowest levels for the accomplishment of a assigned task.

E: Energy, in Joules
P: Total power for the device to function
T: The time is needed for the task to be executed.

$$E = P * T \tag{6.2}$$

The following equation will be useful for the final results that will follow:

$$\frac{EnergyConsumption}{Image} = min\left(\frac{Power(Total)}{Throughput}, Power(Total) * Latency\right) \tag{6.3}$$

$$\frac{Images}{Joule} = max\left(\frac{Throughput}{Power(Total)}, \frac{1}{Power(Total) * Latency}\right) \tag{6.4}$$

Table 6.3 depicts the comparison results between the prototype software implementations running on single CPU core and the ZCU102 FPGA at the maximum 256x256 resolution. The efficiency and speedup values are calculated compared to the C environment(Single CPU Core - no optimizations) for batch= 1.

TABLE 6.3: Performance Analysis - Prototypes & FPGA

	Python(Numpy)	**C(Eclipse)**	**ZCU102**
Clock Frequency (MHz)	3100	3100	200
Latency (s)	199.78	25.85	6.44
Latency Speedup	0.129	1x	4.013x
Throughput (Images/s)	0.005	0.04	0.16
Throughput Speedup	0.125x	1x	4x
Total On-Chip Power (Watt)	8.75	8.75	4.68
Power Efficiency	1x	1x	1.87x
Energy Consumption (Joule)	1748.11	226.19	30.14
Energy Efficiency	0.129x	1x	7.504x
Images/Joule	0.0006	0.0044	0.03

A visualization of the table 6.3 is presented below:

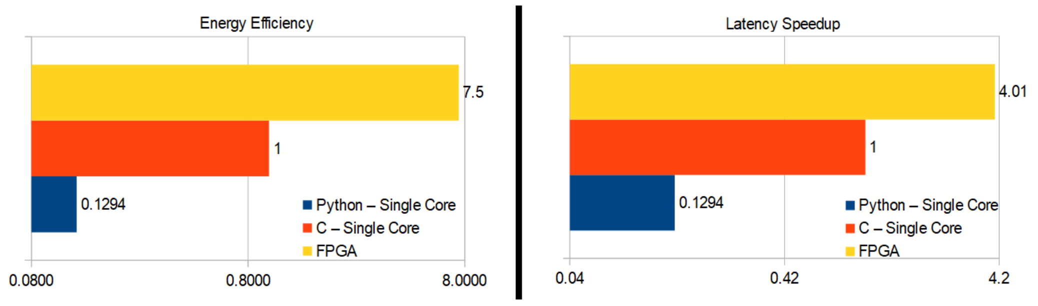


FIGURE 6.7: Latency Speedup & Energy Efficiency chart

The figure 6.7 above shows that ZCU102 is leading in every aspect of the tests, from latency to power measurements with the given batch size of one. These tests are built to provide a general visualization of the original UNET architecture running on FPGA, so the classic and friendly environment served from Python that can be used by anyone can also be significantly accelerated or even executed on a smaller embedded device.

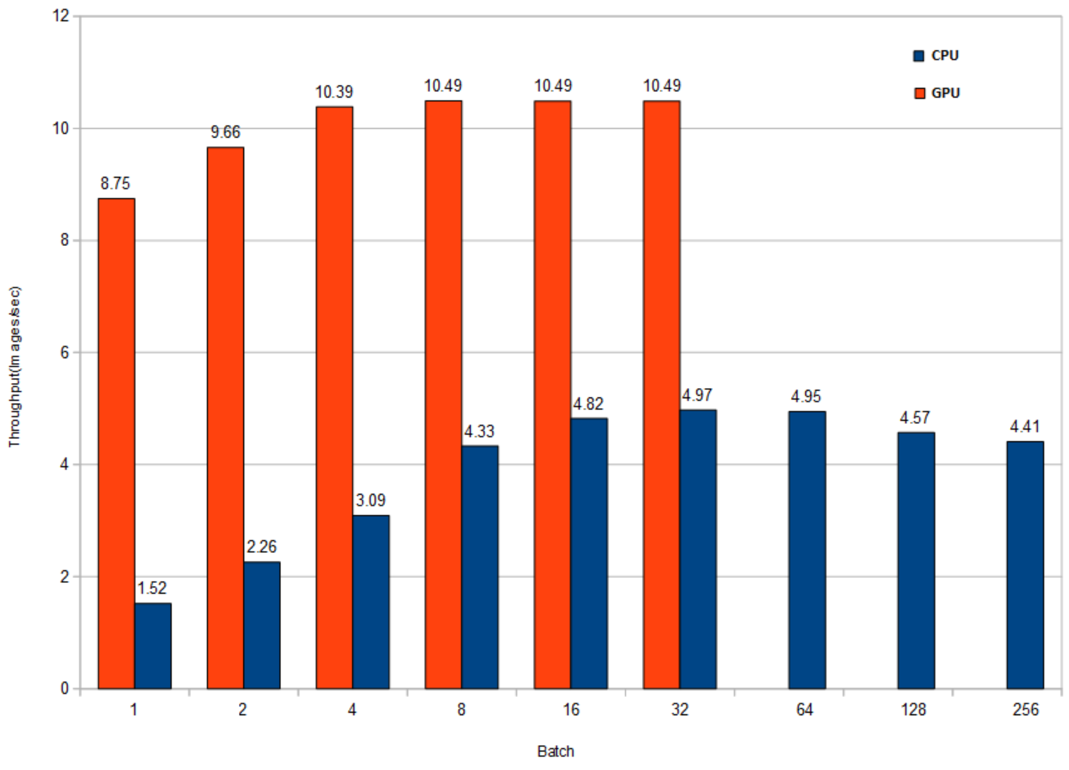


FIGURE 6.8: Matlab(PCT) CPU & GPU throughput per batch size (I/O not Included)

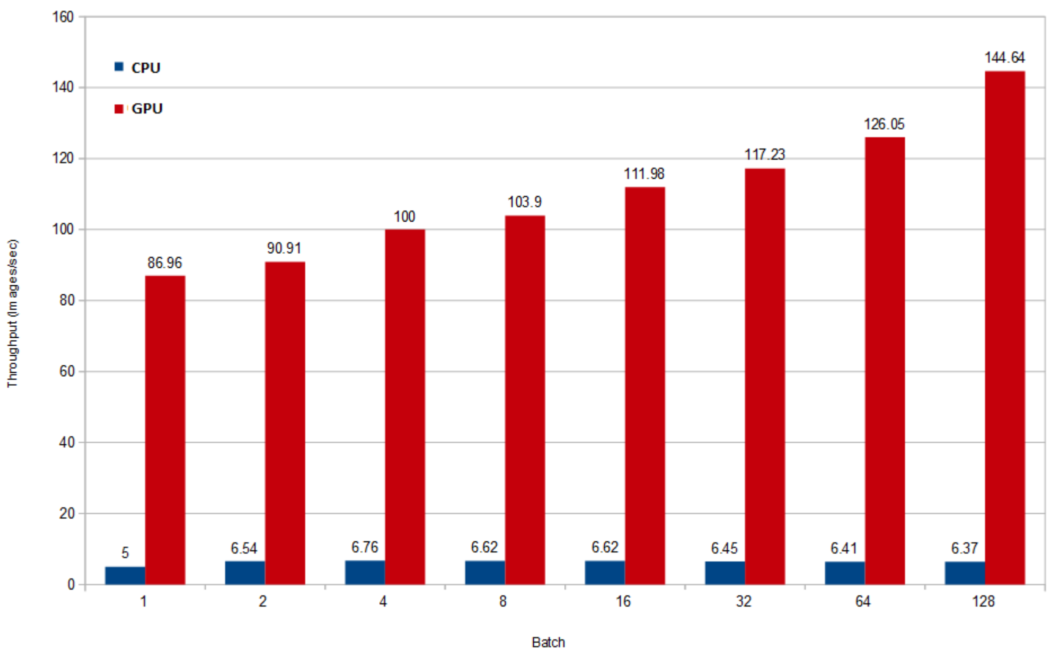


FIGURE 6.9: Keras CPU & GPU throughput per batch size (I/O not Included)

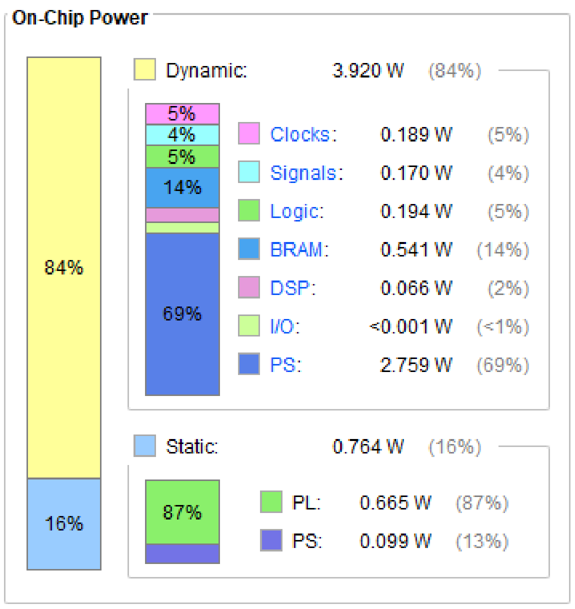
Table 6.4 extends the comparison of table 6.3 to more advanced/powerful platforms like Matlab's Parallel Computing Toolbox(PCT) and Keras-Tensorflow running on both CPU and GPU. The efficiency and speedup values are calculated compared to the CPU(Keras) environment(Multi-Core - Heavy optimizations). The throughput values are selected optimally with respect to the previous throughput-per-batch figures. 6.9,6.8.

TABLE 6.4: Performance analysis - High Level Environments & FPGA for the maximum supported resolution of 256×256

	CPU(PCT)	GPU(PCT)	CPU(Keras)	GPU(Keras)	ZCU102
Clock Frequency (MHz)	3110	1281	3100	3100	200
Latency (s)	0.66	0.17	0.2	0.0115	6.44
Latency Speedup	0.304x	1.16x	1x	17.39x	0.03
Throughput (Images/s)	4.97	10.49	6.76	144.64	0.16
Throughput Speedup	0.735x	1.55x	1x	21.39x	0.02x
Total On-Chip Power (Watt)	35	60	35	60	4.68
Power Efficiency	1x	0.583x	1x	0.583x	7.48x
Energy Consumption (Joule)	23.03	10.35	5.18	0.69	30.14
Energy Efficiency	0.225x	0.5x	1x	7.503x	0.17x
Images/Joule	0.043	0.097	0.19	1.449	0.03

*The benchmarks above illustrate a indicative promo(proof of concept) of the initial ’template’ architecture which is build as a highly adjustable back bone. The Version 2 of the architecture is presented in the next section providing a huge boosted compared with the base design.

The latency between the High-level environments versus the FPGA is quite considerable. GPU is the most powerful device among the rest, but at the same time, it is the one with the highest power demand reaching 60 Watts. On the other side of the coin, FPGA can be slower and less energy efficient, but the fact that 4.68 can compute such a heavy workload is quite impressive. Having 4.68 Watts as the power consumption value means that the FPGA platform can be supplied from just small batteries [67], which can provide 1.2-1.8 Volts at 50mA constant for hours, producing about 3-5 Wh according to its chemical composition. The part where FPGA has greatly improved is where transposed convolution takes place. This work’s simple convolution IP is not optimal and can be optimized or even replaced with other already published convolution IPs that are common and familiar to anyone.



On-Chip Power

🟨	Dynamic:	3.920 W	(84%)
🟪	Clocks:	0.189 W	(5%)
🟦	Signals:	0.170 W	(4%)
🟩	Logic:	0.194 W	(5%)
🟦	BRAM:	0.541 W	(14%)
🟪	DSP:	0.066 W	(2%)
🟩	I/O:	<0.001 W	(<1%)
🟦	PS:	2.759 W	(69%)
🟦	Static:	0.764 W	(16%)
🟩	PL:	0.665 W	(87%)
🟪	PS:	0.099 W	(13%)

FIGURE 6.10: On-Chip Power Report

The FPGA clock can also be increased with or without any architectural changes. The medium BRAM utilization makes the architecture to be compatible with a broader family of FPGA platforms. As mentioned earlier, the results and comparisons are just some indicative visualizations, with the main focus of this work to be the ecosystem between all these environments from high-level training to

FPGA implementation that was not available to everyone till today, so even more researchers can step on some work to further accelerate the classic architecture of U-NET.

6.5 Architecture Version 2.0

In this section, the results and comparisons between all the abstraction levels and platforms will be presented. The indicative/initial design has been customized; therefore, a refreshed version of the architecture is being made. This new design is comparable with the existing systems in terms of Latency when the power sector dominates most cases.

This approach's variable type is 32-bit fixed-point, providing a total of 20 bits for the integer part(including the sign bit) and the rest 12 bits for the fractional part. Below 6.11 the total utilization of the ZCU102 is visualized.

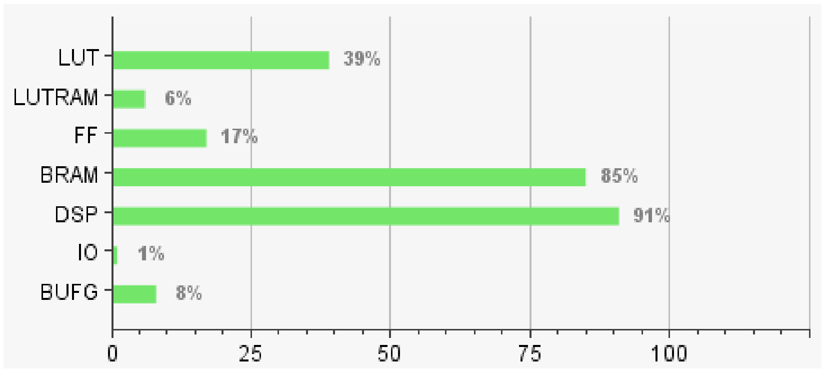


FIGURE 6.11: Overall Zynq UltraScale+ MPSoC Utilization - V2

Since, the utilization of the PL fabric has been increased, the total power consumption(fig: 6.12) subsequently reached a peak of approximately 7 Watt.

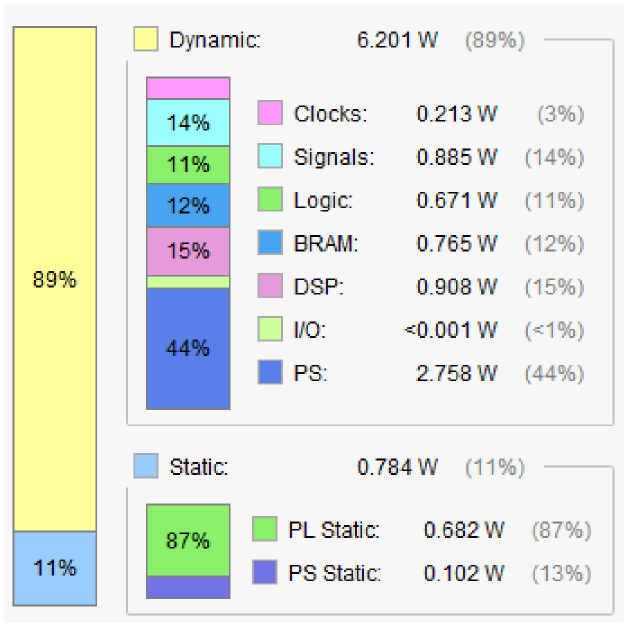


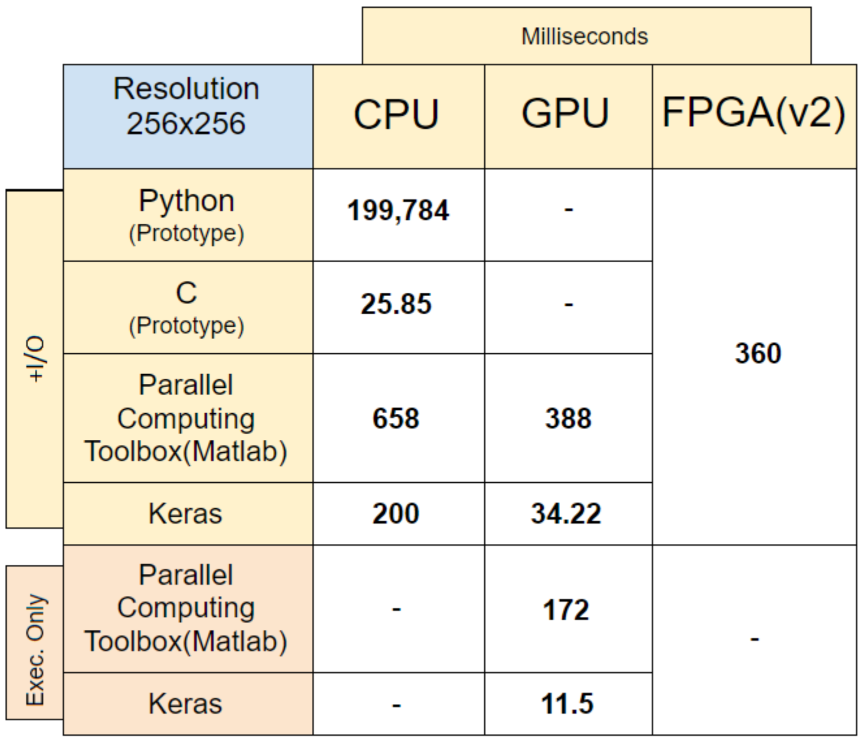
FIGURE 6.12: Overall Zynq UltraScale+ MPSoC Power Consumption - V2

Custom UNET implementations and FPGA comparison

The figure 6.13, describes the comparison between similar platforms versus the FPGA implementation. Moreover, all the CPU based platforms are used showing that FPGA can clearly compared with the fastest optimized version of CPU(Keras) that makes use of all the available CPU features and acceleration techniques. Furthermore, the zcu is compared to the GPU for Matlab(Parallel Computing Toolbox) and Keras for both raw execution time and overall latency including I/O operations. Matlab latency measurements was not stable producing wide spread between each test. The actual recorded measurements were the values with the lowest latency result(since a true cold-start matlab execution could lead to 10x worst results).

The I/O for Keras is calculated as follows:

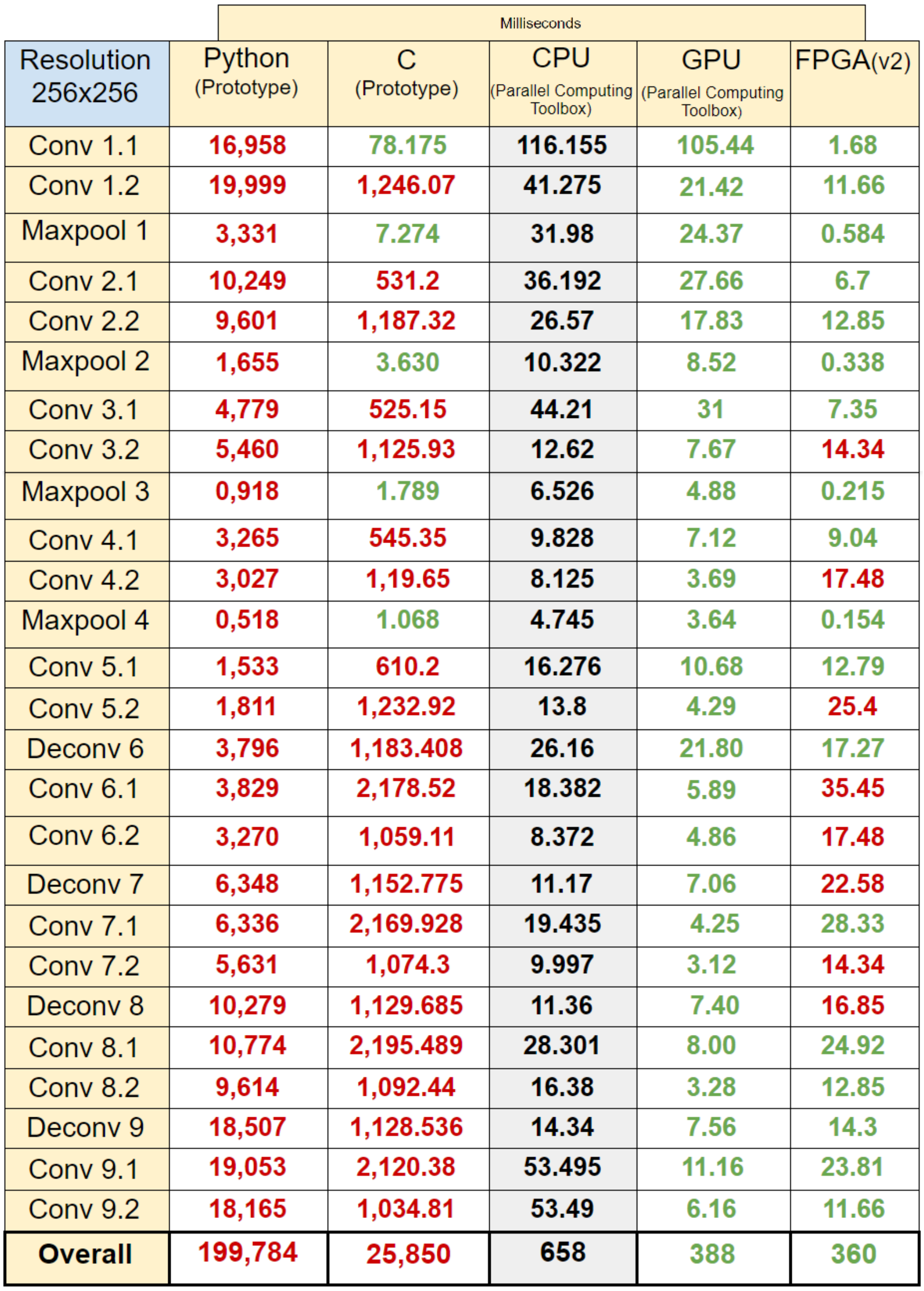
Test GPU *Maximum* transfer speed is 88GB/s (ideal transfers)
Keras needs the whole available VRAM during the UNET initialization and modeling process before the inference part, so the time takes to fill up the 2GB on the current test model is $t = \frac{1ms*2GB}{88GB} = 22.72ms$



		Milliseconds		
	Resolution 256x256	CPU	GPU	FPGA(v2)
+I/O	Python (Prototype)	199,784	-	360
	C (Prototype)	25.85	-	
	Parallel Computing Toolbox(Matlab)	658	388	
	Keras	200	34.22	
Exec. Only	Parallel Computing Toolbox(Matlab)	-	172	-
	Keras	-	11.5	

FIGURE 6.13: CPU & GPU setups vs FPGA

Below (Figure 6.14), all the layers of each category(max pool, convolution, transposed convolution) are presented and compared.
The values are calculated after multiple simulations/executions representing the average performance of each system including the I/O procedures in order each platform to fully execute the inference. A batch of one was used for the following example.



	Milliseconds				
Resolution 256x256	Python (Prototype)	C (Prototype)	CPU (Parallel Computing Toolbox)	GPU (Parallel Computing Toolbox)	FPGA(v2)
Conv 1.1	16,958	78.175	116.155	105.44	1.68
Conv 1.2	19,999	1,246.07	41.275	21.42	11.66
Maxpool 1	3,331	7.274	31.98	24.37	0.584
Conv 2.1	10,249	531.2	36.192	27.66	6.7
Conv 2.2	9,601	1,187.32	26.57	17.83	12.85
Maxpool 2	1,655	3.630	10.322	8.52	0.338
Conv 3.1	4,779	525.15	44.21	31	7.35
Conv 3.2	5,460	1,125.93	12.62	7.67	14.34
Maxpool 3	0,918	1.789	6.526	4.88	0.215
Conv 4.1	3,265	545.35	9.828	7.12	9.04
Conv 4.2	3,027	1,19.65	8.125	3.69	17.48
Maxpool 4	0,518	1.068	4.745	3.64	0.154
Conv 5.1	1,533	610.2	16.276	10.68	12.79
Conv 5.2	1,811	1,232.92	13.8	4.29	25.4
Deconv 6	3,796	1,183.408	26.16	21.80	17.27
Conv 6.1	3,829	2,178.52	18.382	5.89	35.45
Conv 6.2	3,270	1,059.11	8.372	4.86	17.48
Deconv 7	6,348	1,152.775	11.17	7.06	22.58
Conv 7.1	6,336	2,169.928	19.435	4.25	28.33
Conv 7.2	5,631	1,074.3	9.997	3.12	14.34
Deconv 8	10,279	1,129.685	11.36	7.40	16.85
Conv 8.1	10,774	2,195.489	28.301	8.00	24.92
Conv 8.2	9,614	1,092.44	16.38	3.28	12.85
Deconv 9	18,507	1,128.536	14.34	7.56	14.3
Conv 9.1	19,053	2,120.38	53.495	11.16	23.81
Conv 9.2	18,165	1,034.81	53.49	6.16	11.66
Overall	199,784	25,850	658	388	360

FIGURE 6.14: Latency measurements for 256×256 resolution in milliseconds - Architecture V2

The updated design provides lower latency comparable with the GPU implementation made by Matlab(written in CUDA). Having these minuscule latency differences is a huge first step for the UNET on embedded systems since this can be translated into a huge efficiency boost in the power sector. On the other hand, when FPGA is compared to the prototype (custom) implementations, the gap is massive.

Table 6.5 depicts the comparison results between the prototype software implementations running on single CPU core and the ZCU102 FPGA at the maximum 256×256 resolution. The efficiency and speedup values are calculated compared to the C environment(Single CPU Core - no optimizations) for batch= 1.

TABLE 6.5: Performance Analysis - Prototypes & FPGA

	Python(Numpy)	**C(Eclipse)**	**ZCU102**
Clock Frequency (MHz)	3100	3100	187
Latency (s)	199.78	25.85	0.36
Latency Speedup	0.129	1x	71.81x
Throughput (Images/s)	0.005	0.04	2.78
Throughput Speedup	0.125x	1x	71.81x
Total On-Chip Power (Watt)	8.75	8.75	6.985
Power Efficiency	1x	1x	1.25x
Energy Consumption (Joule)	1748.11	226.19	2.51
Energy Efficiency	0.129x	1x	89.95x
Images/Joule	0.0006	0.0044	0.4

A visualization of the table 6.5 is presented below:

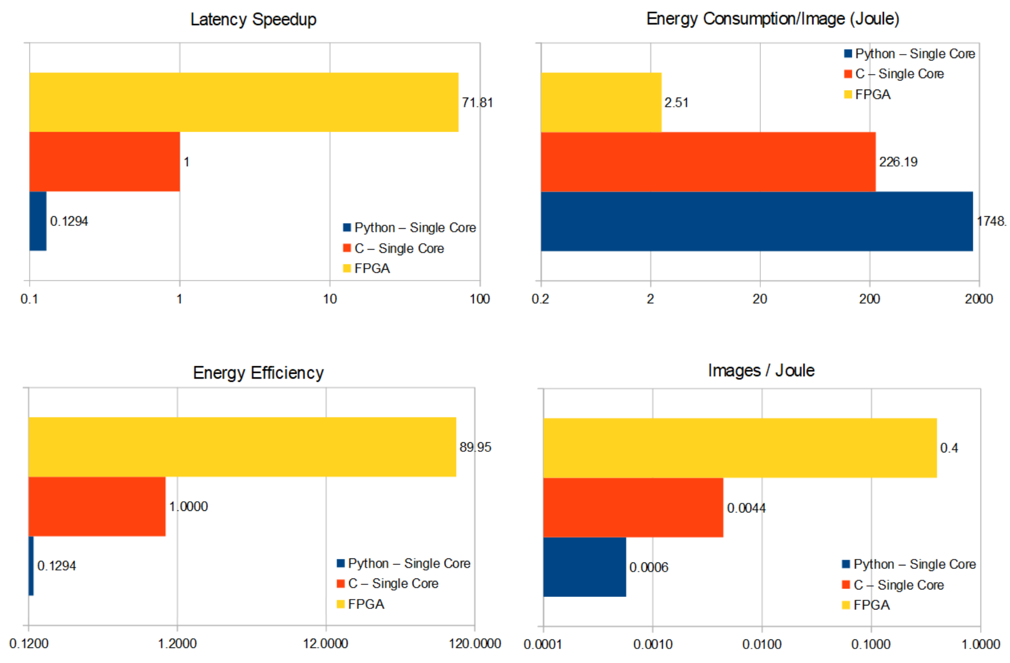


FIGURE 6.15: Latency Speedup & Energy Efficiency chart

High-Level UNET implementations and FPGA comparison

Table 6.6 extends the comparison of table 6.3 to more advanced/powerful platforms like Matlab's Parallel Computing Toolbox(PCT) and Keras-Tensorflow running on both CPU and GPU. The efficiency and speedup values are calculated compared to the CPU(Keras) environment(Multi-Core - Heavy optimizations).Latency values are calculated for batch of 1 while throughput values are selected optimally with respect to the previous throughput-per-batch figures.6.9,6.8.

TABLE 6.6: Performance analysis - High Level Environments & FPGA for the maximum supported resolution of 256 × 256

	CPU(PCT)	**GPU(PCT)**	**CPU(Keras)**	**GPU(Keras)**	**ZCU102**
Clock Frequency (MHz)	3100	1281	3100	3100	187
Latency (s)	0.66	0.39	0.2	0.0342	0.36
Latency Speedup	0.304x	0.52x	1x	5.84x	0.56
Throughput (Images/s)	4.97	4.76	6.76	33.74	2.78
Throughput Speedup	0.735x	0.7x	1x	4.99x	0.41x
Total On-Chip Power (Watt)	35	60	35	60	6.985
Power Efficiency	1x	0.583x	1x	0.583x	5.01x
Energy Consumption (Joule)	23.03	23.28	5.18	2.05	2.51
Energy Efficiency	0.225x	0.22x	1x	2.5x	2.06x
Images/Joule	0.043	0.043	0.19	0.48	0.4

The following benchmarks illustrate the above performance comparison with a simple view, including some of the most important categories.

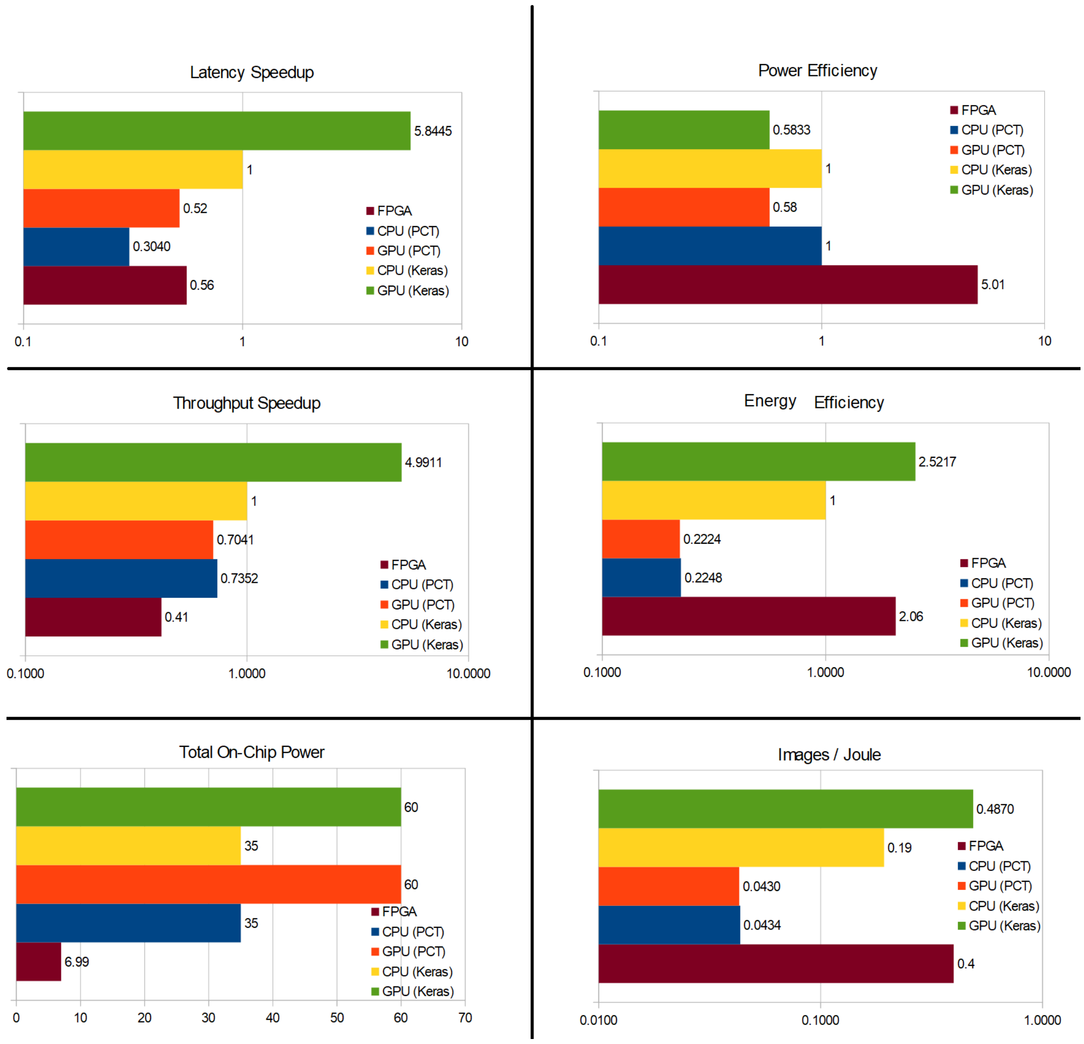


FIGURE 6.16: High level Environments & FPGA - Visualization

The latency between the High-level environments versus the FPGA is relatively small. GPU(Keras) is the most powerful device among the rest, but at the same time, it is the one with the highest power demand reaching 60 Watts. The difference between GPU(Keras) and FPGA on the latency favors GPU, which difference is notably improved for the power section(in comparison with version 1 architecture). On the other side of the coin, when comparing FPGA with any other platforms, FPGA performs well on latency comparisons and is very impressive on the power sector that dominates most of the cases for all the tests. The part where FPGA has dramatically improved is where transposed convolution and MaxPool takes place. With the current version, is possible to replace CPU/GPU setup on many machines that have limited power source running semantic segmentation. Some useful examples include the installation of the final product(embedded system) into technologies that operate with minimum power consumption(such as mini-satellite) replacing a common CPU/GPU and reaching their performance / power efficiency with just a fraction of their total on-chip power.

Chapter 7

Conclusions and Future Work

This chapter outlines and evaluates the work of this study. Moreover, headings for future work, potential expansions, and enhancements are being given.

7.1 Conclusions

In the most recent couple of years, ‘U’ Shaped architectural neural networks have ended up being used for handling complex medical image analysis, thus assisting doctors to a more accurate diagnosis process in favor of patients. This type of neural network keeps astonishing the world with unparalleled and unique effectiveness in a short period since its first appearance. As the data, complexity, and human demands are growing, high-performance computational equipment is essential to countermeasure these demands while it follows an energy-effective path. This thesis’ goal was to build a ‘bridge’ between the U-NET architecture and the actual embedded world. Multiple level interfaces were constructed, offering tools and utilities that are not available as open sources right now to assist many engineering fields by accelerating the development of this fascinating category of image semantic segmentation. On top of that, a more personal approach on hardware is made by Ch. Skoufis, who designed three individual IP cores that accelerate convolutions, transposed convolutions, and max-pooling processes that can be used by multiple FPGA platforms or even customized to reach some specified requirements. The current work supports up to 256x256 image analysis for all the programming levels: Python, C, Embedded-C. In a Python environment, the custom training scenario fully customizable by the user for further knowledge sharing, and the possibility for training implementation on a lower-level language is also possible. FPGA huge performance benefits appear when zcu102 is compared with Python (Prototype source code) and C (Single Core - Prototype Source code). Simultaneously, optimized UNET made by higher-level interfaces like TensorFlow, and Parallel computational tool by Matlab was less effective towards image/sec throughput measurements compared with the FPGA.

7.2 Future Work

The three IPs proposed by this thesis are easily expandable by design for potential use and development, providing many opportunities for its expansion and optimization by other similar U-net neural networks. The offered tools that work as assists during the training process, weight conversion between multiple levels of abstraction, and image pre-processing can also be tuned to meet the user's requirements. Some of these ideas are introduced underneath.

- The current hardware design involves using multiple DMA engines to distribute data to different IP channels so each IP can control and store the data in its local private Block RAM. Some other techniques like 'memory map' on which every memory module, such as Block RAM or DDR, has its fixed mapping and address range boosting the burst mode data transfers. Another way of fast but uncertain data transferring from DDR to IP is via the accelerator coherency port(ACP), which is a 64-bit AXI slave port located on ARM. This port is similar to HP(high performance) ports when the main difference is that the ACP port is directly connected to the Snoop control unit. As a result, whenever an action is initiated from the IP side(AXI Master), then the first thing will happen is the L1, L2 cache check for the specific data(physical address). If an instance of the data is available, they are immediately streamed back, which is a speedy and energy efficient transaction without accessing the DDR. In the case of 'miss,' the transaction will be re-directed to the DDR memory, introducing an additional latency that may degrade the system's performance. Using the ACP should be done with extreme caution.
- Layer pipeline is an excellent way of the overall latency reduction. The main idea is the data forward to the next IP as soon as they get ready. With more detail, this smart scheduling involves each IP that generates elements that will be pushing them directly from their output to the next's IP's input so(the next IP) can start processing on the way.
- Image inference is a similar pipelining trick that can improve the overall throughput of images per second. When an image's calculation is completed for a specific stage-layer, then as this image proceeds to the next computational block, the previous one remains idle until the running image reaches the end of the neural network. To avoid this kind of resource waste and to

diminish such delays is to utilize each stage by pushing a new image in the network each time the 'queue'-pipeline of the image(s) progress.

- The current work has not implemented any weight pruning; thus, it uses the accurate 32-bit floating-point (Architecture Version 1) & 32-bit fixed-point (Architecture Version 2) for a more precise evaluation that is required for the medical sector. A sophisticated quantization is also possible, while the accuracy can be preserved.

- A 'first-pass' more abstract way of optimization is currently applied to the three IPs making room for many other fine-grained tweaks and even deeper loop unrolling.

- An HLS directive called data_pack can be used to 'concatenate' multiple smaller words into one bigger and passed through the stream one significant transaction. For example, having many 8-bit word transactions that can increase overhead due to transferring protocol, four 8-bit words can be packed together, creating a 32-bit word that can be transferred at once while the unpacking can be completed at the recipient's side.

- There is a possible solution that enables Transposed convolution to function without fully loading the input image in its local Block RAM. Optimizing HLS code and mixing the algorithm with other streaming methods, the input image can be read in smaller batches allowing enough space on the PL Fabric for even larger image resolution support. The main idea is to load just the 'running' input row, which is essential for calculating the two output rows. This loading process's scheduling must be carefully implemented to ensure no additional obstruction or loading bottleneck during the intensive computational part of the transposed convolution algorithm.

- A more friendly FPGA environment supporting the conversion of the output result into multiple image formats (stored back to local disk) is also a standout feature that needs to become a reality.

- One necessary imperfection of the current semantic segmentation architecture is the lack of dynamic image resolution and format support. More specifically, a future architecture upgrade must have the ability to accept any size of resolution images(not just in the form of 2^n) that eventually will be resized and cropped during processing as they go through the network. The custom accelerators can also be upgraded into more dynamic modules from any network to match and fit in almost any design.

- The concatenation of the individual IPs is also a possible solution that can decrease the data transfer latency while accelerating the execution by using feed-forward techniques in the same IP. Similarly, with the Relu implemented in the convolution IP, the max pool IP can also be implemented in the Convolution IP, calculating 'on-the-run' the output.
- This work includes batch normalization optimization only for the Python version and training for faster results. This approach can be extended even further on Keras, including learnable weights from Keras to C and FPGA carrying the extra Batch normalization variables that can tremendously help accuracy and training times.

External Links

[1] *Machine Learning - Wikipedia.* Mar. 2020. URL: https://en.wikipedia.org/wiki/Machine_learning.

[2] *Artificial Intelligence - Wikipedia.* Mar. 2020. URL: https://en.wikipedia.org/wiki/Artificial_intelligence.

[3] *Back-propagation - Wikipedia.* Mar. 2020. URL: https://en.wikipedia.org/wiki/Backpropagation.

[4] *Big Data - Wikipedia.* Mar. 2020. URL: https://en.wikipedia.org/wiki/Big_data.

[5] *Central Processing Unit (CPU) - Wikipedia.* Mar. 2020. URL: https://en.wikipedia.org/wiki/Central_processing_unit.

[6] *AlexNet - Wikipedia.* Mar. 2020. URL: https://en.wikipedia.org/wiki/AlexNet.

[7] *VGGNet.* Mar. 2020. URL: https://medium.com/coinmonks/paper-review-of-vggnet-1st-runner-up-of-ilsvlc-2014-image-classification-d02355543a11.

[8] *GoogleNet.* Mar. 2020. URL: https://leonardoaraujosantos.gitbook.io/artificial-inteligence/machine_learning/deep_learning/googlenet.

[9] *Semantic Segmentation - Guide.* URL: https://nanonets.com/blog/semantic-image-segmentation-2020/.

[10] *Convolutional Neural Networks - Wikipedia.* Mar. 2020. URL: https://en.wikipedia.org/wiki/Convolutional_neural_network.

[12] *Nvidia Quadro RTX A6000.* Mar. 2020. URL: https://www.nvidia.com/en-us/design-visualization/quadro/rtx-a6000/.

[13] *Advanced Vector Extensions - Wikipedia.* URL: https://en.wikipedia.org/wiki/Advanced_Vector_Extensions.

[14] *TensorFlow - Official site.* URL: https://www.tensorflow.org/.

[15] *TensorFlow - Wikipedia.* URL: https://en.wikipedia.org/wiki/TensorFlow.

[16] *Parallel Computing Toolbox - Matlab.* URL: https://www.mathworks.com/products/parallel-computing.html.

[17] *Apache MXNet for deep learning.* URL: https://mxnet.apache.org/versions/1.7.0/.

[18] François Chollet et al. *Keras - Official site.* 2015. URL: https://keras.io/.

[19] *PyTorch - Official site.* URL: https://pytorch.org/.

[20] *FPGA BRAM.* URL: https://www.ni.com/documentation/en/labview-comms/latest/fpga-targets/block-memory/.

[21] *Xilinx ZCU102 User Guide - UG1182.* URL: https://www.xilinx.com/support/documentation/boards_and_kits/zcu102/ug1182-zcu102-eval-bd.pdf.

[22] *Residual Networks - Wikipedia.* URL: https://en.wikipedia.org/wiki/Residual_neural_network.

[23] *Gradient Descent Meaning - Wikipedia.* Mar. 2020. URL: https://en.wikipedia.org/wiki/Gradient_descent.

[25] *Sliding Window Intuition - Convolution.* Mar. 2020. URL: https://medium.com/ai-quest/convolutional-implementation-of-the-sliding-window-algorithm-db93a49f99a0.

[26] *Downsampling and Upsampling.* Mar. 2020. URL: https://medium.com/analytics-vidhya/downsampling-and-upsampling-of-images-demystifying-the-theory-4ca7e21db24a.

[27] *Activation Function - Wikipedia.* Mar. 2020. URL: https://en.wikipedia.org/wiki/Activation_function.

[28] *Logistic Function - Wikipedia.* Mar. 2020. URL: https://en.wikipedia.org/wiki/Logistic_function.

[29] *Hidden Layers of a Neural Network.* Mar. 2020. URL: https://deepai.org/machine-learning-glossary-and-terms/hidden-layer-machine-learning.

[30] *Deconvolution - Wikipedia.* Mar. 2020. URL: https://en.wikipedia.org/wiki/Deconvolution.

[31] *Nearest Neighbour Algorithm - Wikipedia.* Mar. 2020. URL: https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm.

[32] *Bilinear interpoleation - Wikipedia.* Mar. 2020. URL: https://en.wikipedia.org/wiki/Bilinear_interpolation.

[33] *Max-unpooling Algorithm.* Mar. 2020. URL: https://www.oreilly.com/library/view/hands-on-convolutional-neural/9781789130331/6476c4d5-19f2-455f-8590-c6f99504b7a5.xhtml.

[35] *Jaccard Index - Wikipedia.* Mar. 2020. URL: https://en.wikipedia.org/wiki/Jaccard_index.

[36] *Dice coefficient(F1 Score) - Wikipedia*. Mar. 2020. URL: https://chenriang.me/2020/05/05/f1-equal-dice-coefficient/.

[37] *Kaggle*. URL: https://www.kaggle.com/.

[38] *CIFAR-10 and CIFAR-100 - Wikipedia*. URL: https://en.wikipedia.org/wiki/CIFAR-10.

[40] *The MNIST database of handwritten digits*. URL: http://yann.lecun.com/exdb/mnist/.

[47] *Dilated-Atrous Convolution*. Mar. 2020. URL: https://towardsdatascience.com/review-dilated-convolution-semantic-segmentation-9d5a5bd768f5.

[48] *Conditional Random Field*. Mar. 2020. URL: https://en.wikipedia.org/wiki/Conditional_random_field.

[49] *NVIDIA CUDA*. URL: https://developer.nvidia.com/cuda-zone.

[50] *Open CV site*. URL: https://opencv.org/.

[51] *RMSProp Optimizer - Keras*. URL: https://keras.io/api/optimizers/rmsprop/.

[53] *Vivado Design Suite - HLx Editions*. URL: https://www.xilinx.com/products/design-tools/vivado.html.

[54] *Vivado Design Suite User Guide: High-Level Synthesis - UG902*. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf#nameddest=xApplyingOptimizationDirecti

[55] *Vitis Platform*. URL: https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html.

[62] *TGS Salt Identification Challenge - Kaggle*. Feb. 2020. URL: https://www.kaggle.com/c/tgs-salt-identification-challenge.

[63] *Boston University - Biomedical Image Library*. Feb. 2020. URL: http://www.cs.bu.edu/~betke/BiomedicalImageSegmentation/.

[64] *Intel i5 3210M Processor*. URL: https://ark.intel.com/content/www/us/en/ark/products/67355/intel-core-i5-3210m-processor-3m-cache-up-to-3-10-ghz-rpga.html.

[65] *NVIDIA GTX 750Ti Gold Sample*. URL: https://www.techpowerup.com/gpu-specs/gainward-gtx-750-ti-gs.b2798.

[66] *Eclipse CDT*. URL: https://www.eclipse.org/cdt/.

[67] *AA Bateries Specs - Wikipedia*. URL: https://en.wikipedia.org/wiki/AA_battery.

References

[11] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. “U-Net: Convolutional Networks for Biomedical Image Segmentation”. In: *CoRR* abs/1505.04597 (May 18, 2015), p. 8. arXiv: cs.CV/1505.04597 [cs.CV]. URL: https://arxiv.org/abs/1505.04597.

[34] Shuanglong Liu; Hongxiang Fan; Xinyu Niu; Hocheung Ng; Yang Chu; Wayne Luk. “Optimizing CNN-based Segmentation with Deeply Customized Convolutional and Deconvolutional Architectures on FPGA”. In: *ACM Transactions on Reconfigurable Technology and Systems* 11 (Dec. 2018), p. 22. ISSN: 1936-7406. DOI: 10.1145/3242900. URL: https://dl.acm.org/doi/10.1145/3242900.

[39] Han Xiao, Kashif Rasul, and Roland Vollgraf. “Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms”. In: *CoRR* abs/1708.07747 (Aug. 28, 2017), p. 6. arXiv: cs.LG/1708.07747 [cs.LG]. URL: https://arxiv.org/abs/1708.07747.

[41] Tobias Pohlen et al. “Full-Resolution Residual Networks for Semantic Segmentation in Street Scenes”. In: *CoRR* abs/1611.08323 (Nov. 24, 2016), p. 12. arXiv: 1611.08323. URL: http://arxiv.org/abs/1611.08323.

[42] Jonathan Long, Evan Shelhamer, and Trevor Darrell. “Fully Convolutional Networks for Semantic Segmentation”. In: *CoRR* abs/1411.4038 (Nov. 14, 2014), p. 10. arXiv: 1411.4038. URL: http://arxiv.org/abs/1411.4038.

[43] Kaiming He et al. “Mask R-CNN”. In: *CoRR* abs/1703.06870 (Mar. 20, 2017), p. 12. arXiv: 1703.06870. URL: http://arxiv.org/abs/1703.06870.

[44] Shaoqing Ren ;Kaiming He ; Ross Girshick ; Jian Sun. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.* Jan. 6, 2016. arXiv: 1506.01497. URL: http://arxiv.org/abs/1506.01497.

[45] Towaki Takikawa ; David Acuna ; Varun Jampani ; Sanja Fidler. “Gated-SCNN: Gated Shape CNNs for Semantic Segmentation”. In: *CoRR* abs/1907.05740 (July 12, 2019), p. 10. arXiv: 1907.05740. URL: http://arxiv.org/abs/1907.05740.

[46] Liang-Chieh Chen ; George Papandreou ; Iasonas Kokkinos ; Kevin Murphy ; Alan L. Yuille. *DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs.* May 12, 2017. arXiv: 1606.00915. URL: http://arxiv.org/abs/1606.00915.

[52] Yuxin Wu and Kaiming He. “Group Normalization”. In: *CoRR* abs/1803.08494 (Mar. 22, 2018), p. 10. arXiv: 1803.08494. URL: http://arxiv.org/abs/1803.08494.

[56] Xilinx. *PG021 - AXI DMA v7.1 Product Guide (v7.1).* June 2019. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf.

[57] Xilinx. *DS891 - Zynq UltraScale+ MPSoC Data Sheet: Overview.* Oct. 2019. URL: https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf.

[58] Xilinx. *MicroBlaze.* Oct. 2019. URL: https://www.xilinx.com/products/intellectual-property/microblazecore.html#overview.

[59] Xilinx. *Joint Test Action Group(JTAG) - Wikipedia.* Oct. 2019. URL: https://en.wikipedia.org/wiki/JTAG.

[60] Xilinx. *Universal Asynchronous Receiver Transmitter(UART) - Wikipedia.* Oct. 2019. URL: https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter.

[61] Xilinx. *UG1037 - Vivado Design Suite: AXI Reference Guide.* July 2017. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf.